



SAPIENZA  
UNIVERSITÀ DI ROMA

## Vulnerabilità web e attacchi informatici: Indagine sulle Tecniche di Exploitation

Facoltà Ingegneria dell'Informazione, Informatica e Statistica  
Dipartimento di Ingegneria Informatica, Automatica, Gestionale  
Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Pietro Costanzi Fantini  
Matricola 1982805

Relatore

Prof. Leonardo Querzoni

Anno Accademico 2023/2024

---

**Vulnerabilità web e attacchi informatici: Indagine sulle Tecniche di Exploitation**  
Tesi di Laurea. Sapienza – Università di Roma

© 2024 Pietro Costanzi Fantini. Tutti i diritti riservati

Questa tesi è stata composta con L<sup>A</sup>T<sub>E</sub>X e la classe Sapthesis.

Email dell'autore: [costanzifantini.1982805@studenti.uniroma1.it](mailto:costanzifantini.1982805@studenti.uniroma1.it) , [pietrocostanzi39@gmail.com](mailto:pietrocostanzi39@gmail.com)

*Ai miei genitori*  
*Alessandra Silvi e Enrico Costanzi Fantini.*

## Sommario

Questa tesi si concentra sulle vulnerabilità e sugli attacchi informatici che colpiscono le applicazioni web, con un particolare focus su tecniche di *exploitation* comunemente utilizzate dagli aggressori. Tra gli attacchi analizzati vi sono *SQL Injection*, *Cross-Site Scripting (XSS)*, *Cross-Site Request Forgery (CSRF)*, *File Inclusion*, *Command Injection* e *Insecure Direct Object Reference (IDOR)*.

Dopo una breve introduzione ai concetti fondamentali della sicurezza informatica, vengono illustrate le principali vulnerabilità sfruttate da questi attacchi, evidenziando come esse possano compromettere i principi di *Confidenzialità*, *Integrità* e *Disponibilità* (CIA) dei dati. Ogni attacco viene esaminato in dettaglio, spiegando le tecniche adottate dagli aggressori per sfruttare le debolezze dei sistemi.

La tesi include la creazione di un ambiente di sviluppo tramite *Docker*, con un *container* che ospita un server *PostgreSQL* e un altro che esegue un'applicazione web *PHP* vulnerabile, utilizzata per simulare e dimostrare gli attacchi studiati. Vengono discussi i meccanismi di difesa e le contromisure adottabili per mitigare le vulnerabilità e prevenire potenziali *exploit*.

Infine, vengono proposte riflessioni sui rischi associati alle vulnerabilità analizzate, concludendo con l'importanza di una continua valutazione e implementazione di misure di sicurezza per proteggere le applicazioni web e i dati sensibili.

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Fondamenti della sicurezza informatica . . . . .	1
1.2	Scopo della tesi . . . . .	3
1.3	Panoramica attacchi informatici . . . . .	4
<b>2</b>	<b>Metodologie di sviluppo e tecnologie</b>	<b>5</b>
2.1	Configurazione di Docker . . . . .	6
2.2	Inserimento dei dati all'interno del database . . . . .	8
2.3	Configurazione del webserver . . . . .	10
<b>3</b>	<b>SQLi</b>	<b>15</b>
3.1	Panoramica . . . . .	15
3.2	Implementazione . . . . .	18
3.3	Risultati sperimentali . . . . .	21
3.3.1	Tautologia . . . . .	21
3.3.2	Commento di fine riga . . . . .	22
3.3.3	Query piggybacked . . . . .	23
3.4	Contromisure . . . . .	24
3.4.1	Defensive coding . . . . .	24
3.4.2	Detection . . . . .	25
3.4.3	Run-time prevention . . . . .	25
<b>4</b>	<b>Cross-site Scripting</b>	<b>26</b>
4.1	Panoramica . . . . .	26
4.2	Implementazione . . . . .	28
4.3	Risultati sperimentali . . . . .	31
4.3.1	Stored XSS . . . . .	32
4.3.2	Reflected XSS . . . . .	33
4.3.3	DOM-based XSS . . . . .	34
4.4	Contromisure . . . . .	36
4.4.1	Sanitizzazione e validazione dell'input . . . . .	36
4.4.2	Content Security Policy (CSP) . . . . .	37
4.4.3	Cookie con attributi di sicurezza . . . . .	38

<b>5</b>	<b>File Inclusion</b>	<b>39</b>
5.1	Panoramica . . . . .	39
5.2	Implementazione . . . . .	41
5.3	Risultati sperimentali . . . . .	42
5.3.1	RFI . . . . .	42
5.3.2	LFI . . . . .	43
5.4	Contromisure . . . . .	44
5.4.1	Validazione dell'input . . . . .	45
5.4.2	Corretta implementazione di file di configurazione . . . . .	45
<b>6</b>	<b>Command Injection</b>	<b>47</b>
6.1	Panoramica . . . . .	47
6.2	Implementazione . . . . .	49
6.3	Risultati sperimentali . . . . .	51
6.3.1	Direct Command Injection . . . . .	51
6.3.2	Blind Command Injection . . . . .	52
6.4	Contromisure . . . . .	53
6.4.1	Controlli di validità e sanificazione dell'input . . . . .	53
6.4.2	Disabilitare l'accesso alla shell . . . . .	54
6.4.3	Utilizzare un Web Application Firewall (WAF) . . . . .	54
<b>7</b>	<b>Insecure Direct Object Reference</b>	<b>55</b>
7.1	Panoramica . . . . .	55
7.2	Implementazione . . . . .	57
7.3	Risultati sperimentali . . . . .	58
7.4	Contromisure . . . . .	60
7.4.1	Sanitizzazione dell'input . . . . .	60
7.4.2	Controllo degli accessi . . . . .	60
7.4.3	Limitazione dell'accesso per ID sensibili . . . . .	61
7.4.4	Access control list . . . . .	61
<b>8</b>	<b>Cross-Site Request Forgery</b>	<b>64</b>
8.1	Panoramica . . . . .	64
8.2	Implementazione . . . . .	66
8.3	Risultati sperimentali . . . . .	68
8.4	Contromisure . . . . .	69
8.4.1	Limitare l'applicazione di richieste GET . . . . .	69
8.4.2	Token CSRF . . . . .	70
<b>9</b>	<b>Conclusione</b>	<b>72</b>

<b>Indice</b>	<b>vi</b>
<b>Bibliografia</b>	<b>73</b>
<b>Ringraziamenti</b>	<b>75</b>

# 1 Introduzione

La sicurezza informatica rappresenta uno degli ambiti più cruciali nel contesto tecnologico attuale. Con la crescente digitalizzazione e l'adozione massiccia di soluzioni basate su Internet, la protezione dei dati e delle risorse informatiche è diventata una priorità assoluta. L'obiettivo della sicurezza informatica è quello di implementare e gestire contromisure volte a proteggere gli *asset* informatici, che sono costantemente esposti a una vasta gamma di minacce e attacchi.

In questo scenario, lo studio degli attacchi informatici e delle relative contromisure è di fondamentale importanza per prevenire danni e garantire un livello adeguato di sicurezza nelle applicazioni web e nei sistemi informatici.

## 1.1 Fondamenti della sicurezza informatica

Il *NIST Computer Security Handbook* definisce la **Sicurezza Informatica** come la protezione di un sistema informativo automatizzato, con l'obiettivo di garantire l'**integrità**, la **disponibilità** e la **riservatezza** delle sue risorse. Queste risorse includono componenti hardware, software, firmware, dati e le reti di telecomunicazione. Da questa definizione, emergono tre obiettivi chiave che costituiscono la base della sicurezza informatica:

- **Confidenzialità:** è fondamentale assicurare che le informazioni sensibili non siano divulgate ed accessibili a entità non autorizzate, permettendo agli utenti di controllare quali dati vengono raccolti e utilizzati e da chi.
- **Integrità:** è fondamentale assicurare che tutte le modifiche ai dati e programmi siano autorizzate e che il sistema sia in grado di effettuare le sue funzioni senza essere manipolato in modo illecito.
- **Disponibilità:** è essenziale garantire che il sistema funzioni in modo regolare e sia sempre accessibile agli utenti autorizzati.



Questi tre concetti formano lo standard **CIA**, che rappresenta gli obiettivi fondamentali per proteggere i dati e i sistemi informatici. Tuttavia per avere una definizione di sicurezza informatica più completa, è importante considerare altri due concetti:

- **Autenticità:** la capacità di verificare l'identità degli utenti e garantire che le informazioni trasmettere e ricevute provengano da fonti legittime.
- **Contabilità:** è importante tener traccia di tutte le azioni effettuate da un certo utente all'interno di un sistema informatico, permettendo una revisione accurata in caso di compromissione del sistema.

Inoltre, è essenziale definire chiaramente i diversi attori che operano all'interno di un sistema informatico e come essi influenzano la sua sicurezza.

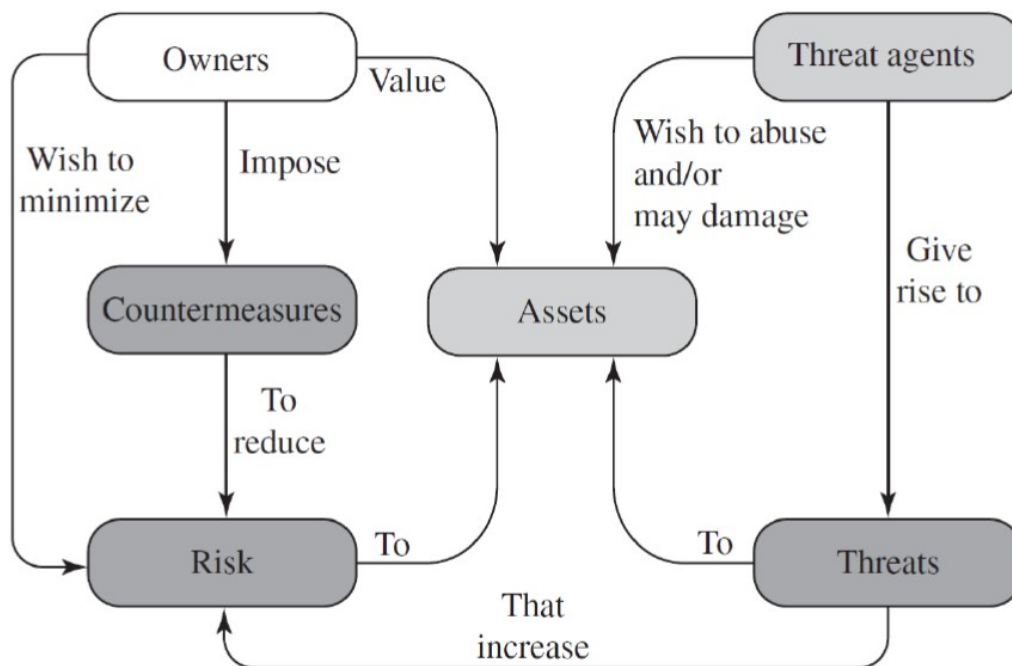


Figura 1.1. Relazioni sicurezza informatica

Nel campo della sicurezza informatica, l'obiettivo primario è di eliminare le **vulnerabilità** presenti nelle risorse di sistema. Le **minacce**, o *threats*, sono elementi che possono sfruttare queste vulnerabilità per fini malevoli. Gli **attacchi informatici** rappresentano le minacce concretizzate che, se riusciti, portano ad una violazione della sicurezza del sistema.

Un attacco è effettuato da un attaccante, noto anche come **threat agent**, e può essere classificato in due tipi principali:

- **Attacco attivo:** tentativo di alterare delle risorse di sistema o influenzarne il funzionamento.
- **Attacco passivo:** tentativo di ottenere o utilizzare delle informazioni dal sistema senza alterare le risorse di sistema.

Gli attacchi possono inoltre essere classificati in base alla loro origine:

- **Attacco insider:** effettuato da un'entità all'interno del perimetro di sicurezza del sistema informatico.
- **Attacco passivo:** effettuato da un'entità esterna al perimetro di sicurezza, da un utente non autorizzato o illecito.

Infine definiamo una **contromisura** come una tentativo per eliminare o mitigare un determinato attacco al sistema informatico. Idealmente, una contromisura dovrebbe avere un effetto **preventivo**, impedendo che un attacco possa avere successo. Tuttavia, quando la prevenzione non è possibile, è cruciale essere in grado di **rilevare** l'attacco e implementare strategie di **recupero** per limitare i danni causati.

## 1.2 Scopo della tesi

Questa tesi si propone di esplorare e analizzare una serie di attacchi informatici comunemente riscontrati in ambienti web, presentando una descrizione dettagliata delle tecniche di attacco ed un'implementazione pratica di tali attacchi attraverso l'uso di un ambiente di sviluppo sicuro (sandbox), configurato tramite **Docker**. Lo scopo è quello di illustrare come vulnerabilità apparentemente semplici possano essere sfruttate per compromettere il sistema informatico.

Inoltre questa tesi espone una serie di contromisure e *best practices* per proteggere le applicazioni da tali minacce. Oltre agli attacchi **SQLi** e **XSS**, che rappresentano alcune delle minacce più diffuse, verranno analizzati anche attacchi più specifici come **LFI**, **RFI**, e **Command Injection**.

## 1.3 Panoramica attacchi informatici

Le tecniche di attacco studiate in questa tesi sono tra le più comuni nel panorama della sicurezza web. Gli attacchi trattati sfruttano delle vulnerabilità presenti nei campi input per l'utente. Sono inoltre affrontati attacchi che invece sfruttano delle tecniche di *social engineering*, inducendo gli utenti a compiere delle azioni illecite, spesso a loro insaputa.

L'**SQL Injection** è una tecnica che permette a un attaccante di inserire o "iniettare" codice SQL dannoso all'interno di query SQL legittime, al fine di ottenere accesso non autorizzato a dati sensibili o manipolare i dati stessi. Un attacco **Cross-site Scripting (XSS)** invece consente di iniettare script maligni all'interno di pagine web visualizzate da altri utenti, sfruttando vulnerabilità nella validazione degli input. **File Inclusion**, invece, è una tecnica che sfrutta il caricamento dinamico di file all'interno di un'applicazione web, che, se mal configurata, permette all'attaccante di includere file locali (LFI) o remoti (RFI), con potenziali gravi ripercussioni per la sicurezza. Gli attacchi di **Command Injection** permettono all'attaccante di sfruttare delle vulnerabilità nei campi di input dell'utente per eseguire dei comandi di sistema all'interno dell'applicazione, accedendo così a dati sensibili del sistema stesso. L'**Insecure Direct Object Reference (IDOR)** è un tipo di attacco che sfrutta la mancanza di un adeguato controllo degli accessi, permettendo all'attaccante di accedere a oggetti o risorse senza autorizzazione. Infine il **Cross-Site Request Forgery** è un attacco che sfrutta la proprietà della *autenticazione implicita* degli utenti e utilizza delle tecniche di *social engineering* per indurli ad effettuare delle azioni lecite, ma non desiderate, per conto dell'attaccante.

Questi attacchi non solo compromettono la sicurezza delle applicazioni web, ma rappresentano anche una minaccia significativa per la privacy degli utenti e l'integrità dei sistemi. Pertanto, una parte essenziale di questa tesi sarà dedicata alle contromisure per mitigare tali rischi. Tecniche come la sanitizzazione degli input, l'implementazione di Web Application Firewall (WAF), l'uso di token anti-CSRF, e la corretta gestione della configurazione del server, rappresentano alcuni dei metodi chiave per proteggere le applicazioni dalle minacce descritte.

## 2 Metodologie di sviluppo e tecnologie

Il progetto si propone di esaminare una serie di attacchi informatici come l'SQL injection, il Cross-Site scripting, il Cross-Site Request Forgery e molti altri. Questi tipi di attacchi rappresentano una minaccia significativa per le applicazioni web, consentendo agli aggressori di compromettere le proprietà di **Confidenzialità**, **Integrità** e **Disponibilità** (CIA) dei dati.

Per dimostrare questa vulnerabilità, è stato creato un ambiente di sviluppo utilizzando Docker. In questo contesto, un container ospita un server PostgreSQL, un robusto database relazionale, mentre un altro container esegue un'applicazione web PHP. **Docker** è una piattaforma in grado di automatizzare l'**avvio**, la **configurazione** e la **gestione** di applicazioni in container. I container sono pacchetti leggeri e portatili che includono tutto ciò di cui un'applicazione ha bisogno per funzionare. Possiamo pensare a questi container come degli ambienti di sviluppo assestanti all'interno della nostra macchina.

