Immagine che contiene testo, Carattere, simbolo, emblema

Descrizione generata automaticamente

Dipartimento di Informatica

Corso di laurea “Sicurezza Informatica”

Sicurezza nelle Applicazioni 2024/2025 Prof. **Pasquale Ardimento**

**Membri**: Schiralli Gaetano

**Obiettivo Progetto:** Implementare un'applicazione web sicura in Java, dimostrando di saper gestire correttamente le vulnerabilità comuni nelle applicazioni web. Ogni sezione deve essere sviluppata in modo sicuro, seguendo le *best practice* di sicurezza indicate nel corso. L'applicazione dovrà essere progettata per gestire in modo sicuro i dati sensibili e prevenire attacchi.

**REQUISITI FUNZIONALI E DI SICUREZZA RISPETTATI**

* **Registrazione Utente con Foto Profilo:**
  + **Permette la registrazione con e-mail valida, password sicura e conferma password**: Sono stati implementati metodi comuni anche per la fase di login, in cui vengono sanitizzati e validati gli input garantendo che siano conformi ai requisiti di sicurezza. Nella classe **InputSanitizer** troviamo:
    - **isValidEmail** che usa una regex per accettare solo e-mail valide trasformate tutte in minuscolo.
    - **isValidPassword** che valida la password, controllando la lunghezza, i caratteri maiuscoli, minuscoli, i numeri ed i caratteri speciali richiesti.
    - **noInvalidCharacters** che controlla delle Blacklist di caratteri proibiti**.** Tale metodo rileva e blocca nella password, caratteri pericolosi (< > ' " \ ;), prevenendo vulnerabilità come **SQL Injection o XSS**.

Nella servlet di registrazione o **RegisterServlet**, una volta confrontata la password con la conferma password, con **BCrypt** si trasforma la password in una stringa cifrata che può essere salvata in modo sicuro nel database. Anche con la stessa password, l'hash sarà diverso ogni volta grazie al metodo *BCrypt.gensalt()* che ogni volta genererà un salt casuale aggiunto alla password per renderla più resistente agli attacchi di tipo **dizionario o rainbow table**.

* + **Consente il caricamento di un'immagine di profilo nei formati .png, .jpeg, o .jpg con validazione del formato.** Sempre all’interno di **RegisterServlet** viene recuperato il file caricato dall'utente attraverso il campo *profileImage* del modulo HTML. Utilizzando il metodo *profileImagePart.getInputStream(),* viene letto il contenuto del file direttamente come *InputStream*. Questo approccio è più sicuro rispetto al salvataggio del file in una posizione temporanea, mitigando attacchi di tipo **TOCTOU (Time of Check to Time of Use**). Inoltre, con **Apache Tika**, si rileva il tipo **MIME** effettivo del file analizzandone il contenuto binario determinando il tipo reale del file, indipendentemente dall’estensione. In seguito, se il file non rispetta entrambi i criteri (estensione e tipo MIME), l'operazione viene interrotta con un messaggio di errore: *"Invalid file. Only PNG, JPG, and JPEG images are allowed”.* Infine, dopo la validazione, l'*InputStream* viene resettato per essere utilizzato successivamente.

Tutto questo previene che un attaccante possa caricare file malevoli camuffati come immagini (es., un file .exe rinominato in .png) oppure manipolare i file temporanei creati dal server durante il caricamento, sostituendo il file originale con un file malevolo tra la fase di controllo e quella di utilizzo. Inoltre, viene fornito un riscontro chiaro all'utente per migliorare l'esperienza.

Successivamente viene riempito nel database il campo *email, password\_hash, profile\_image* (BLOB) e *created-at*, che mantiene le info di data e ora creazione account.