Il progetto si compone di tre elementi principali che rendono possibile il suo compimento:

- **Docker**: configurazione dei container contenenti i servizi richiesti dalle specifiche
- **Database**: utilizzando PostgreSQL per memorizzare i nostri dati sensibili
- **Web Server**: un server Apache con PHP per eseguire l'applicazione web

## 2.1 Configurazione di Docker

Per la configurazione di **Docker** abbiamo utilizzato, in particolare, **Docker Compose**, uno strumento che consente di gestire e avviare applicazioni che richiedono più *container*, semplificando la configurazione e rendendo l'ambiente di sviluppo più strutturato. Grazie a *Docker Compose*, infatti, è possibile avviare tutti i servizi necessari con un solo comando.

```
1  services:
2
3      webserver:
4          container_name: php-webserver
5
6          build:
7              context: .
8              dockerfile: Dockerfile
9          volumes:
10             - ./app:/var/www/html
11
12          ports:
13             - 4000:80
14          depends_on:
15             - postgres
```

**Listing 2.1.** "docker-compose.yml"

Il file *docker-compose.yml* è indispensabile per definire i servizi necessari per l'applicazione. In questa sezione del file, viene definito il servizio *webserver*, il quale avrà un *container\_name* per poter essere identificato, un *Dockerfile* per la configurazione interna, e vengono specificate le porte utilizzate. Nello specifico, la porta 4000 del computer locale è mappata sulla porta 80 del container, dove è in esecuzione il web server.

```
1 FROM php:8.2-apache
2
3 RUN apt-get update \
4     && apt-get install -y libpq-dev \
5     && docker-php-ext-install pdo pdo_pgsql
6
7 COPY ./app /var/www/html
```

Listing 2.2. "Dockerfile"

Il seguente *Dockerfile* è stato definito appositamente per la configurazione del web server. In particolare, il comando **RUN** permette l'esecuzione di comandi da terminale per installare una serie di pacchetti necessari al corretto funzionamento dell'applicazione, tra cui *pdo\_pgsql* che permette la creazione di *query* direttamente al database.

```
1 postgres:
2   image: postgres:16.0
3   container_name: postgres
4
5   environment:
6     POSTGRES_DB: database
7     POSTGRES_USER: user
8     POSTGRES_PASSWORD: .UYr930Qr
9
10  ports:
11    - "5432:5432"
12
13  volumes:
14    - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

Listing 2.3. "docker-compose.yml"

Infine, questa sezione del file *docker-compose.yml* è dedicata alla configurazione del database PostgreSQL, utilizzando variabili d'ambiente e specificando i dati contenuti nel file *init.sql*. Per accedere al database, vengono inserite le credenziali predefinite per i database *POSTGRES*.

## 2.2 Inserimento dei dati all'interno del database

Per simulare l'accesso non autorizzato a dati sensibili e l'utilizzo di funzionalità riservate esclusivamente agli amministratori, abbiamo creato un database per raccogliere i dati di accesso di una serie di utenti.

Un *database* è una banca dati strutturata in grado di immagazzinare dati per una o più applicazioni. Esistono vari tipi di database; nel nostro caso utilizziamo un **database relazionale**, il quale suddivide i dati raccolti in delle tabelle, anche chiamate *relazioni*.

Ogni tabella è composta da righe e colonne. Le colonne vengono anche chiamate *attributi* mentre le righe rappresentano *istanze* specifiche di un determinato oggetto. Ad esempio se abbiamo una relazione che memorizza i dati di un utente, gli attributi di questa potranno essere *nome\_utente*, *email* e *età*; le righe, invece, rappresenteranno le istanze degli utenti salvate nel database.

Nel nostro progetto, il database relazionale salva i dati nelle tabelle definite nel file **init.sql**, che viene passato direttamente al database come specificato nel file *docker-compose.yml*.

```
DROP TABLE IF EXISTS users;

CREATE TABLE users(
    id SERIAL primary key,
    username varchar(255) NOT NULL,
    password varchar(255) NOT NULL,
    email varchar(255) NOT NULL
);

INSERT INTO users (username, password, email)
VALUES
    ('admin', 'admin', 'admin@localhost'),
    ('user', 'user', 'user@localhost'),
    ('guest', 'guest', 'guest@localhost');
```

Listing 2.4. "init.sql"

Per interagire con il database e inviare istruzioni, utilizziamo un linguaggio specifico chiamato **SQL** (*Structured Query Language*).

Inizialmente, controlliamo se la tabella **users** sia già presente, in tal caso, questa viene eliminata. Successivamente, definiamo la tabella *users* con una serie di attributi, tra cui un *id*, ovvero la chiave primaria utilizzata per identificare in modo univoco ciascuna istanza di utente nella tabella. Notiamo anche che la *password* dell'utente viene salvata in chiaro.

Eseguendo il comando per avviare i *docker container*, ovvero **docker compose up -d**, verrà avviato anche il container del database, che passerà il file **init.sql** al database sotto forma di query. A questo punto, il database conterrà i nostri dati sensibili, accessibili solo mediante un login corretto sul web server.

```
database=# select * from users;
 id | username | password |      email
-----+-----+-----+-----
  1 | admin   | admin   | admin@localhost
  2 | user    | user    | user@localhost
  3 | guest   | guest   | guest@localhost
(3 rows)
```

**Figura 2.1.** Dati all'interno del database alla creazione del database.

Un problema importante è il salvataggio delle password in chiaro. In caso di accesso non autorizzato al database, un aggressore potrebbe raccogliere tutte le credenziali degli utenti. Per mitigare questo rischio, è possibile cifrare i dati sensibili utilizzando funzioni **hash**, algoritmi che trasformano dati di qualsiasi dimensione in un output di dimensione fissa, chiamato valore **hash** o **digest**.

La sicurezza delle funzioni *hash* dipende principalmente dalla lunghezza degli *hash* prodotti. Per migliorare ulteriormente la sicurezza delle password memorizzate con queste funzioni, si può utilizzare un valore aggiuntivo chiamato **salt**, un elemento casuale di lunghezza pari a **n** bit. L'aggiunta del *salt* aumenta la complessità nello scoprire la password di un fattore pari a  $2^n$ .



## 2.3 Configurazine del webserver

Il container che viene caricato da Docker utilizza la versione 8.2 di PHP e configura un *webserver* Apache, che ospiterà la nostra applicazione web. La particolarità di quest'ultima risiede nella possibilità di simulare un gran numero di attacchi informatici. La pagina principale è composta da un ***form di login*** e da una sezione dedicata alla ***selezione dell'attacco da simulare***.

Per accedere a tutte le funzionalità del sito, è necessario effettuare il login con uno dei profili elencati nella tabella *users*. Questo permetterà di creare una **sessione PHP**, che memorizza una serie di attributi, tra cui il nome utente selezionato. L'applicazione segue una struttura gerarchica nella gestione delle pagine: vi è una cartella dedicata per ogni tipologia di attacco e si utilizza ***index.php*** per gestire tutte le chiamate ai vari moduli.

```
28 $dsn = 'pgsql:host=postgres;port=5432;dbname=database';
29 $username = 'user';
30 $password = '.UYr930Qr';
31
32 $pdo = new PDO($dsn, $username, $password);
```

Listing 2.5. "index.php"

Notiamo che, per accedere al database, definiamo una variabile locale chiamata ***\$dsn***, che specifica tre elementi fondamentali per la connessione al database: il ***nome dell'host***, la ***porta utilizzata*** e il ***nome del database*** a cui accedere. Le credenziali predefinite per l'accesso al database *PostgreSQL* vengono definite utilizzando le variabili *\$username* e *\$password*. Infine, viene creata una variabile per inizializzare il collegamento con il database, utilizzando ***PDO*** (*PHP Data Objects*), un'interfaccia in PHP che permette di accedere ai database in modo uniforme, indipendentemente dal tipo di database utilizzato.

```
38     echo "<table class='userTable'>";
39     echo "<tr>";
40         echo "<th>ID</th>";
41         echo "<th>Name</th>";
42         echo "<th>Email</th>";
43         echo "<th>Password</th>";
44     echo "</tr>";
45     $sql = "SELECT * FROM users";
46     $result = $pdo->query($sql);
47     while($row = $result->fetch()) {
48         echo "<tr>";
49             echo "<td>" . $row['id'] . "</td>";
50             echo "<td>" . $row['username'] . "</td>";
51             echo "<td>" . $row['email'] . "</td>";
52             echo "<td>" . $row['password'] . "</td>";
53         echo "</tr>";
54     }
55     echo "</table>";
```

Listing 2.6. "index.php"

Questa sezione di codice rappresenta la tabella degli utenti attualmente presenti nel database. Viene definita una variabile locale ***\$sql***, che rappresenta la *query SQL* inviata al database, con cui si richiede di selezionare tutte le istanze presenti nella tabella *users*. Successivamente, viene invocato il metodo ***query()*** dell'oggetto *PDO*, che invia la *query SQL* al database. Se la query viene eseguita correttamente, restituisce un oggetto della classe ***PDOStatement***, che rappresenta il risultato della query.

Il risultato viene poi elaborato e organizzato in una tabella mediante l'ausilio di un ciclo *while*, che itera sui dati ottenuti dalla query eseguita in precedenza.

```
59     echo "<form method='GET'>";
60         echo "<input type='text' name='username'
61             placeholder='Username'>";
62     echo "<br>";
63     echo "<br>";
64     echo "<input type='password' name='password'
65         placeholder='Password'>";
66     echo "<br>";
67     echo "<br>";
68     echo "<input type='submit' value='Login'>";
69     echo "</form>";
```

Listing 2.7. "index.php"

E' stato poi creato un form in cui è possibile inserire le credenziali di un utente presente nel database per accedere a tutte le funzionalità del sito. Notiamo che il form utilizza il metodo **GET**, il cui scopo è di raccogliere le informazioni immesse dall'utente nei campi *username* e *password*.

```
75     if (isset($_GET['username']) && isset($_GET['password'])) {
76         $username = $_GET['username'];
77         $password = $_GET['password'];
78         $sql = "SELECT * FROM users
79             WHERE username = '$username'
80             AND password = '$password'";
81         $stmt = $pdo->query($sql);
82
83         if ($stmt->rowCount() > 0) {
84             $_SESSION['username'] = $_GET['username'];
85         } else {
86             echo "<div style='margin: 3vh auto 3vh auto'>
87                 Invalid credentials</div>";
88         }
89     }
```

Listing 2.8. "index.php"

Questa parte rappresenta il controllo sulla correttezza delle credenziali inserite. Viene eseguito un controllo preliminare sulla corretta compilazione dei campi *username* e *password*.

Successivamente, interroghiamo il database per verificare la correttezza delle credenziali. La *query SQL* seleziona tutte le istanze nella tabella *users*, applicando due vincoli: il nome utente e la password devono corrispondere alle credenziali inserite. Se le credenziali sono corrette, impostiamo `$_SESSION['username']`, associando il nome utente verificato alla sessione.

```
106 <?php
107     if(isset($_SESSION['username'])) {
108         echo "<style> .check{ display:flex } </style>";
109     } else {
110         echo "<style> .check{ display:none } </style>";
111     }
112 ?>
```

Listing 2.9. "index.php"

Il comportamento predefinito dell'applicazione è quello di nascondere la sezione dedicata alla selezione dell'attacco informatico da simulare fino a quando non viene effettuato l'accesso con credenziali valide, da qui la proprietà CSS *display:none*. Una volta verificato che lo *username di sessione* sia stato impostato correttamente dopo la verifica delle credenziali, tutte le funzionalità del sito vengono rese disponibili all'utente.



Figura 2.2. Landing page del sito all'avvio dei contenitori docker

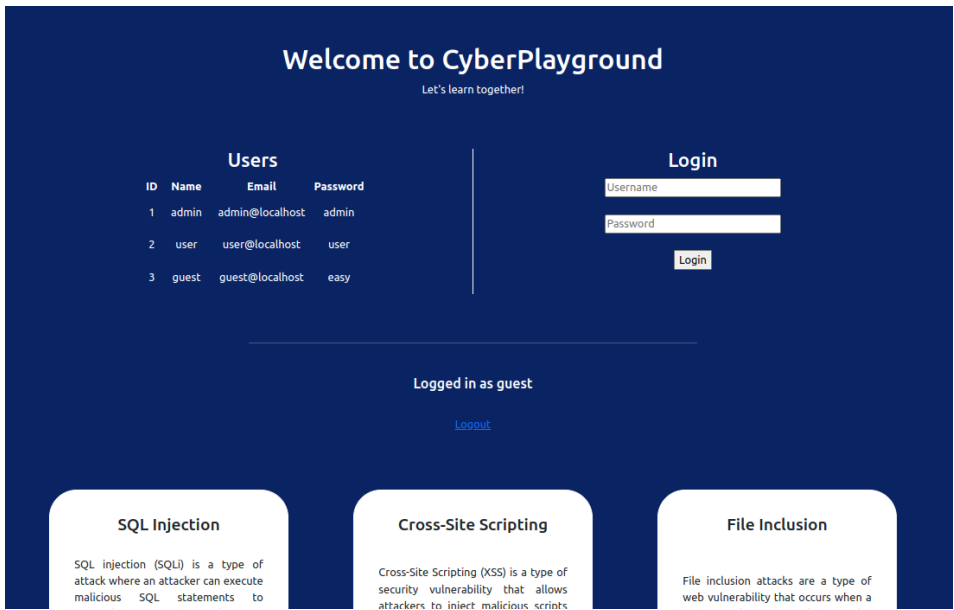


Figura 2.3. Messaggio di login dopo la verifica delle credenziali



Figura 2.4. Sezione di selezione dell'attacco da simulare

## 3 SQLi

### 3.1 Panoramica

Il linguaggio **SQL** (Structured Query Language) è al centro delle operazioni di gestione dei dati nei database relazionali. Sfruttando le sue capacità, le applicazioni interagiscono con i dati salvati nei database attraverso query SQL. I database relazionali utilizzano nello specifico delle tabelle, anche chiamate relazioni, per memorizzare dati. Le query nello specifico sono invece delle istruzioni interpretabili dal database per la manipolazione dei dati. Nel nostro caso, abbiamo scelto PostgreSQL come database per la sua conformità agli standard SQL e per il supporto a estensioni avanzate.

Gli attacchi **SQL Injection** avvengono quando un utente malintenzionato inserisce input malevoli in campi destinati alle query SQL. Questo può permettere all'attaccante di manipolare le query, ottenere informazioni riservate o compromettere l'integrità dei dati. A differenza della prevalenza di pagine web statiche usate in passato, la maggior parte delle pagine web attualmente utilizzano componenti e contenuti dinamici. Queste informazioni dinamiche vengono spesso trasferite da e verso **database back-end** che contengono una grande quantità di dati sensibili. Gli obiettivi più comuni degli aggressori sono quelli di cancellare, estrarre o modificare i dati salvati sui database.

Questi tipi di attacchi sfruttano vulnerabilità all'interno del database. Un aggressore che usa questa forma di attacco è in grado di estrarre o manipolare i dati dell'applicazione web. Solitamente questi attacchi hanno successo nel caso in cui l'input dell'utente all'interno dell'applicazione non è filtrato e controllato correttamente, permettendo all'aggressore di sfruttare delle proprietà del linguaggio SQL per effettuare delle funzionalità riservate agli amministratori dell'applicazione web.

Possiamo caratterizzare gli attacchi SQLi in base al loro vettore di attacco e al tipo di attacco. I principali vettori di attacco sono i seguenti:

- ***Inpute dell'utente***: in questo caso, l'aggressore è in grado di inserire dei comandi SQL andando a sfruttare delle vulnerabilità solitamente presenti nei form di input di un sito.
- ***Variabili del server***: le variabili del server sono un insieme di variabili che contengono intestazioni HTTP, intestazioni del protocollo di rete e variabili d'ambiente. Nel caso in cui queste variabili vengano inserite in un database senza eseguire un controllo, ciò potrebbe creare una vulnerabilità di SQL injection. Poiché un aggressore può facilmente falsificare i valori presenti nelle intestazioni HTTP e di rete, possono sfruttare questa vulnerabilità inserendo dati direttamente nelle intestazioni.
- ***Second-order injection***: questo si verifica quando i meccanismi di prevenzione di attacchi SQL injection non sono del tutto efficaci. Utilizzando questo vettore di attacco, un aggressore può sfruttare i dati già presenti nel database per effettuare un attacco SQL injection.
- ***Cookies***: Quando un utente esegue l'accesso ad un'applicazione web, i cookie possono essere utilizzati per ripristinare lo stato di informazioni dell'utente, evitando di dover accedere nuovamente nel caso si chiuda l'applicazione. Poiché il client ha il controllo sui cookie, un attaccante potrebbe modificarli in modo tale che, quando il server dell'applicazione costruisce una query SQL basata sul contenuto del cookie, la struttura e la funzione della query vengano alterate.

Per la tipologia di attacco, si possono suddividere in tre categorie:

- ***In-band***: questo tipo di attacco utilizza lo stesso canale di comunicazione sia per iniettare codice SQL sia per ricevere i risultati.
- ***Inference***: questo tipo di attacco non ha l'obiettivo di modificare direttamente i dati presenti nel database, ma mira piuttosto a raccogliere informazioni sul database stesso, al fine di pianificare un attacco più mirato e specifico su quel sistema.
- ***Out-of-band***: in questo tipo di attacco, i dati vengono recuperati utilizzando un canale diverso (ad esempio, viene generata un'email con i risultati della query e inviata al tester).

Gli attacchi *in-band* a loro volta possono verificarsi in più modi, tra cui:

- **Tautologia:** Inserimento di condizioni sempre vere per ottenere accesso non autorizzato.
- **Commenti di fine riga:** Utilizzo di commenti per disattivare parti delle query originali.
- **Query piggybacked:** Inserimento di query aggiuntive per alterare il comportamento del database.

Questo progetto esplora dettagliatamente come questi attacchi in particolare possono essere eseguiti e come mitigarli efficacemente attraverso *best practices* di sicurezza informatica e buone pratiche di sviluppo delle applicazioni.

Gli attacchi *inference* possono anch'essi verificarsi in più modi, tra cui:

- **Query SQL errate:** spesso i messaggi di errore restituiti server applicativi sono molto descrittivi e possono dare delle informazioni nello specifico del sistema utilizzato. Un aggressore può sfruttare questa vulnerabilità per reperire delle informazioni importanti in modo da effettuare degli attacchi più mirati per quel tipo di sistema.
- **Blind SQL injection:** in questo caso gli aggressori sono in grado di dedurre i dati presenti in un sistema di database anche quando il sistema è sufficientemente sicuro da non mostrare informazioni di errore all'attaccante. In questo tipo di attacco, l'aggressore invia al server delle istruzioni condizionali, se l'istruzione iniettata restituisce vero, il sito continua a funzionare normalmente. Al contrario, se l'istruzione restituisce falso, pur non visualizzando un messaggio di errore descrittivo, la pagina si differenzia significativamente da quella che appare in condizioni normali."



## 3.2 Implementazione

Per simulare questa forma di attacco, è stato sviluppato un modulo apposito in cui l'utente può simulare delle *SQL injections*, in particolare di tipo *in-band*.

```
16     $dsn = 'pgsql:host=postgres;port=5432;dbname=database';
17     $username = 'user';
18     $password = '.UYr930Qr';
19
20     $pdo = new PDO($dsn, $username, $password);
```

Listing 3.1. "../sqli/app.php"

Essendo necessaria la verifica delle credenziali inserite per simulare l'accesso ad un'area riservata del sito, viene effettuata la connessione al database. La procedura è la stessa vista in precedenza, in cui si utilizza la *porta 5432* e le credenziali predefinite per l'accesso al database.

```
16     echo "<form method='GET'>";
17         echo "<input type='text' name='username'
18             placeholder='Username'>";
19         echo "<br>";
20         echo "<br>";
21         echo "<input type='password' name='password'
22             placeholder='Password'>";
23         echo "<br>";
24         echo "<br>";
25         echo "<input type='submit' value='Login'>";
26     echo "</form>";
```

Listing 3.2. "../sqli/app.php"

Anche in questo caso si è utilizzato un form con metodo **GET** per poter prelevare le informazioni inserite dall'utente.

**SQL Injection**


SQL injection (SQLi) is a type of attack where an attacker can execute malicious SQL statements to manipulate a web application's database. This can lead to unauthorized access to sensitive data, modification of data, or even deletion of the entire database. SQLi attacks are categorized into different types, including in-band (same channel), inferential (blind), and out-of-band.

**Login**

Username

Password

[Create Account](#)



**Figura 3.1.** *Form di login iniziale*

```
41  if (isset($_GET['username']) && isset($_GET['password'])) {  
42      $username = $_GET['username'];  
43      $password = $_GET['password'];  
44      $sql = "SELECT * FROM users  
45      WHERE username = '$username'  
46      AND password = '$password'";  
47      $stmt = $pdo->query($sql);  
48  
49      if ($stmt->rowCount() > 0) {  
50          echo "<div class='container'  
51          style='text-align:center;  
52          margin: 3vh auto 3vh auto'>";  
53          echo "<h2>Logged in</h2>";  
54          echo "<p>Welcome , $username</p>";  
55          echo "</div>";  
56          ...  
}
```

**Listing 3.3.** `"../sql/app.php"`

In questa sezione è stato implementato un controllo per verificare che i campi *username* e *password* siano stato compilati correttamente dall'utente. Successivamente, si verifica la validità di tali credenziali confrontandole con quelle presenti nel database. Se risultano corrette, viene simulato l'accesso a una sezione riservata agli utenti autorizzati.

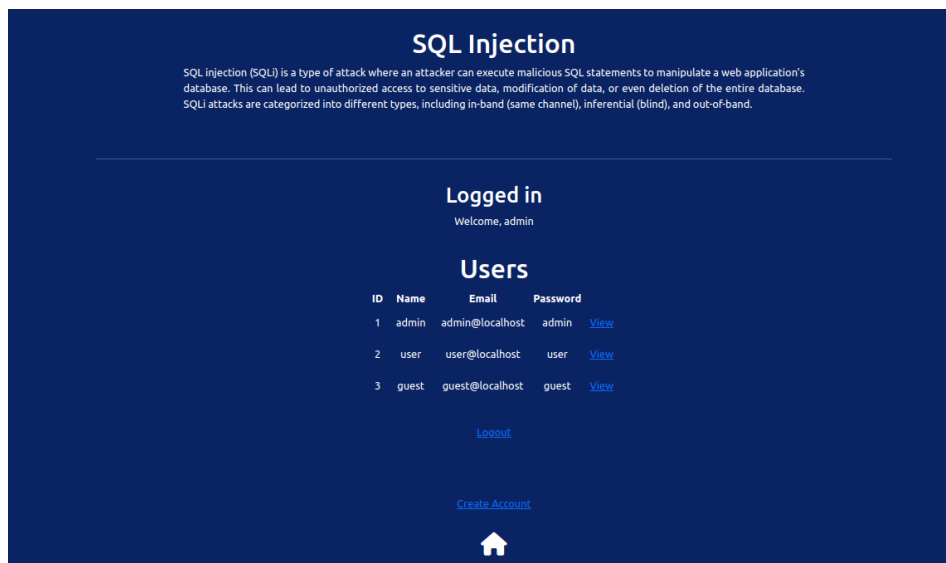


Figura 3.2. Portale riservato ad utenti autorizzati

Per gestire l'eventualità in cui l'utente desideri creare un nuovo account nel database, è stata sviluppata una sezione dedicata alla creazione di un nuovo account.

```
38     if ($_SERVER['REQUEST_METHOD'] == 'POST') {
39         $username = $_POST['username'];
40         $password = $_POST['password'];
41         $email = $_POST['email'];
42         $sql = "INSERT INTO users
43             (username, email, password)
44             VALUES ('$username', '$email', '$password')";
45
46         $stmt = $pdo->exec($sql);
47     }
```

Listing 3.4. "../sql/create.php"

In questo caso, notiamo che il form di registrazione utilizza il metodo **POST**, il quale inserisce i dati direttamente nel database. Dopo aver verificato il metodo corretto, si inseriscono i dati nel database con una *query SQL* specifica, prendendo i dati forniti dall'utente.

### 3.3 Risultati sperimentali

Per dimostrare la vulnerabilità del sistema, andiamo ad analizzare nel dettaglio la query che eseguiamo per richiedere i dati dal database. Analizziamo quindi il file `../sql/app.php` contenente la logica dietro il login al portale e andiamo a studiare nel dettaglio la richiesta che fa al database:

```
41     if (isset($_GET['username']) && isset($_GET['password'])) {  
42         $username = $_GET['username'];  
43         $password = $_GET['password'];  
44         $sql = "SELECT * FROM users  
45             WHERE username = '$username'  
46             AND password = '$password'";  
47         $stmt = $pdo->query($sql);
```

Listing 3.5. `../sql/app.php`

#### 3.3.1 Tautologia

Notiamo che la query passata al database prende in considerazione il `$username` che inseriamo nel form di login della pagina iniziale. Questa caratteristica può essere sfruttata passando degli input malevoli nella casella di testo di login del tipo:

```
admin' OR '1'='1
```

Questo fa in modo che il database riceva una query alterata del tipo:

```
1     $sql = "SELECT * FROM users WHERE  
2     username = 'admin' OR '1' = '1'  
3     AND password = '$password'";
```

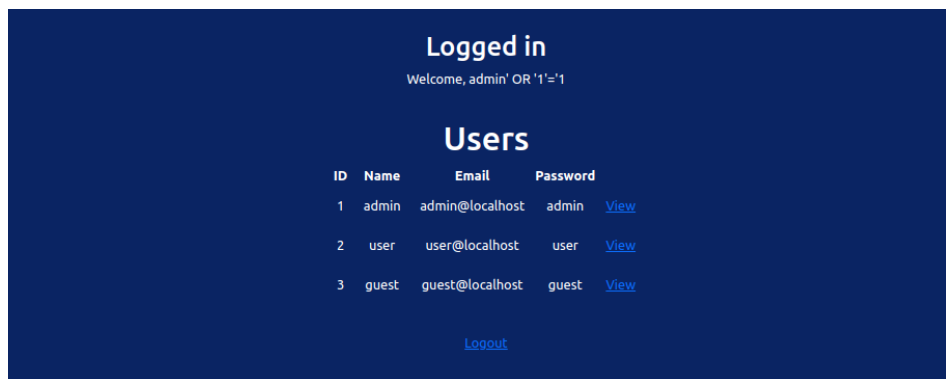


Figura 3.3. Form di login bypassato da una tautologia

Questo è un esempio di una **tautologia** e il controllo della password verrà ignorato, essendo la clausola **WHERE** sempre verificata. Un attacco di questo tipo permette all'aggressore di accedere nella schermata di login, pur non conoscendo la password dell'utente.

### 3.3.2 Commento di fine riga

Un altro esempio di input malevolo che può portare ad accessi impropri:

admin' OR 1=1 –

Questo input rappresenta il SQL injection di categoria **commento di fine riga**. Lo scopo in questo caso è commentare il controllo che avviene sulla password dell'utente nella query diretta al database. Questo attacco in particolare sfrutta la proprietà del linguaggio SQL dove per inserire un commento è necessario inserire `--`. La query risulterà essere:

```
1 $sql = "SELECT * FROM users WHERE
2 username = 'admin' OR 1=1 -- AND password = '$password'";
```

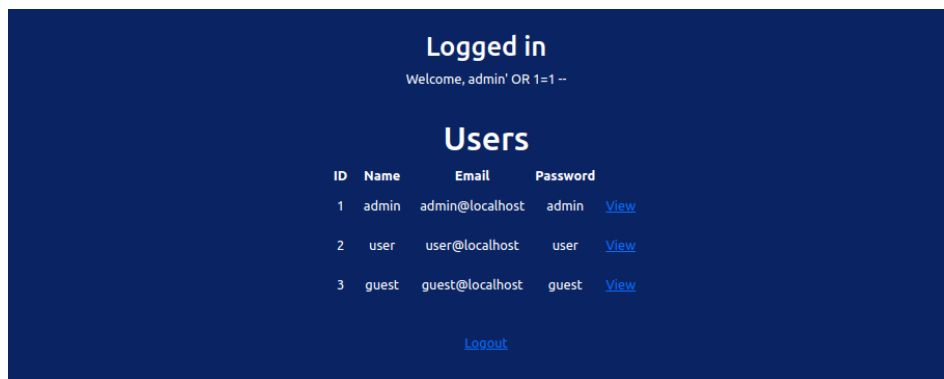


Figura 3.4. Accesso consentito con commento di fine riga

### 3.3.3 Query piggybacked

L'ultimo tipo di SQL injection che osserviamo sarà la **query piggybacked**. Questa consiste nel concatenare delle query assieme a quella già presente per la selezione di dati dal database. L'aggressore potrà quindi inserire una query malevola concatenata a quella lecita in modo da alterare o accedere ai dati sensibili:

```
admin'; DROP TABLE users; --
```

Il database quindi riceverà una query del tipo:

```
1 $sql = "SELECT * FROM users WHERE
2 username = 'admin'; DROP TABLE users;
3 -- AND password = '$password'";
```

Alla ricezione di questa query il database andrà a verificare la presenza dello username e andrà successivamente a eseguire la **DROP TABLE users** in quanto query concatenata. Anche in questo caso l'attacco sfrutta una caratteristica del linguaggio SQL dove per concatenare delle istruzioni basta separarle con un punto e virgola.

## 3.4 Contromisure

Abbiamo dimostrato come un'applicazione web vulnerabile possa essere compromessa tramite un attacco di tipo SQL injection, compromettendo le proprietà di **confidenzialità**, **integrità** e **accessibilità** dei dati. Questo tipo di attacco è particolarmente pericoloso per le applicazioni web che utilizzano database contenenti dati sensibili. Tuttavia, esistono diverse contromisure per mitigare il problema. Queste contromisure possono essere suddivise in tre categorie:

- **Defensive coding**
- **Detection**
- **Run-time prevention**

### 3.4.1 Defensive coding

Molti attacchi di questo tipo hanno successo a causa di uno scarso sviluppo del codice di controllo nell'inserimento dell'input dell'utente e dell'inadeguata metodologia di passaggio delle query al database.

#### 3.4.1.1 Defensive coding nel pratico

Lo sviluppatore di un'applicazione web può implementare una serie di controlli sull'input dell'utente, come ad esempio il controllo del tipo di input inserito nella casella di testo. Se nel campo di testo non sono previsti numeri, l'inserimento di questi porterà a un errore.

#### 3.4.1.2 Parametrizzare i dati nelle query

Un altro approccio che lo sviluppatore può utilizzare è quello di passare al database delle query con dei parametri contenenti dei dati segnaposto (placeholder), i quali verranno sostituiti con i dati effettivi in un secondo momento. Questo metodo impedisce l'esecuzione diretta di query malevoli.

```
1      $stmt = $conn->prepare("SELECT * FROM users WHERE
2      username = ? AND password = ?");
3
4      // Bind dei parametri
5      $stmt->bind_param("ss", $username, $password);
6
7      $stmt->execute();
```

### 3.4.2 Detection

Esistono una serie di contromisure in grado di individuare possibili vulnerabilità a questi attacchi all'interno delle applicazioni web.

#### 3.4.2.1 Studio di comportamenti anomali

Questo approccio ha lo scopo di memorizzare uno schema comportamentale nominale dell'utilizzo dell'applicazione e individuare possibili comportamenti anomali da parte degli utenti. Solitamente, tali sistemi devono attraversare una fase di *training* in cui memorizzano lo schema comportamentale nominale dell'applicazione, per poi implementare la *detection phase* dei comportamenti anomali.

#### 3.4.2.2 Analisi del codice

Esistono degli strumenti appositi per individuare potenziali vulnerabilità all'interno di sistemi informatici esposti a questo tipo di attacchi.

### 3.4.3 Run-time prevention

Esistono delle tecniche sviluppate come contromisure agli attacchi SQL injection, capaci di analizzare le query nel momento della loro esecuzione e scartare eventuali query non conformi agli standard dettati dal sistema.



## 4 Cross-site Scripting

### 4.1 Panoramica

Gli attacchi *Cross-site scripting (XSS)* fanno parte di una categoria di attacchi informatici che mirano a sfruttare vulnerabilità presenti nei campi di input forniti dall'utente. In particolare, un aggressore può inserire uno script malevolo all'interno del contenuto HTML di una pagina attendibile, che verrà successivamente eseguito dal browser dell'utente vittima.

Il principale problema legato a questo tipo di attacco risiede nel presupposto che tutto il contenuto di una pagina web sia sicuro. Di conseguenza, il browser non esegue controlli specifici sugli script presenti, permettendo agli aggressori di inserire codice malevolo che viene eseguito senza ulteriori verifiche.

Gli script inseriti dagli aggressori mirano generalmente a intercettare dati sensibili, come ad esempio i *cookie di sessione*, con l'obiettivo di ottenere le credenziali di accesso dell'utente. Solitamente i siti web più vulnerabili sono quelli che accettano l'input dell'utente, ad esempio con barre di ricerca, caselle per commenti o moduli di accesso. L'aggressore allega il codice malevolo al sito web legittimo, in tal modo da far eseguire ai *browser* degli utenti il codice malevolo ogni volta che il sito viene caricato.

A seconda di come il *payload* viene inserito, è possibile che esso non sia effettivamente presente nel codice della pagina web, ma rappresenti un elemento transitorio che appare come parte del sito solo durante l'esecuzione dell'attacco. Questo può dare l'impressione che un sito web legittimo sia stato compromesso, anche se in realtà non è stato effettivamente violato.

La particolarità di questi attacchi risiede nel fatto che l'esecuzione del *payload* potrebbe essere attivata automaticamente quando la pagina viene caricata o quando un utente posiziona il puntatore su elementi specifici della pagina, come ad esempio collegamenti ipertestuali, messaggi o anche email. Esistono tre principali categorie di attacco XSS, ciascuna caratterizzata da modalità differenti di inserimento del codice malevolo nelle pagine:

- **Reflected cross-site scripting (reflected XSS):** si tratta della tipologia di attacco XSS più comune. Consiste nell'inserire il *payload* malevolo direttamente nella richiesta inviata al server web. Successivamente, il server "riflette" il payload, inserendolo nella risposta HTTP inviata al browser dell'utente. Gli attaccanti utilizzano tecniche come link malevoli, email di *phishing* e altre forme di ingegneria sociale per indurre la vittima a inviare una richiesta contenente il payload. Una volta ricevuta la risposta dal server, il payload dell'aggressore viene eseguito nel browser della vittima, compromettendone la sicurezza.
- **Stored cross-site scripting:** Questa tipologia di attacco è considerata tra le più dannose. Si verifica quando l'input fornito da un utente viene archiviato in una pagina web per essere successivamente visualizzato da altri utenti. Tipicamente, questo tipo di attacco si riscontra nei forum di messaggi, nelle sezioni dei commenti sui blog, nei profili utente e nei campi relativi al nome utente. Un aggressore sfrutta tali vulnerabilità inserendo *payload XSS* nelle pagine con una maggiore affluenza di utenti, con l'obiettivo di coinvolgere il maggior numero di persone possibile. Quando un utente vittima accede alla pagina contenente il payload malevolo, il browser dell'utente esegue il payload, causando potenziali danni.
- **DOM-based cross-site scripting:** questa tipologia di attacco, al contrario degli altri attacchi XSS, sfrutta delle vulnerabilità presenti direttamente all'interno del **DOM** (Document Object Model), anziché nel codice HTML della pagina. In particolare con questo attacco il codice HTML sorgente e la risposta del server restano inalterati. Di conseguenza, il *payload* non compare nella risposta HTTP e può essere rilevato solo durante il *runtime* o analizzando direttamente il **DOM** della pagina.

## 4.2 Implementazione

Per simulare gli attacchi *cross-site scripting*, è stato implementato un form vulnerabile, che consente agli utenti di inserire un proprio commento. Per mantenere traccia di tutti i commenti degli utenti, è stata creata la relazione **comments** all'interno del database, permettendo così di registrare e gestire i vari contributi degli utenti.

```
CREATE TABLE comments (  
    id SERIAL primary key,  
    username varchar(255) NOT NULL,  
    comment text NOT NULL  
);
```

Listing 4.1. "init.sql"

```
28     $dsn = 'pgsql:host=postgres;port=5432;dbname=database';  
29     $username = 'user';  
30     $password = '.UYr930Qr';  
31  
32     $pdo = new PDO($dsn, $username, $password);
```

Listing 4.2. "../xss/app.php"

Inizialmente, si stabilisce la connessione al database, poiché sarà necessario accedere alle informazioni relative ai commenti inseriti.

```
34     if(isset($_SESSION['username'])) {  
35         $user = $_SESSION['username'];  
36         echo "<form method='POST'>";  
37         echo "<h2 style='color:white;'>Welcome ,  
38             ".$user."!</h2>";  
39         echo "<form method='POST'>";  
40         echo "<textarea name='comment' "  
41             "placeholder='Enter your comment'></textarea>";  
42         echo "<br><br>";  
43         echo "<input type='submit' value='Submit Comment'>";  
44         echo "</form>";
```

Listing 4.3. "../xss/app.php"

Una volta effettuata la connessione, viene reso disponibile un form all'utente, con il metodo **POST**, per consentire l'invio delle informazioni compilate al database. Il campo relativo al nome utente viene automaticamente valorizzato con il nome dell'utente attualmente *loggato*, grazie alla variabile di sessione.

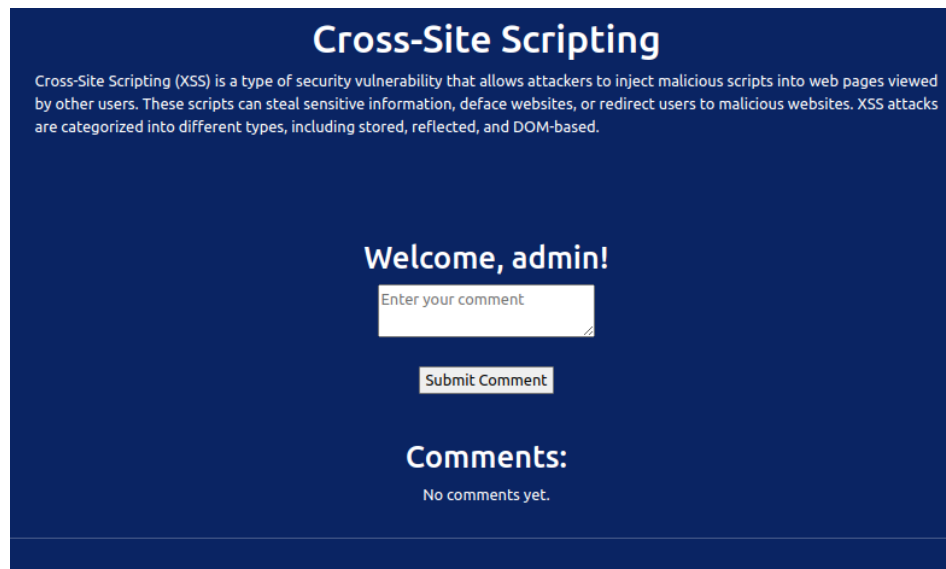


Figura 4.1. Form di inserimento commenti

```
44     if (isset($_POST['comment'])) {  
45         $user_comment = $user;  
46         $comment = $_POST['comment'];  
47  
48         $sql = "INSERT INTO comments (username, comment)  
49             VALUES ('$user', '$comment')";  
50         $stmt = $pdo->exec($sql);  
51     }
```

Listing 4.4. "../xss/app.php"

Successivamente, si verifica che il commento sia stato effettivamente inserito dall'utente e viene eseguita una *query SQL* per memorizzare i dati nel database. Questa query ha il compito di inserire una nuova istanza all'interno della tabella *comments*.

```
61     $sql = "SELECT * FROM comments";
62     $stmt = $pdo->query($sql);
63
64     if($stmt->rowCount() == 0) {
65         echo "<p>No comments yet.</p>";
66     }else{
67         while ($row = $stmt->fetch()) {
68             $usr = $row['username'];
69             $cmt = $row['comment'];
70
71             echo "<p><strong>$usr:</strong> $cmt</p>";
72         }
73     }
```

Listing 4.5. "../xss/app.php"

Infine, vengono visualizzati tutti i commenti inseriti dagli utenti attraverso una *query SQL* di tipo *select*. I commenti vengono mostrati in chiaro, senza applicare alcun filtro sull'input dell'utente, il che espone l'applicazione a potenziali attacchi XSS. Nel caso in cui non siano presenti commenti, viene visualizzato un messaggio informativo.

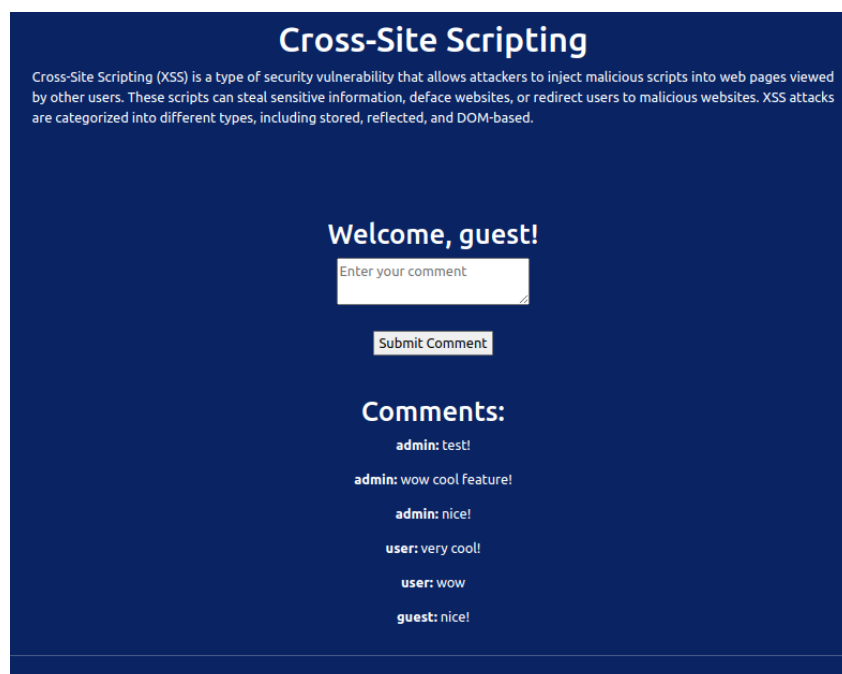


Figura 4.2. Commenti inseriti dai vari utenti

## 4.3 Risultati sperimentali

Per dimostrare la vulnerabilità del sistema, analizziamo due aspetti chiave: l'*inserimento* dei commenti nel database e la *stampa* dei commenti degli utenti. Poiché non vengono eseguiti controlli sull'*input*, un attaccante è in grado di inserire codice malevolo sotto forma di script direttamente nel campo di testo.

```
44     if (isset($_POST['comment'])) {
45         $user_comment = $user;
46         $comment = $_POST['comment'];
47
48         $sql = "INSERT INTO comments (username, comment)
49             VALUES ('$user', '$comment')";
50         $stmt = $pdo->exec($sql);
51     }
```

Listing 4.6. "../xss/app.php - inserimento commenti"

```
77     $sql = "SELECT * FROM comments";
78     $stmt = $pdo->query($sql);
79
80     if($stmt->rowCount() == 0) {
81         echo "<p>No comments yet.</p>";
82     }else{
83         while ($row = $stmt->fetch()) {
84             $usr = $row['username'];
85             $cmt = $row['comment'];
86
87             echo "<p><strong>$usr:</strong> $cmt</p>";
88         }
89     }
```

Listing 4.7. "../xss/app.php - stampa commenti"

### 4.3.1 Stored XSS

Come menzionato in precedenza, questo tipo di attacco si verifica quando l'input fornito da un utente viene archiviato in una pagina web per essere successivamente visualizzato da altri utenti. In questo caso, abbiamo simulato una sezione commenti di una pagina affidabile. Poiché l'input dell'utente non viene sanificato, l'attaccante può inserire un *payload dannoso* direttamente in un commento.

Quando un altro utente visualizza la pagina dei commenti, il payload viene eseguito nel browser della vittima. Ad esempio, l'attaccante potrebbe inserire uno script del tipo:

```
<script>alert('evil payload!')</script>
```

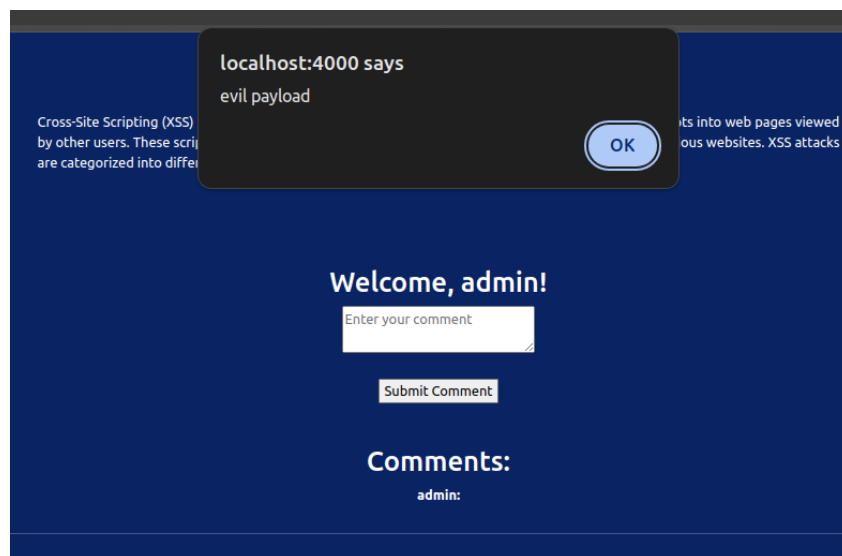


Figura 4.3. Esecuzione payload

In questo modo, il prossimo utente che aprirà la pagina eseguirà automaticamente il codice *JavaScript*.

### 4.3.2 Reflected XSS

In questo tipo di attacco, il payload malevolo viene inserito direttamente nella richiesta inviata al server web. Successivamente, il server "riflette" il payload, includendolo nella risposta HTTP inviata al browser dell'utente. Per simulare ciò, abbiamo implementato un form con metodo *GET* che riceve informazioni dall'utente, per poi visualizzarle direttamente nella pagina.

```
26 <form method="GET">
27   <input type="text" name="search"
28     placeholder="Search...">
29   <br><br>
30   <input type="submit" value="Search">
31 </form>
32
33 <?php
34   if (isset($_GET['search'])) {
35     $search = $_GET['search'];
36     echo "<p>Result: <strong>$search</strong></p>";
37   }
38 ?>
```

Listing 4.8. "../xss/app.php"

Figura 4.4. Form con casella di testo con dati da prelevare



Anche in questo caso, l'attaccante può inserire un payload dannoso nella casella di testo, che verrà eseguito dal browser quando la pagina viene caricata.

```
<script>alert('new test!')</script>
```

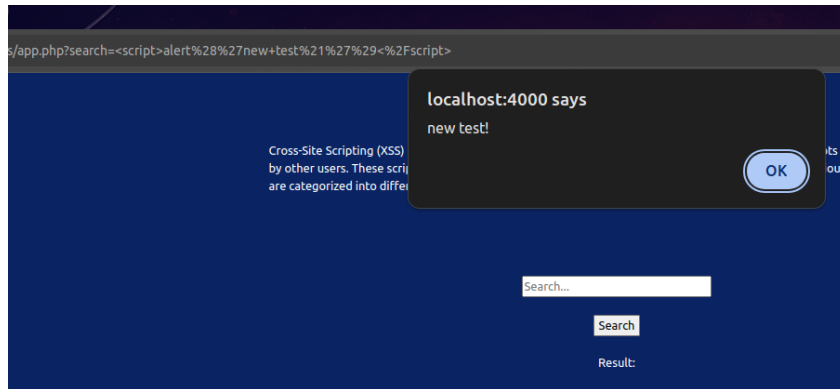


Figura 4.5. Payload eseguito dalla casella di testo e inserito nella richiesta HTTP

### 4.3.3 DOM-based XSS

Questa tipologia di attacco, a differenza degli altri attacchi XSS, sfrutta vulnerabilità presenti direttamente nel DOM (Document Object Model), anziché nel codice HTML della pagina. Per simulare questo tipo di attacco, abbiamo implementato uno script che gestisce l'input nel campo di ricerca (*search*) fornito dall'utente.

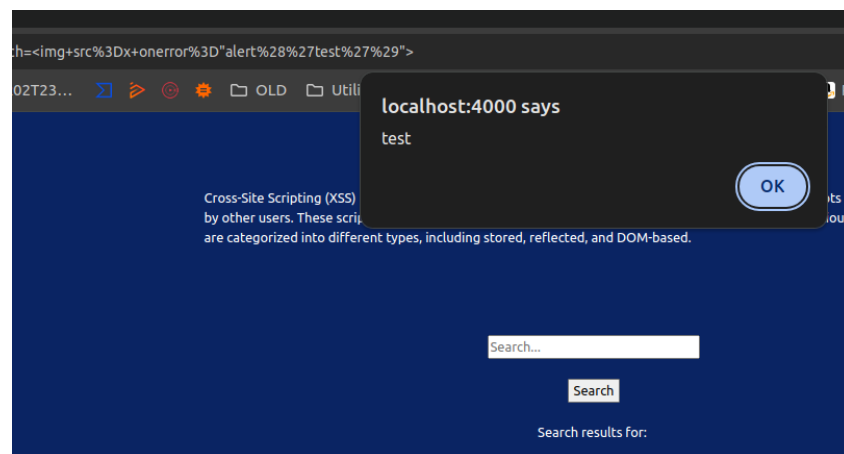
```
104 <script>
105     let urlParams =
106     new URLSearchParams(window.location.search);
107     let searchTerm = urlParams.get('search');
108
109     if (searchTerm) {
110         document.getElementById('results').innerHTML =
111         "Search results for: " + searchTerm;
112     }
113 </script>
```

Listing 4.9. "../xss/app.php"

In particolare, questo script estrae i parametri della richiesta inviata al server, seleziona il parametro di ricerca (*search*) e lo utilizza direttamente. Se il campo di ricerca è compilato, il valore inserito dall'utente viene passato al DOM tramite la proprietà ***innerHTML***, permettendo l'esecuzione di potenziali script malevoli direttamente nel browser.

```
<img src=x onerror="alert('test')">
```

Ad esempio, inserendo un input come il seguente, si tenta di caricare un'immagine inesistente, che genera un errore. Al verificarsi dell'errore, l'attaccante sfrutta la vulnerabilità eseguendo uno script *JavaScript* dannoso.



**Figura 4.6.** *Payload con immagine inesistente*

Questa tecnica dimostra come il DOM possa essere sfruttato per eseguire codice non autorizzato senza mai interagire direttamente con il server, rendendo più difficile la rilevazione dell'attacco.

## 4.4 Contromisure

La problematica principale dietro agli attacchi XSS è la mancanza di controlli sull'input dell'utente e sull'output dell'applicazione. Una delle strategie più efficaci per mitigare il rischio di questi attacchi è sanificare e validare correttamente l'input e l'output.

### 4.4.1 Sanitizzazione e validazione dell'input

Uno strumento molto utile per sanificare l'input dell'utente è l'utilizzo delle *espressioni regolari*. Attraverso questi vincoli sui dati inseriti dall'utente, è possibile filtrare potenziali attacchi XSS. Le espressioni regolari vengono applicate direttamente all'input e, se i dati non rispettano i vincoli definiti dal programmatore, l'input viene scartato. Ad esempio, per il campo *username* possiamo usare una regola che permette solo caratteri alfanumerici e trattini bassi:

```
1 $username = $_POST['username'];
2 if (!preg_match('/^[a-zA-Z0-9_]+$/', $username)) {
3     echo "Input non valido.";
4 } else {
5     echo "Input valido.";
6 }
```

**Listing 4.10.** "Caratteri alfanumerici"

Con questa espressione regolare andiamo a permettere l'inserimento di solo caratteri alfanumerici e trattini bassi per il campo *username*. Abbiamo anche la possibilità di inserire un'espressione regolare che controlli anche per la presenza di tag HTML ('<' o '>') in modo da evitare l'esecuzione interna di script malevoli:

```
1 $comment = $_POST['comment'];
2 $sanitized_comment = preg_replace('<[^>]+>', '',
3 $comment);
```

**Listing 4.11.** "Controllo su caratteri speciali"

Un altro metodo di difesa è l'*escaping dei caratteri speciali* come &, <, >, ", e '. Questo impedisce che questi caratteri vengano interpretati come codice eseguibile all'interno del browser.

```
1 $comment = $_POST['comment'];  
2 $comment = htmlspecialchars($comment, ENT_QUOTES, 'UTF-8');
```

Listing 4.12. "Escaping per caratteri speciali"

#### 4.4.2 Content Security Policy (CSP)

Il *Content Security Policy*, o CSP, è un meccanismo di sicurezza che permette ai proprietari di applicazioni web di controllare quali risorse possono essere caricate ed eseguite nel browser. Questo rappresenta una forma di controllo che limita la presenza di script malevoli. La particolarità di questo modulo di sicurezza aggiuntivo è la possibilità di creare delle proprie *policy* per definire quale risorse accettare all'interno della propria applicazione. Per abilitare CSP è necessario aggiungere un tag HTML all'interno dell'applicazione, ovvero configurare il proprio web server che ritorni il *Content-Security-Policy HTTP header*.

L'utilizzo di CSP permette anche la creazione delle proprie *policy*, in modo tale da supervisionare l'origine del contenuto all'interno propria applicazione. Ad esempio:

```
1 Content-Security-Policy: default-src 'self'
```

In questo caso questa *policy* permette la visualizzazione di contenuti che provengono esclusivamente dall'origine dell'applicazione stessa.

```
1 Content-Security-Policy: default-src 'self'  
2 example.com *.example.com
```

Questa policy permette la visualizzazione di contenuti che provengono esclusivamente di contenuti provenienti esclusivamente da domini e sottodomini fidati.

```
1 Content-Security-Policy: default-src 'self';  
2 img-src *;  
3 media-src example.org example.net;  
4 script-src userscripts.example.com
```

Infine questa policy permette la presenza di immagini provenienti da qualsiasi origine, però limita l'utilizzo di elementi *audio* e *video* esclusivamente da *provider* fidati.

#### 4.4.3 Cookie con attributi di sicurezza

I *cookie* rappresentano degli elementi critici per la *user experience* dell'utente all'interno di un'applicazione web. Un'utilità che hanno è il salvataggio della sessione di un determinato utente, il quale, nel caso esca dall'applicazione per poi tornare, non dovrà effettuare la procedura di *login* in quanto il cookie si sarà salvato un *id di sessione*, il quale verrà ripristinato al rientro dell'utente.

Per impedire ad eventuali aggressori di utilizzare queste caratteristiche dei cookie per scopi illeciti, l'amministratore di sistema può inserire degli attributi di sicurezza ai cookie per mitigare il rischio di exploit di questi.

```
1 setcookie('session_id', $session_id,  
2 ['httponly' => true, 'secure' => true,  
3 'samesite' => 'Strict']);
```

In questo caso, l'attributo *httponly* è utilizzato per impedirne l'accesso tramite JavaScript, proteggendo i cookie da eventuali attacchi XSS. L'attributo *secure* è utilizzato per garantire che vengano trasmessi solo su connessioni HTTPS. Infine l'attributo *samesite* è utilizzato per controllare come e quando i cookie vengono inviati dal browser in relazione al contesto di origine (*origin context*) del sito web.

## 5 File Inclusion

### 5.1 Panoramica

**File Inclusion** è una tecnica utilizzata per includere file su un server attraverso una richiesta HTTP specifica. Tuttavia, questa richiesta può portare ad accessi non autorizzati a file riservati, esporre le applicazioni web a ulteriori rischi e compromettere completamente il sistema. Questi attacchi sono generalmente effettuati per includere ed eseguire un file, che può essere locale o remoto (server controllato dall'attaccante). Questo tipo di attacco sfrutta la funzionalità di un'applicazione web che consente ad un utente di caricare o includere *file*.

Esistono due principali tipologie di attacchi *file inclusion*: il **Remote File Inclusion (RFI)** e il **Local File Inclusion (LFI)**. Un aggressore che effettua un attacco RFI andrà ad utilizzare un server controllato per includere un *file malevolo* all'interno dell'applicazione legittima. Al contrario, un aggressore che effettua un attacco LFI, include file locali (già presenti sul file della vittima), spesso riservati o critici per il sistema.

Questi attacchi sono solitamente utilizzati per prelevare dati sensibili, preparare una base per ulteriori attacchi (ad esempio, un attacco di **path traversal**) o ottenere un accesso non autorizzato al sistema. Si verificano principalmente in applicazioni web prive di adeguati controlli sull'input dell'utente, con scarsi controlli di sicurezza o in server web configurati in modo inadeguato.

*Remote File Inclusion* (RFI) è una vulnerabilità che si verifica quando un'applicazione include file esterni attraverso un campo di input che non correttamente "*sanificato*". Ciò consente all'aggressore di inserire codice malevolo o direttamente dei *link* che puntano ad altri server esterni sotto il suo controllo, sui quali sono presenti dei codici specifici per l'acquisizione non autorizzata di dati.

Per le applicazioni PHP questo tipo di attacco è particolarmente critico, poichè sfrutta la funzione *include* di PHP, che permette di includere codice sorgente da altri file, inclusi quelli remoti. Se esiste un campo di testo per l'utente che consente di specificare il percorso del file da includere, e tale input non è adeguatamente controllato, l'aggressore può modificare l'input per includere file malevoli da server remoti.

Diversamente dagli attacchi RFI, il *Local File Inclusion* (LFI) utilizza file già presenti all'interno del server della vittima. In particolare questo tipo di attacco si verifica spesso a causa di errori di sviluppo da parte del programmatore dell'applicazione web. In molte situazioni, il comando *include* viene utilizzato per accedere a file locali senza adeguati controlli, esponendo l'applicazione web a vulnerabilità.

Un altro vettore di attacco sono le applicazioni che sono sviluppate per visualizzare delle immagini tramite l'inserimento di uno specifico *URL* fornito dall'utente. In questi casi, un aggressore può sfruttare un attacco LFI per accedere direttamente a file locali del server, ovvero persino un RFI per includere file remoti.

## 5.2 Implementazione

Per simulare queste tipologie di attacco, è stato sviluppato un form iniziale con metodo *GET*, in cui l'utente ha la possibilità di inserire il nome di un file presente sul server, al fine di visualizzarne il contenuto.

```
35     echo "<form method='GET'>";
36     echo "<input type='text' name='file'
37     placeholder='Search for data to show...'>";
38     echo "<br>";
39     echo "<br>";
40     echo "<input type='submit' value='search'>";
41     echo "</form>";
```

**Listing 5.1.** "../lfi/app.php"

Dopo l'inserimento del nome del file da visualizzare, viene utilizzato il comando PHP *include* per aprire e mostrare il contenuto del file.

```
42     if (isset($_GET['file'])) {
43         $data = $_GET['file'];
44         if(!include($data)){
45             echo "<p>File not found!</p>";
46         }
47     }
```

**Listing 5.2.** "../lfi/app.php"

**Figura 5.1.** Form iniziale per inserire il nome del file da visualizzare



## 5.3 Risultati sperimentali

Per dimostrare la vulnerabilità del sistema, analizziamo l'aspetto più critico: l'utilizzo del comando PHP *include*. Poiché non vengono effettuati controlli sull'input, un aggressore è in grado di inserire nel campo di testo il nome di file riservati, visualizzandone il contenuto senza disporre delle corrette autorizzazioni.

### 5.3.1 RFI

La peculiarità degli attacchi RFI risiede nel fatto che l'aggressore può accedere a file riservati utilizzando un server remoto sotto il suo controllo, contenente un programma con codice malevolo in grado di cercare i file sensibili all'interno del server della vittima. Per dimostrazione, in questo caso è stato utilizzato lo stesso server locale per simulare l'accesso da remoto. In un contesto reale, l'aggressore utilizzerebbe un *URL* che punta al suo server con codice malevolo.

```
http://localhost/lfi/attack.php
```

Inserendo una stringa di questo tipo all'interno del campo del form, grazie al comando *include*, l'applicazione legittima andrà ad eseguire il file *attack.php* presente su un eventuale server remoto (in questo caso, su *localhost*).

```
3  $file = '../password.txt';
4  if (file_exists($file)) {
5      echo "<p>Contenuto del file:</p>";
6      echo nl2br(file_get_contents($file));
7  } else {
8      echo "<p>File non trovato.</p>";
9  }
```

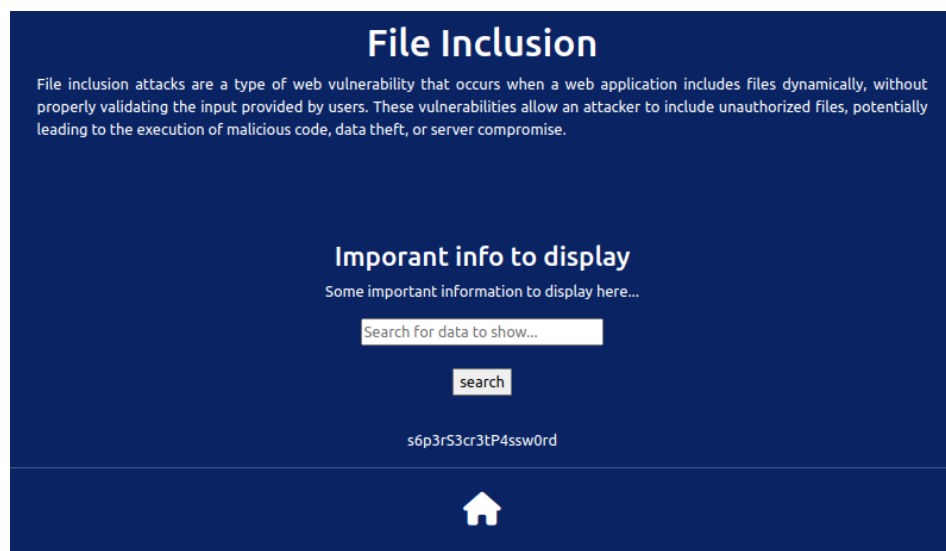
**Listing 5.3.** "../lfi/attack.php"

Nel campo *\$file*, l'aggressore può specificare quale file riservato desidera accedere, ad esempio *password.txt*.

### 5.3.2 LFI

Nel caso di un attacco LFI, l'aggressore può accedere direttamente ai file locali presenti sul server. Per simulare un accesso non autorizzato ad un file riservato, è stato creato un file chiamato *password.txt* situato nella cartella *root* dell'applicazione. Poiché il form iniziale accetta il nome di un file da visualizzare ed esegue direttamente il comando *include* senza ulteriori controlli, è possibile sfruttare questa vulnerabilità includendo file riservati, ad esempio:

```
../password.txt
```

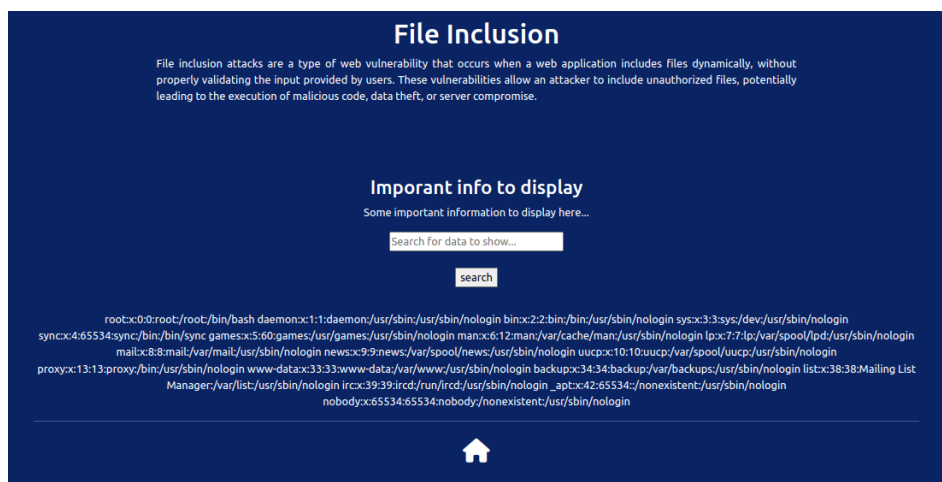


**Figura 5.2.** Inserendo *../password.txt* si accede a file riservati

L'attaccante potrebbe anche decidere di iniziare un attacco di tipo *path traversal*, fornendo un percorso specifico che gli consenta di accedere a directory riservate sul server. Ad esempio:

```
../../../../etc/passwd
```

Inserendo una stringa di questo tipo, l'attaccante potrebbe uscire dalla *directory* dell'applicazione web e accedere a un file riservato che contiene *password* o altre informazioni sensibili.



**Figura 5.3.** Inserendo `../../../../etc/passwd` si accede a file riservati

## 5.4 Contromisure

Per mitigare questo tipo di attacco, il miglior approccio è *sanificare* l'input dell'utente, rimuovendo eventuali stringhe che potrebbero portare ad un'intrusione nel sistema. Tuttavia, non è possibile sanificare completamente l'input dell'utente, motivo per cui è necessario implementare anche dei controlli di *input validation*.

L'*input validation* consiste nel confrontare l'input dell'utente con una serie di caratteri che sono consentiti e non consentiti. Se l'input contiene caratteri non ammessi, viene ignorato. E' necessario eseguire una verifica di validità anche sui file richiesti dall'utente. Un modo efficace per prevenire accessi indesiderati a file riservati è collocarli nelle *directory* appropriate, evitando posizioni dove un aggressore possa inserire codice malevolo e di conseguenza causare un *leak* di dati critico.

Inoltre, per garantire la sicurezza delle directory dell'applicazione, è fondamentale limitare i permessi di esecuzione, implementare una *whitelist* per i tipi di file consentiti e limitare le dimensioni massime dei file caricati. Si consiglia inoltre, di ottimizzare le configurazioni del server, ad esempio inviando automaticamente intestazioni di download invece di eseguire direttamente file all'interno di una directory specifica. Per prevenire attacchi di *directory traversal* invece, è opportuno limitare l'inclusione di file solo da directory autorizzate. Infine, è essenziale eseguire test mirati per verificare se il codice è vulnerabile a exploit di inclusione di file.

### 5.4.1 Validazione dell'input

Come già menzionato, è fondamentale inserire controlli di validazione e sanificazione dell'input dell'utente. Nel nostro caso, è possibile implementare questi controlli direttamente sulla casella di input dell'utente, bloccando caratteri non consentiti.

```
44 $file = preg_replace('/[^\a-zA-Z0-9\_]/', '',  
45 $_GET['file']);
```

**Listing 5.4.** "../lfi/app.php"

Attraverso l'utilizzo di *espressioni regolari*, si possono consentire solo input validi (ad esempio, nomi di file attesi) e rimuovere caratteri potenzialmente pericolosi come `..`, `/` e `%00`.

```
40 $whitelist = ['file1.php', 'file2.php'];  
41 if (in_array($_GET['file'], $whitelist)) {  
42     include($_GET['file']);  
43 } else {  
44     echo "File non autorizzato.";
```

**Listing 5.5.** "../lfi/app.php"

Un altro approccio alla validazione dell'input dell'utente consiste nell'implementare una *whitelist* di file a cui utente è può accedere. Questo permette di limitare gli accessi non autorizzati a file riservati.

### 5.4.2 Corretta implementazione di file di configurazione

E' importante implementare una serie di controlli direttamente all'interno del file di configurazione del server web, per ridurre la possibilità di attacchi di tipo *file inclusion*. Una possibile soluzione è disattivare l'opzione `allow_url_include` nel file `php.ini`, impedendo così di l'inclusione di file remoti. Questo previene l'esecuzione di file remoti tramite la funzione `include()` o `require()`.

```
allow_url_include = Off
```

Un'altra misura per limitare questi attacchi è configurare il parametro *open\_basedir* in PHP, che limita le *directory* da cui PHP può accedere ai file di sistema. Questa opzione impedisce l'accesso a directory riservate.

```
open_basedir = /var/www/html:/var/www/include/
```

Infine, è possibile disabilitare funzioni potenzialmente pericolose utilizzando il parametro *disable\_functions* nel file di configurazione..

```
disable_functions = "system, exec, shell_exec, passthru, eval, include,  
include_once, require, require_once"
```

## 6 Command Injection

### 6.1 Panoramica

Gli attacchi di *Command Injection* si verificano quando una stringa inserita da un attaccante, solitamente attraverso un campo di input, viene interpretata non come semplice testo, ma come un comando capace di influenzare il comportamento dell'applicazione web.

Queste vulnerabilità derivano generalmente da una gestione inadeguata dei dati forniti dall'utente, combinata con l'uso di *comandi shell* impiegati per scopi legittimi all'interno dell'applicazione. Inoltre, tali vulnerabilità si manifestano anche a causa della mancanza di controlli adeguati sull'input, che potrebbe contenere caratteri particolari come punteggiatura, punti di domanda o *caratteri di escape*, rendendo il processo di *parsing* dei comandi più complesso e pericoloso.

E' opportuno definire la differenza tra *code injection* e *command injection*.

Un attacco di **code injection** si verifica quando l'attaccante riesce a inserire codice malevolo che viene successivamente eseguito dall'applicazione. Tali attacchi sono spesso facilitati dall'assenza di un'adeguata validazione dei dati in ingresso. Tuttavia, una limitazione significativa di questi attacchi è che l'esecuzione del codice malevolo è confinata ai limiti dell'applicazione o del sistema attaccato. Ad esempio, in un attacco di *code injection* su un sistema che utilizza Python, l'attaccante sarebbe vincolato dai permessi assegnati a Python sul sistema host.

La particolarità degli attacchi di *command injection* è che, in questo caso, l'attaccante non si limita a eseguire codice interno all'applicazione, ma espande il suo funzionamento eseguendo comandi di sistema arbitrari. L'obiettivo dell'attaccante è eseguire *comandi di sistema* direttamente sul server dell'applicazione, senza dover necessariamente inserire codice specifico.

Questi attacchi sono resi possibili dall'uso di form di input non adeguatamente validati, nonché da cookie o header HTTP non sicuri. Un attacco di command injection può compromettere non solo l'applicazione stessa, ma anche i dati in essa contenuti, i server collegati al sistema e altri dispositivi connessi alla rete.

## 6.2 Implementazione

Per simulare questo tipo di attacco, è stato sviluppato un form con metodo *POST*, in cui l'utente può inserire un indirizzo IP per eseguire un **ping**, ovvero un comando che invia una serie di pacchetti **ICMP** per verificare la connettività di un determinato server web.

```
29     echo "<form method='POST'>";
30     echo "<input type='text' name='ip'
31         placeholder='Enter ip to ping...'>";
32     echo "<input type='submit'
33         value='Execute' style='margin-bottom:4vh'>";
34     echo "</form>";
35
36     if(isset($_POST['ip'])) {
37         include('ping.php');
38     }
```

Listing 6.1. "../cmi/app.php"

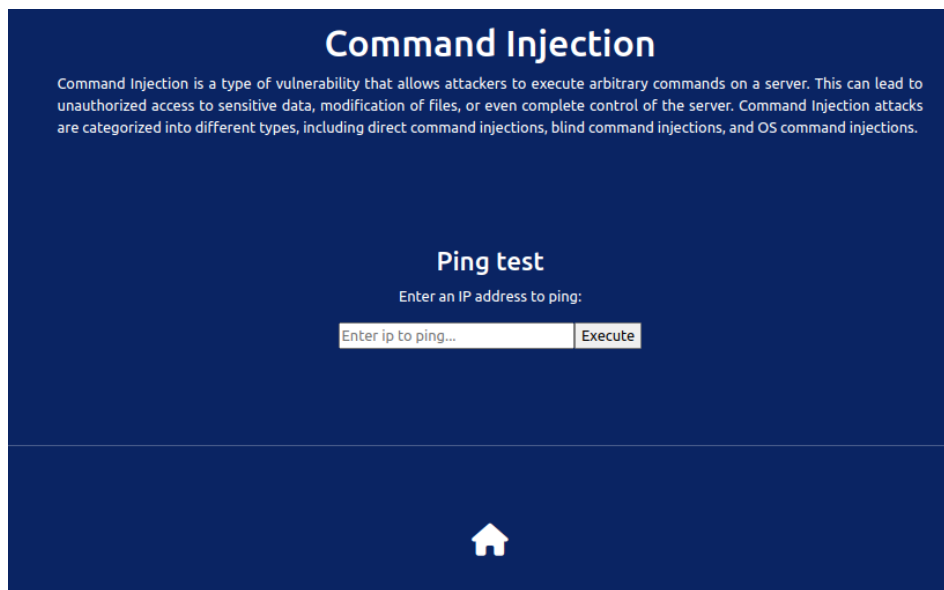
La scelta di utilizzare il metodo POST è motivata dal fatto che l'esecuzione del comando è gestita tramite un file PHP specifico, chiamato **ping.php**. Dopo l'inserimento dell'indirizzo IP da *pingare*, per semplicità viene utilizzato il comando PHP include per visualizzare l'esecuzione della pagina ping.php.

```
1     <?php
2         if (isset($_POST['ip'])) {
3             $ip = $_POST['ip'];
4
5             $output = shell_exec("ping -c 4 " . $ip);
6
7             echo "<pre>$output</pre>";
8         } else {
9             echo "Nessun IP fornito.";
10        }
11    ?>
```

Listing 6.2. "../cmi/app.php"

In questo modo, si verifica se l'indirizzo IP è stato inserito correttamente e, in caso positivo, l'input dell'utente viene passato direttamente al comando di sistema *ping*. Tuttavia, questa implementazione presenta una vulnerabilità critica, poiché l'input dell'utente non è adeguatamente validato.





**Command Injection**

Command Injection is a type of vulnerability that allows attackers to execute arbitrary commands on a server. This can lead to unauthorized access to sensitive data, modification of files, or even complete control of the server. Command Injection attacks are categorized into different types, including direct command injections, blind command injections, and OS command injections.

**Ping test**

Enter an IP address to ping:

Enter ip to ping...

Home icon

**Figura 6.1.** *Form iniziale dove inserire l'IP*



**Command Injection**

Command Injection is a type of vulnerability that allows attackers to execute arbitrary commands on a server. This can lead to unauthorized access to sensitive data, modification of files, or even complete control of the server. Command Injection attacks are categorized into different types, including direct command injections, blind command injections, and OS command injections.

**Ping test**

Enter an IP address to ping:

Enter ip to ping...

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data:
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.056 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.070 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.085 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.085 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3085ms
rtt min/avg/max/mdev = 0.056/0.074/0.085/0.012 ms
```

**Figura 6.2.** *Inserimento indirizzo e output del comando*

Per semplicità, è stato utilizzato l'indirizzo 127.0.0.1, ovvero *localhost* dove è *hostata* l'applicazione.

## 6.3 Risultati sperimentali

La problematica principale, come per molti attacchi nell'ambito della sicurezza informatica, risiede nella mancanza di adeguati controlli sull'input dell'utente. Un attaccante, manipolando l'input, può infatti inserire codice malevolo direttamente in un comando di sistema. Esistono diverse varianti di questo attacco, tra cui le principali sono:

### 6.3.1 Direct Command Injection

In questo tipo di attacco, l'input fornito dall'attaccante viene passato direttamente alla **shell** del sistema. Se un'applicazione accetta input e lo passa a comandi di sistema come *ping* o *ls*, un attaccante potrebbe sfruttare caratteri speciali come `&&` o `;` per concatenare comandi arbitrari.

```
; ls -a
```

Inserendo una stringa di questo tipo direttamente nel campo di input dell'utente, l'attaccante sfrutta il punto e virgola (`;`) per concatenare un nuovo comando. In questo caso, il comando *ping* viene ignorato e si esegue *ls -a*, che elenca tutte le directory presenti nella cartella dell'applicazione.



Figura 6.3. Inserimento del comando per elencare le directory

Un ulteriore esempio di input malevolo è il caso in cui l'attaccante decida di cercare dei file riservati all'interno di *directory* al di fuori di quella dell'applicazione web, difatti inserendo:

```
; cat ../../../../etc/passwd;
```

l'attaccante può ispezionare altre directory fino a raggiungere file sensibili, come il file **passwd** che contiene informazioni sugli account di sistema.

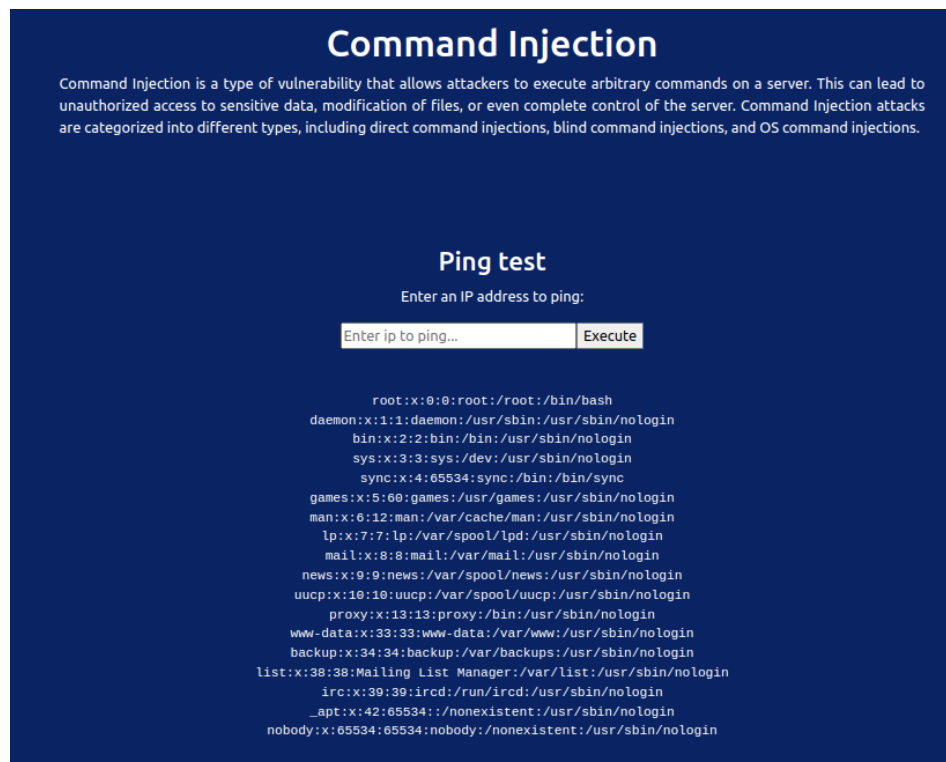


Figura 6.4. Inserimento del comando per accedere a file riservati

### 6.3.2 Blind Command Injection

A differenza del *direct command injection*, dove l'output del comando di sistema viene visualizzato immediatamente dall'utente, in un attacco di *blind command injection* l'attaccante non riceve un output esplicito. In questi casi, l'attaccante può usare comandi come **sleep** per verificare se l'attacco ha avuto successo.

Questo tipo di attacco è utile quando si vuole testare la vulnerabilità del sistema senza causare subito danni visibili, manipolare file di configurazione di sistema, o eseguire comandi di rete.

```
; echo 'modifica' > /usr/local/etc/php.ini; echo 'Attacco riuscito' > /tmp/command_injection.txt; sleep 10;
```

In questo scenario, l'attaccante simula la modifica di un file di sistema, inserendo la stringa "modifica" nel file **php.ini**. Non avendo un output immediato dell'operazione, l'attaccante concatena altri due comandi per ottenere conferma del successo dell'attacco: la creazione di un file temporaneo e l'attesa di 10 secondi, indicativi che l'attacco è stato eseguito correttamente.

## 6.4 Contromisure

Quando le applicazioni gestiscono l'input degli utenti, è fondamentale assumere sempre che questo possa essere malevolo. Sia che si tratti di un attacco intenzionale o di un errore umano, l'input potrebbe influenzare il *parsing* dei comandi di sistema, esponendo l'applicazione a potenziali rischi. Per questo motivo, esistono diverse strategie che possono mitigare il rischio di attacchi di questo tipo:

### 6.4.1 Controlli di validità e sanificazione dell'input

Poiché l'input dell'utente rappresenta una delle principali *superfici di attacco*, è cruciale implementare adeguati controlli di validità e meccanismi di sanificazione. L'obiettivo è consentire agli utenti di inserire input legittimi, ma al contempo bloccare eventuali tentativi di sfruttamento.

```
1  <?php
2      if (isset($_POST['ip'])) {
3          $ip = $_POST['ip'];
4
5          if (preg_match('/^[a-zA-Z0-9.-]+$/ ', $ip)) {
6              system("ping -c 4 " . escapeshellarg($ip));
7          } else {
8              echo "Dominio non valido.";
9          }
10     ?>
```

Listing 6.3. "../cmi/app.php"

Utilizzando la funzione PHP *escapeshellarg()*, è possibile gestire in modo sicuro l'input dell'utente, ignorando automaticamente i caratteri potenzialmente pericolosi, come quelli utilizzati per concatenare comandi.

### 6.4.2 Disabilitare l'accesso alla shell

È buona pratica limitare l'uso della shell e accedervi solo quando strettamente necessario per l'esecuzione di comandi di sistema.

```
1 $filename = $_GET['filename'];  
2 system("cat " . $filename);
```

Listing 6.4. "example.php"

Per esempio, in un contesto in cui è richiesto leggere il contenuto di un file, anziché usare un comando come **cat**, è consigliabile ricorrere a funzioni PHP native, come *file\_get\_contents()*, per evitare l'interazione diretta con la shell, riducendo così il rischio di vulnerabilità.

```
1 $filename = $_GET['filename'];  
2  
3 if (preg_match('/^[a-zA-Z0-9._-]+$/', $filename)) {  
4     $file_content =  
5     file_get_contents("/path/to/dir/" . $filename);  
6     echo "<pre>$file_content</pre>";  
7 } else {  
8     echo "File non valido.";  
9 }
```

Listing 6.5. "example.php"

### 6.4.3 Utilizzare un Web Application Firewall (WAF)

Un Web Application Firewall (WAF) è uno strumento efficace per filtrare il traffico di rete in ingresso e prevenire l'esecuzione di richieste contenenti codice malevolo o comandi di sistema non autorizzati. Un esempio comune è **ModSecurity**, un WAF che può essere configurato per mitigare attacchi di Command Injection. Ecco un esempio di regola che può essere applicata per bloccare caratteri pericolosi all'interno dei parametri URL:

```
SecRule REQUEST_URI "@rx cmd=" "id:'1000001', phase:1, t:none,  
deny, log,msg:'Possible Command Injection'"
```

## 7 Insecure Direct Object Reference

### 7.1 Panoramica

*Insecure direct object reference* (IDOR) è una vulnerabilità che si manifesta quando un attaccante ha la possibilità di accedere o modificare determinati oggetti attraverso la manipolazione degli identificatori utilizzati negli *URL* o nei parametri delle applicazioni web. Tali attacchi si verificano principalmente in assenza di adeguati controlli sui permessi di accesso degli utenti a specifiche sezioni dell'applicazione.

Uno degli elementi fondamentali della sicurezza informatica è il concetto di controllo degli accessi (**access control**). Questo meccanismo regola l'accesso ai dati e alle risorse di un'applicazione, decidendo se una richiesta effettuata da un utente debba essere accettata o rifiutata in base alle azioni che sta cercando di eseguire o alle risorse a cui desidera accedere. I **ruoli** assegnati agli utenti determinano i loro privilegi all'interno del sistema: ad esempio, gli amministratori avranno più permessi rispetto agli utenti standard. Il controllo degli accessi è strettamente legato al modello di autorizzazione implementato nel sistema e può essere classificato in tre categorie principali:

- **Vertical access control:** si occupa di regolare le restrizioni di accesso alle funzionalità dell'applicazione in base ai ruoli degli utenti. Ad esempio, un moderatore avrà più privilegi rispetto a un utente comune, mentre un amministratore disporrà del livello massimo di accesso.
- **Horizontal access control:** gestisce l'accesso alle risorse tra utenti che hanno lo stesso livello di ruolo. Anche se gli utenti condividono lo stesso ruolo, non dovrebbero poter accedere ai dati di altri utenti.

- **Context-dependent access control:** regola le restrizioni in base a circostanze o azioni specifiche compiute dall'utente. Ad esempio, un utente che ha acquistato un prodotto su un sito di *e-commerce* non dovrebbe poter modificare l'indirizzo di spedizione dopo che l'ordine è stato inviato.

Gli amministratori di sistema possono anche implementare varie *policy* per gestire il controllo degli accessi all'interno delle loro applicazioni, che si suddividono in quattro categorie principali:

- **Discretionary access control (DAC):** l'accesso è controllato in base all'identità dell'utente e alle autorizzazioni concesse. È detto "*discrezionale*" perché un'entità che possiede permessi su una risorsa può decidere di concedere l'accesso a un'altra entità.
- **Mandatory access control (MAC):** l'accesso è regolato confrontando i parametri di sicurezza delle risorse (ad esempio, il livello di sensibilità) con le autorizzazioni dell'entità richiedente. È definito "*obbligatorio*" perché le decisioni di accesso non sono a discrezione dell'utente.
- **Role-based access control (RBAC):** l'accesso è determinato dai ruoli che un'entità ricopre nel sistema e dal modello di autorizzazione adottato. Gli utenti accedono alle risorse in base al loro ruolo, che stabilisce i loro privilegi.
- **Attribute-based access control (ABAC):** l'accesso è regolato in base agli attributi dell'entità, della risorsa e alle variabili di contesto. Gli attributi possono includere informazioni sull'utente (come la posizione o le autorizzazioni) e proprietà della risorsa.

Un attacco di tipo *insecure direct object reference (IDOR)* si verifica quando un attaccante riesce a manipolare gli input dell'applicazione per accedere a un oggetto in modo illecito. Gli attaccanti possono aggirare i meccanismi di autorizzazione, ottenendo così accesso non autorizzato alle risorse del sistema attraverso questa vulnerabilità.

## 7.2 Implementazione

Per simulare questo tipo di attacco, è stata sviluppata una pagina PHP che carica una serie di dati relativi all'utente attualmente autenticato, simulando una tipica pagina di profilo utente in un contesto reale.

Quando l'utente accede alla pagina, il suo nome utente viene passato come parametro, permettendo la visualizzazione del relativo profilo. La vulnerabilità risiede nell'assenza di un adeguato meccanismo di controllo degli accessi: in questo modo, un attaccante può manipolare il parametro dell'utente corrente, accedendo a dati non autorizzati.

```
146 <?php
147     $user = $_SESSION['username'];
148     echo "<a class='btn btn-primary'
149         href='../idor/app.php?user=" . urlencode($user) . "'>
150         Start</a>";
151     ?>
```

**Listing 7.1.** "index.php"

```
26     $dsn = 'pgsql:host=postgres;port=5432;dbname=database';
27     $username = 'user';
28     $password = '.UYr930Qr';
29
30     $pdo = new PDO($dsn, $username, $password);
31
32     if (isset($_GET['user'])) {
33         $user = $_GET['user'];
34         $sql = "SELECT * FROM users WHERE username = '$user'";
35         $stmt = $pdo->query($sql);
36         ...
```

**Listing 7.2.** "../idor/app.php"

In primo luogo, viene stabilita la connessione al database, poiché è necessario accedere ai dati dell'utente memorizzati per mostrarli nella pagina. Tuttavia, si nota l'assenza di un meccanismo di controllo degli accessi nella fase di esecuzione della query SQL per la selezione dei dati dell'utente. L'unica condizione presente è che i dati recuperati abbiano l'attributo *"username"* corrispondente al valore della variabile *\$user*, che viene definita in base al parametro passato nell'URL.



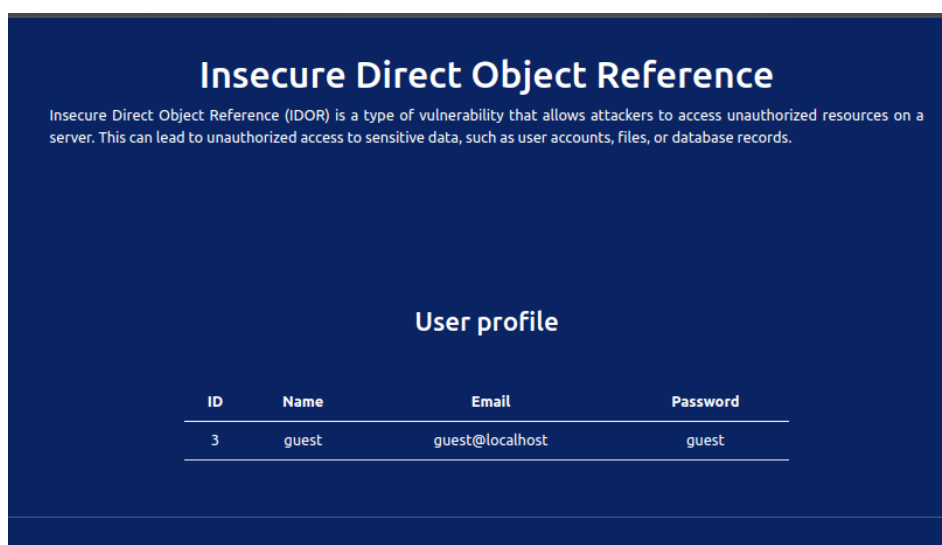


Figura 7.1. Sezione profilo dell'utente

## 7.3 Risultati sperimentali

Per dimostrare la vulnerabilità del sistema, analizziamo la criticità principale legata alla modalità di accesso ai dati degli utenti. In assenza di un adeguato meccanismo di controllo degli accessi, un attaccante può facilmente navigare tra i profili di tutti gli altri utenti. Questo è possibile grazie alla manipolazione del parametro *user* passato alla pagina.

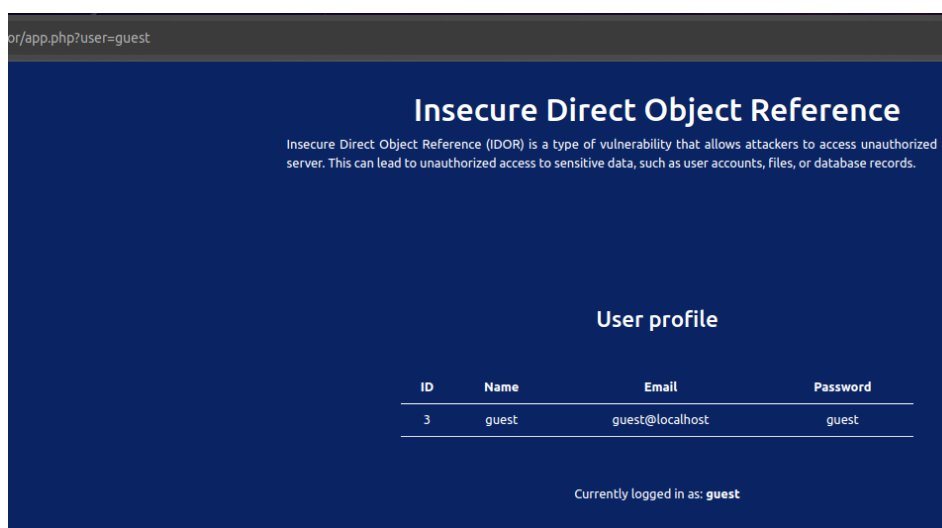
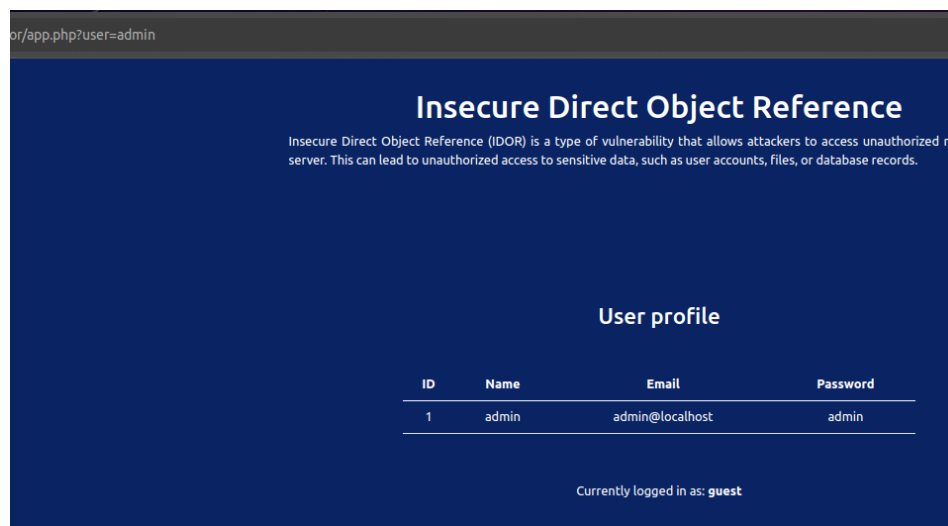


Figura 7.2. Sezione profilo dell'utente correntemente loggato

Per maggiore chiarezza, è stata aggiunta una sezione che indica quale profilo ha effettuato l'accesso all'applicazione.



**Figura 7.3.** Sezione profilo dell'utente *admin*

Modificando direttamente il parametro *user* nell'URL, un attaccante, pur essendo *loggato* come utente *guest*, può accedere e visualizzare il profilo di *admin*, senza disporre delle autorizzazioni o dei ruoli appropriati.

## 7.4 Contromisure

Come menzionato in precedenza, questo tipo di attacco si verifica spesso a causa dell'assenza di un adeguato meccanismo di controllo degli accessi, il che introduce una vulnerabilità critica all'interno dell'applicazione. Tuttavia, l'implementazione di un controllo degli accessi non è l'unica soluzione possibile.

### 7.4.1 Sanitizzazione dell'input

Nel nostro esempio di applicazione vulnerabile, il parametro *user* viene inserito direttamente all'interno dei campi della query SQL inviata al database. Questo rappresenta una potenziale vulnerabilità, poiché apre la porta a un possibile attacco di tipo *SQL injection*. Per mitigare questo rischio, è essenziale implementare query SQL parametrizzate, che evitano l'inserimento di input malevoli all'interno delle query.

### 7.4.2 Controllo degli accessi

Prima di mostrare i dati di un utente, è fondamentale verificare che l'utente che richiede l'informazione disponga effettivamente dei permessi necessari per accedere a tali dati. Una soluzione di base consiste nell'utilizzare le **sessioni utente**, verificando che l'utente *loggato* possa accedere esclusivamente alle proprie informazioni. Anche se rudimentale, questo approccio rappresenta un primo passo verso un meccanismo di *controllo degli accessi* efficace.

```
26     if (isset($_SESSION['username'])) {
27         $loggedInUser = $_SESSION['username'];
28         if ($loggedInUser === $_GET['user']) {
29             ...
30         } else {
31             echo "Accesso negato!";
32         }
33     } else {
34         echo "Non sei autenticato.";
35     }
```

Listing 7.3. "../idor/app.php"

### 7.4.3 Limitazione dell'accesso per ID sensibili

Per ridurre il rischio di attacchi IDOR, una possibile strategia consiste nell'utilizzare **identificatori non sequenziali** per riferire a oggetti sensibili. Nel nostro caso, i dati dell'utente sono caricati esclusivamente in base al parametro user. Tuttavia, l'adozione di identificatori randomici o complessi per i profili utente può contribuire a mitigare il rischio di un attacco IDOR.

### 7.4.4 Access control list

Un'ulteriore misura efficace che un amministratore può implementare all'interno di un'applicazione è l'uso di una **Access Control List (ACL)**. Un'ACL è una lista di regole che specifica quali utenti o ruoli hanno il permesso di accedere a determinate risorse o di eseguire specifiche azioni all'interno del sistema. Una ACL può essere gestita direttamente all'interno di un database.

```
-- Tabella utenti con ruolo
CREATE TABLE users(
    id SERIAL primary key,
    username varchar(255) NOT NULL,
    password varchar(255) NOT NULL,
    email varchar(255) NOT NULL,
    role_id int REFERENCES roles(id)
);

-- Tabella dei ruoli
CREATE TABLE roles (
    id SERIAL primary key,
    role_name varchar(255) NOT NULL
);

-- Tabella delle autorizzazioni per ogni ruolo
CREATE TABLE permissions (
    id SERIAL primary key,
    role_id int REFERENCES roles(id),
    resource varchar(255) NOT NULL,
    can_access boolean NOT NULL DEFAULT false
);
```

Listing 7.4. "init.sql"

In particolare, con l'introduzione di nuove relazioni, emerge l'importanza delle **chiavi esterne**. Nel nostro esempio, nella tabella *users* è stato aggiunto l'attributo *role\_id*, che rappresenta una chiave esterna collegata alla chiave primaria della tabella *roles*. Quest'ultima definisce i vari ruoli che possono essere assegnati agli utenti. La tabella *permissions*, infine, stabilisce i permessi associati a ciascun ruolo.

```
-- Esempio di ruoli
INSERT INTO roles (role_name) VALUES ('admin'), ('user'), ('guest');

-- Esempio di permessi: solo gli admin possono accedere alla risorsa
"admin_panel"
INSERT INTO permissions (role_id, resource, can_access) VALUES (1,
    'admin_panel', true);
```

Listing 7.5. "init.sql"

Nel nostro esempio, simuleremo l'accesso a una risorsa riservata, denominata *admin\_panel*, a cui possono accedere solo gli utenti con il ruolo di *admin*.

```
20     function userHasAccess($pdo, $userId, $resource) {
21         $sql = "
22             SELECT p.can_access
23             FROM users u
24             JOIN roles r ON u.role_id = r.id
25             JOIN permissions p ON r.id = p.role_id
26             WHERE u.id = :userId AND p.resource = :resource
27         ";
28         $stmt = $pdo->prepare($sql);
29         $stmt->execute(['userId' => $userId,
30             'resource' => $resource]);
31         return $stmt->fetchColumn() === '1';
32     }
33
34     if (isset($_SESSION['username']) &&
35         userHasAccess($pdo, $_SESSION['username'], 'admin_panel')) {
36         echo "Benvenuto nell'area riservata!";
37     } else {
38         echo "Accesso negato!";
39     }
```

Listing 7.6. "../idor/app.php"

Per verificare se l'utente che tenta di accedere alla risorsa riservata *admin\_panel* possiede le autorizzazioni necessarie, è stata creata una funzione che prende in input la variabile *\$pdo* (necessaria per la connessione al database), lo *username* della sessione e la risorsa a cui si desidera accedere. All'interno della funzione viene eseguita una query SQL che effettua una **JOIN** tra le tabelle *users*, *roles* e *permissions*. Una JOIN è un'operazione SQL che consente di combinare i dati di due o più tabelle in base a una condizione comune, come una colonna condivisa tra le relazioni (ad esempio una chiave esterna).

Successivamente, viene eseguita la query SQL e si effettua un controllo sul risultato tramite la funzione *fetchColumn()*, che estrae il valore della prima colonna della prima riga. Se il valore è uguale a '1', il quale corrisponde al ruolo di *admin*, la funzione restituisce *true*, indicando che l'utente ha i permessi necessari; altrimenti, restituisce *false*.

## 8 Cross-Site Request Forgery

### 8.1 Panoramica

Gli attacchi di tipo **Cross-Site Request Forgery (CSRF)** si verificano quando un sito web malevolo induce il browser di un utente a eseguire azioni non volute su un sito legittimo. Questo tipo di attacco rappresenta una vulnerabilità spesso sottovalutata, ma con potenziali impatti critici per le applicazioni web. Esiste, inoltre, l'erronea convinzione che adottando contromisure per l'attacco **Cross-Site Scripting (XSS)** si sia al sicuro anche dagli attacchi CSRF, ma vedremo che non è così.

Questi attacchi si basano fortemente su tecniche di **social engineering** contro gli utenti, poiché è sufficiente che un utente visiti un sito malevolo per innescare l'attacco. Inoltre, il CSRF sfrutta una caratteristica comune a molti siti web: l'**autenticazione implicita** degli utenti. Una volta che un utente accede al sito e inserisce le proprie credenziali, ogni successiva richiesta non richiede nuovamente l'autenticazione. Questa funzionalità migliora l'esperienza utente, rendendo la navigazione più fluida, poiché il sito web accetta ogni successiva richiesta come legittima e proveniente dall'utente autenticato.

Tuttavia, questa caratteristica può essere sfruttata da un attaccante per inviare una serie di richieste a nome dell'utente autenticato, senza che quest'ultimo ne sia consapevole. In pratica, l'attaccante ha bisogno solo che l'utente visiti un sito malevolo per avviare un attacco di tipo CSRF. Consideriamo un esempio in cui un sito legittimo offra una funzionalità per inviare email a nome di un utente autenticato, utilizzando un modulo HTML con richiesta *GET*.

Supponiamo che il sito legittimo utilizzi una pagina dedicata all'invio delle email con i dati forniti dall'utente. L'attaccante, tramite tecniche di **social engineering**, può indurre l'utente a visitare un sito malevolo.

In questo sito, l'attaccante potrebbe inserire un tag HTML `<img>` con l'attributo `src` impostato sull'indirizzo della pagina del sito legittimo che invia le email, allegando i dati che desidera. Di conseguenza, l'email verrebbe inviata a nome dell'utente, ma con i dati inseriti dall'attaccante, senza che l'utente se ne accorga.

Nel caso in cui il modulo utilizzi il metodo *POST*, l'attaccante può comunque sfruttare tecniche di *social engineering* per far sì che l'utente visiti il sito malevolo. In questo caso, sul sito malevolo sarà presente un modulo con campi nascosti che contengono i dati da inviare al sito legittimo. L'utente, cliccando su un pulsante apparentemente innocuo, in realtà invierà i dati al sito legittimo, avviando così l'attacco *CSRF*.

È evidente che questo tipo di attacco rappresenti una seria vulnerabilità per le applicazioni web che utilizzano l'autenticazione implicita. Poiché l'attacco sfrutta l'identità dell'utente legittimo, i suoi effetti possono essere particolarmente gravi se l'utente ha privilegi elevati sul sito legittimo. In generale, ogni volta che viene utilizzata l'autenticazione implicita, esiste il rischio di un attacco *CSRF*.

Teoricamente, il rischio potrebbe essere eliminato richiedendo l'inserimento delle credenziali per ogni richiesta effettuata al sito, ma questa soluzione comprometterebbe drasticamente l'esperienza utente. È quindi necessario cercare soluzioni alternative che bilancino **sicurezza** e **usabilità** dell'applicazione.



## 8.2 Implementazione

Per simulare un attacco di tipo CSRF, è stato creato un form HTML specifico, che consente all'utente autenticato di cambiare la propria password. Il form, che utilizza il metodo *POST*, invia la nuova password al database, aggiornando così le credenziali dell'utente. Inoltre, è stato implementato un pulsante che simula l'accesso a un sito malevolo: cliccando su questo pulsante, l'utente innesca un attacco CSRF.

```
3      $dsn = 'pgsql:host=postgres;port=5432;dbname=database';
4      $username = 'user';
5      $password = '.UYr930Qr';
6
7      $pdo = new PDO($dsn, $username, $password);
```

Listing 8.1. "../csrf/app.php"

Dal momento che la funzionalità di modifica della password comporta la modifica delle credenziali memorizzate nel database, il primo passo consiste nella connessione al database.

```
37      echo "<form method='POST'>";
38      echo "<input type='password' name='password'";
39      echo "placeholder='New password'>";
40      echo "<br>";
41      echo "<br>";
42      echo "<input type='submit' value='Change password'>";
43      echo "</form>";
44
45      if ($_SERVER['REQUEST_METHOD'] === 'POST') {
46          $newPassword = $_POST['password'];
47
48          $sql = "UPDATE users SET password='$newPassword'";
49          $sql .= "WHERE username='$user'";
50          $stmt = $pdo->query($sql);
51      }
```

Listing 8.2. "../csrf/app.php"

Successivamente, viene generato il form con metodo *POST*, che consente all'utente di inserire una nuova password. Il sistema verifica quindi che il metodo di richiesta sia effettivamente *POST* e, in caso positivo, aggiorna la password dell'utente attualmente autenticato nel sistema.

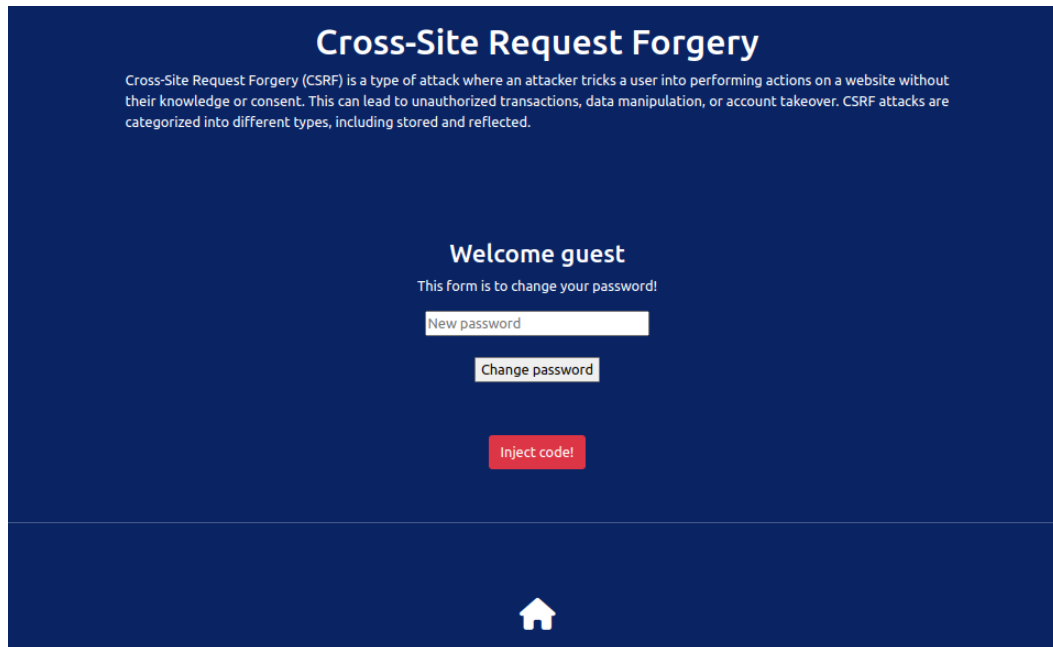
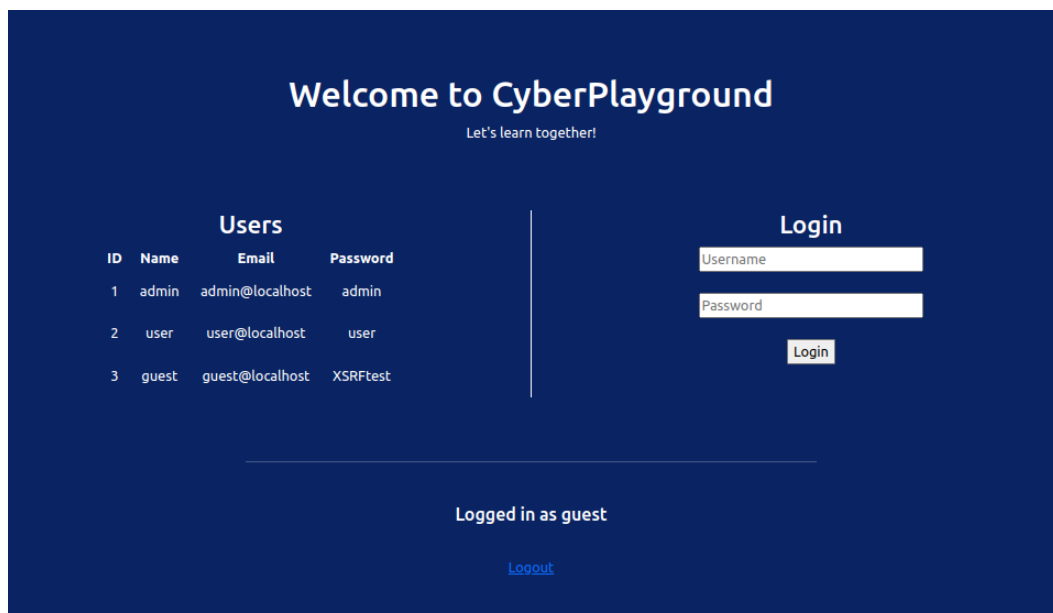


Figura 8.1. Form iniziale di cambio password



ID	Name	Email	Password
1	admin	admin@localhost	admin
2	user	user@localhost	user
3	guest	guest@localhost	XSRFtest

Figura 8.2. Password cambiata con successo

## 8.3 Risultati sperimentali

Per dimostrare la vulnerabilità del sistema, analizziamo la principale criticità nel form di cambio password. Il problema principale risiede nell'utilizzo dell'autenticazione implicita per il cambio della password. Nella nostra applicazione, infatti, non viene richiesta la vecchia password per confermare l'autenticità dell'utente che sta effettuando la richiesta. Questa mancanza di controllo consente a un attaccante di lanciare un attacco di tipo Cross-Site Request Forgery (CSRF), simulato in questo caso tramite un semplice bottone.

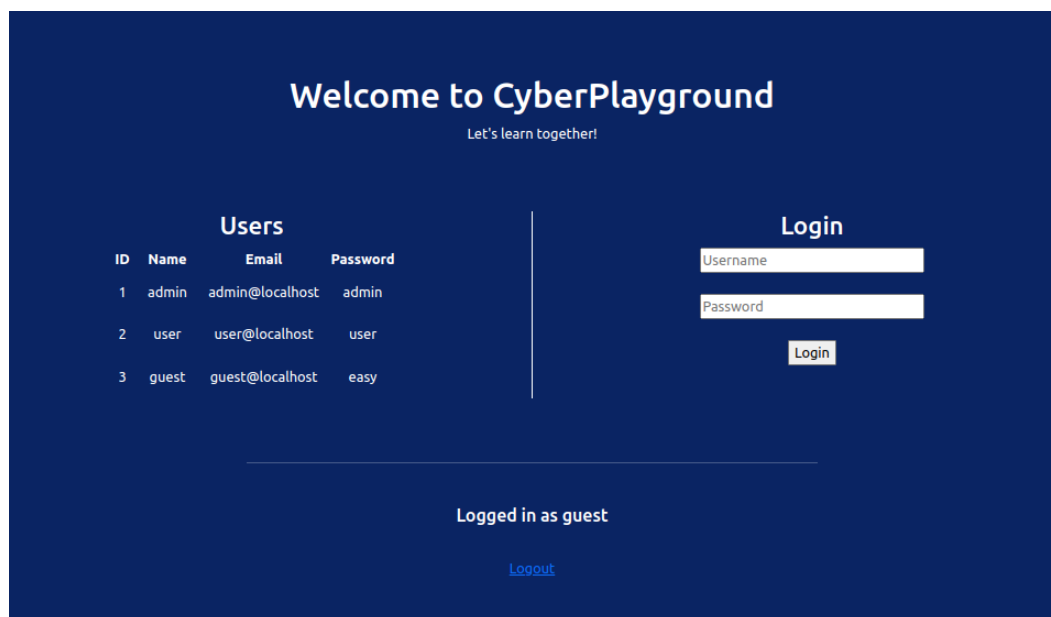
```
58 <button class="btn btn-danger"
59 onclick="window.location.href='attack.html'">
60 Inject code!</button>
```

Listing 8.3. "../csrf/app.php"

A scopo dimostrativo, cliccando sul bottone che innesca l'attacco, l'utente viene reindirizzato alla pagina *attack.html*. All'interno di questa pagina è presente un form invisibile che viene automaticamente inviato alla nostra applicazione non appena la pagina viene caricata, grazie a un piccolo script *JavaScript*. Il form invia la nuova password scelta dall'attaccante, che in questo caso è "easy", senza che l'utente ne sia a conoscenza.

```
2 <body>
3 <form action="http://localhost:4000/csrf/app.php"
4 method="POST">
5 <input type="hidden" name="password" value="easy">
6 <input type="submit" value="Send malicious code!">
7 </form>
8
9 <script>
10 document.forms[0].submit();
11 </script>
12 </body>
```

Listing 8.4. "../csrf/attack.html"



**Figura 8.3.** Password cambiata con successo per conto dell'attaccante

## 8.4 Contromisure

La criticità di questo tipo di attacco deriva dal compromesso tra usabilità e sicurezza. Sebbene la soluzione più sicura consisterebbe nel richiedere le credenziali dell'utente per ogni richiesta effettuata, questa misura comprometterebbe l'esperienza utente. Tuttavia, esistono delle alternative valide.

### 8.4.1 Limitare l'applicazione di richieste GET

Una delle problematiche delle richieste *GET* è che i parametri vengono inseriti nell'URL, esponendo potenzialmente l'applicazione a manipolazioni da parte di un attaccante per eseguire un attacco *CSRF*. La soluzione migliore sarebbe evitare di utilizzare richieste *GET* per modificare dati sul server. Queste richieste dovrebbero essere utilizzate esclusivamente per **recuperare** dati, evitando così di esporre l'applicazione a tali rischi. Tuttavia, come dimostrato, l'uso di tecniche come *JavaScript* e form invisibili può comunque portare a un attacco, indipendentemente dal metodo della richiesta.

### 8.4.2 Token CSRF

I **token CSRF** rappresentano una soluzione efficace per prevenire richieste non autorizzate inviate a nome degli utenti. Un *token CSRF* è una stringa pseudocasuale generata dal server e associata alla sessione o alla richiesta dell'utente. L'uso di *token* per singola richiesta può essere problematico quando si utilizzano pulsanti di ritorno alla pagina precedente, in quanto il token potrebbe non essere più valido, generando un *alert di sicurezza*.

L'adozione di token legati alla sessione, invece, prevede che il server verifichi l'esistenza e la validità del token per ogni richiesta, confrontandolo con quello presente nella sessione dell'utente. Se i token non coincidono, la richiesta viene scartata.

Un token CSRF sicuro deve avere tre caratteristiche fondamentali:

- **Unico:** se due o più token sono uguali per sessioni diverse basta individuare uno per compromettere l'autenticità dell'utente
- **Segreto:** in modo da impedire che venga utilizzato da un attaccante per effettuare un attacco XSRF
- **Imprevedibile:** è importante che si crei un token con un valore randomico molto grande in modo da evitare che venga indovinato da un attaccante

Nel caso della nostra applicazione, iniziamo generando un token CSRF randomico e lo salviamo all'interno della sessione dell'utente:

```
2     if (empty($_SESSION['csrf_token'])) {  
3         $_SESSION['csrf_token'] = bin2hex(random_bytes(32));  
4     }
```

**Listing 8.5.** "../csrf/app.php"

Successivamente, per proteggere il form di cambio password dall'essere sfruttato come superficie di attacco CSRF, inseriamo il token all'interno del form come campo nascosto:

```
37     echo "<form method='POST'>";
38     echo "<input type='hidden' name='csrf_token' value='"
39     . $_SESSION['csrf_token'] . "'>";
40     echo "<input type='password' name='password'
41     placeholder='New password'>";
42     echo "<br><br>";
43     echo "<input type='submit' value='Change password'>";
44     echo "</form>";
```

Listing 8.6. "../csrf/app.php"

Infine, quando viene richiesta la modifica della password nel database, verificiamo la correttezza del token inviato, confrontandolo con quello salvato nella sessione. Se il controllo ha esito positivo, il sistema procede con l'aggiornamento della password; in caso contrario, la richiesta viene rifiutata.

```
45     if ($_SERVER['REQUEST_METHOD'] === 'POST') {
46
47         if (!isset($_POST['csrf_token'])
48             || $_POST['csrf_token'] !== $_SESSION['csrf_token']) {
49             die('CSRF token mismatch! Possible CSRF attack.');
```

Listing 8.7. "../csrf/app.php"

## 9 Conclusione

Lo studio ha avuto come obiettivo principale l'analisi delle principali vulnerabilità esistenti, come SQL Injection, Cross-Site Scripting (XSS), File Inclusion, Command Injection, Insecure Direct Object Reference (IDOR) e Cross-Site Request Forgery (CSRF), unitamente alle relative contromisure necessarie per mitigarne i rischi.

Attraverso un'implementazione pratica e l'utilizzo di tecnologie come Docker, è stato possibile simulare scenari di attacco su un ambiente controllato, esaminando da vicino come tali vulnerabilità possono essere sfruttate. Questo approccio sperimentale ha evidenziato non solo la pericolosità di tali attacchi, ma anche l'importanza di adottare misure preventive per mitigare gli attacchi informatici. Tra le principali contromisure identificate e analizzate vi sono la **validazione** e la **sanitizzazione** dell'input, la corretta **configurazione** dei server e dei file, il **monitoraggio** continuo delle risorse e l'implementazione di un adeguato meccanismo di **controllo degli accessi**.

Una riflessione finale è la consapevolezza che le minacce evolvono costantemente, così come devono evolvere le tecniche di difesa. In tal senso, la prevenzione rappresenta la prima linea di difesa, ma è altrettanto fondamentale essere in grado di **individuare** tempestivamente eventuali attacchi e saper rispondere con adeguati piani di **recovery** e **backup**.

In conclusione, questa tesi ha fornito una panoramica dettagliata sulle principali vulnerabilità delle applicazioni web e sulle relative tecniche di difesa, con l'obiettivo di offrire una piattaforma per simulare e sensibilizzare la pericolosità di questi attacchi.

# Bibliografia

- [1] William Stallings, Lawrie Brown, "*Computer Security: Principles and Practice*, Third Edition, Pearson, 2015
- [2] Mozilla Developer Network (MDN). "Content Security Policy (CSP)."  
<https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>
- [3] Himalaya College of Engineering, "SQL Injection Prevention and Mitigation," 2019.  
<https://www.hcoe.edu.np/uploads/attachments/r96oytechsacgzi4.pdf>
- [4] Hussein, Dina & Ibrahim, Dina & Alsalamah, Mona. (2021). A Review Study on SQL Injection Attacks, Prevention, and Detection. 13. 1-9.  
[https://www.researchgate.net/publication/362751864\\_A\\_Review\\_Study\\_on\\_SQL\\_Injection\\_Attacks\\_Prevention\\_and\\_Detection](https://www.researchgate.net/publication/362751864_A_Review_Study_on_SQL_Injection_Attacks_Prevention_and_Detection)
- [5] Ramasamy, Perumalsamy & Abburu, Sunitha. (2012). SQL INJECTION ATTACK DETECTION AND PREVENTION. International Journal of Engineering Science and Technology. 4.  
[https://www.researchgate.net/publication/267243666\\_SQL\\_INJECTION\\_ATTACK\\_DETECTION\\_AND\\_PREVENTION](https://www.researchgate.net/publication/267243666_SQL_INJECTION_ATTACK_DETECTION_AND_PREVENTION)
- [6] Gupta, S., Gupta, B.B. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. Int J Syst Assur Eng Manag 8 (Suppl 1), 512–530 (2017).  
<https://doi.org/10.1007/s13198-015-0376-0>
- [7] OWASP. Testing for Local File Inclusion.  
[https://owasp.org/www-project-web-security-testing-guide/v42/4-Web\\_Application\\_Security\\_Testing/07-Input\\_Validation\\_Testing/11.1-Testing\\_for\\_Local\\_File\\_Inclusion](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.1-Testing_for_Local_File_Inclusion)
- [8] Ninghui Li, "Command Injections," Software Security Course, Department of Computer Sciences, University of Wisconsin-Madison, 2007.  
[https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/3\\_8\\_2-Command-Injections.pdf](https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/3_8_2-Command-Injections.pdf)



- [9] OWASP. Command Injection.  
[https://owasp.org/www-community/attacks/Command\\_Injection](https://owasp.org/www-community/attacks/Command_Injection)
- [10] OWASP ModSecurity Core Rule Set Project. ModSecurity FAQ.  
[https://github.com/owasp-modsecurity/ModSecurity/wiki/ModSecurity-Frequently-Asked-Questions-%28FAQ%29#user-content-Configuring\\_ModSecurity](https://github.com/owasp-modsecurity/ModSecurity/wiki/ModSecurity-Frequently-Asked-Questions-%28FAQ%29#user-content-Configuring_ModSecurity)
- [11] OWASP. Insecure Direct Object Reference Prevention Cheat Sheet.  
[https://cheatsheetseries.owasp.org/cheatsheets/Insecure\\_Direct\\_Object\\_Reference\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html)
- [12] OWASP. Cross-Site Request Forgery (CSRF).  
<https://owasp.org/www-community/attacks/csrf>
- [13] David Wagner, "Cross-Site Request Forgery Attacks," University of California, Berkeley.  
<https://people.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf>
- [14] Chaohai Ding, "Cross-Site Request Forgery: Attack and Defence," University of Southampton, 2015.  
[https://www.southampton.ac.uk/~cd8e10/paper/INF06003\\_Cross\\_Site\\_Request\\_Forgery\\_Attack\\_And\\_Defence\\_Chaohai%20Ding.pdf](https://www.southampton.ac.uk/~cd8e10/paper/INF06003_Cross_Site_Request_Forgery_Attack_And_Defence_Chaohai%20Ding.pdf)
- [15] OWASP. Cross-Site Request Forgery Prevention Cheat Sheet.  
[https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.html#synchronizer-token-pattern](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#synchronizer-token-pattern)
- [16] Repository Github del progetto.  
<https://github.com/Nabster101/CyberPlayground>

# Ringraziamenti

Posso affermare con certezza che questo è stato un percorso lungo e sofferto, ma che mi ha dato una sensazione di realizzazione senza eguali. In questo periodo della mia vita ho imparato a gestire meglio il mio tempo, a dare la giusta priorità alle cose e a rialzarmi nei momenti più difficili.

In primo luogo, ringrazio i miei genitori, Alessandra Silvi e Enrico Costanzi Fantini, per essere stati sempre al mio fianco. Il loro supporto, la loro pazienza e il loro affetto sono stati fondamentali per arrivare fin qui. Senza di loro, non sarei riuscito a raggiungere questo traguardo.

Un ringraziamento speciale va anche ai miei amici, nonché compagni di vita, Mattia Cecamore, Federico Gerardi e Gabriele Onorato, per avermi ascoltato, incoraggiato e aiutato nei momenti più difficili. Non ce l'avrei fatta senza le nostre chiacchierate e le risate che ci siamo fatti durante queste giornate intense.

Ringrazio anche Soykat Amin, Cristian Di Iorio, Cristian Apostol, Leonardo Miralli e Giuseppe Marchio, che mi hanno accompagnato in questo percorso. È stato un piacere condividere con loro momenti belli che mi hanno sempre dato la forza di andare avanti.

Infine, un grazie di cuore a tutte le persone che, in un modo o nell'altro, hanno reso questi anni speciali e mi hanno aiutato a crescere, sia a livello personale che professionale.