* **Autenticazione Tramite Login:**
  + **Login con e-mail e password registrati.** Se l’opzione *RememberMe* non è selezionata dall’utente, nella **LoginServlet** si ripetono gli stessi controlli sull’input che avvengono in fase di registrazione. Inoltre, con il metodo *authenticateUser* si verifica se un utente può autenticarsi, confrontando l’e-mail e la password fornite, con i dati memorizzati nel database. Se l’e-mail esiste nel database, il metodo estrae l'hash della password memorizzata associata a quell'email. Successivamente, utilizza la funzione *BCrypt.checkpw*, che confronta la password inserita dall'utente con l'hash salvato nel database. Se questo confronto tra la password fornita e l'hash salvato nel database ha esito positivo, il metodo restituisce *true*, indicando che l'utente è autenticato. Altrimenti, restituisce false, che può accadere se l’e-mail non esiste, se la password non corrisponde o se si verifica un errore durante l'esecuzione della query.
  + **Opzione per utilizzare i cookie per mantenere la sessione attiva.** Se le credenziali sono corrette, sempre nella stessa servlet di login, si procede alla gestione della sessione e dei cookie. Per prima cosa, controlla se esiste già una sessione attiva. Se la sessione corrente non è nuova, la invalida utilizzando *session.invalidate()* e ne crea una nuova per garantire una connessione sicura e isolata.

Successivamente, controlla se l'utente ha richiesto l'opzione "*Remember Me*" (ricorda l'accesso). Se questa opzione è attiva, viene creato un cookie sicuro contenente l’e-mail dell'utente crittografata. La crittografia avviene con una funzione dedicata chiamata *encrypt(e-mail),* che garantisce che i dati salvati nel cookie non siano leggibili direttamente. Questo cookie viene configurato con diverse opzioni di sicurezza:

* + - **HttpOnly**, per impedire l'accesso via JavaScript; Questo previene attacchi di tipo **Cross-Site Scripting (XSS)**, in cui un attaccante potrebbe sfruttare vulnerabilità del sito per accedere ai cookie dell'utente e rubare informazioni sensibili, come i token di sessione.
    - **Secure**, per consentirne la trasmissione solo su connessioni HTTPS. Questo impedisce che il cookie venga intercettato da attaccanti durante la trasmissione in reti non sicure, come connessioni HTTP non criptate, riducendo il rischio di attacchi di tipo **Man-in-the-Middle (MitM).**
    - **SameSite=Strict**, impedisce al cookie di essere inviato insieme alle richieste provenienti da altri siti. Questo protegge contro attacchi **CSRF (Cross-Site Request Forgery)**, in cui un sito malevolo tenta di sfruttare le sessioni autentiche di un utente per inviare richieste non autorizzate al server.
    - Inoltre, il cookie ha un percorso globale (/), un timeout di un giorno e viene aggiunto alla risposta HTTP.

Infine, la sessione viene associata all'utente impostando l'attributo "*user*" con l’e-mail dell'utente. Per maggiore sicurezza, il timeout della sessione viene impostato a 15 minuti, garantendo che l'utente debba autenticarsi nuovamente dopo un periodo di inattività se non ha scelto di salvare i cookie.

* **Caricamento Proposta Progettuale:**
  + **Utente autenticato può caricare una proposta progettuale in formato .txt**: In questa servlet, **UploadProposalServlet,** si gestisce il caricamento sicuro di file .txt contenenti proposte progettuali, assicurando che siano salvate correttamente nel database e rispettando i requisiti di sicurezza.

Come per le immagini, per garantire che solo file di testo validi vengano caricati, la servlet utilizza la libreria **Apache Tika**, che analizza il contenuto del file per determinarne il tipo MIME. In questo caso, il tipo atteso è *text/plain*. Oltre al tipo MIME, il nome del file viene controllato per assicurarsi che termini con .txt. Se una di queste condizioni non è soddisfatta, la servlet interrompe l'elaborazione e restituisce un errore, evitando il caricamento di file non conformi.

Il caricamento e l'elaborazione del file vengono eseguiti in un blocco sincronizzato per mitigare gli attacchi **TOCTOU**. Utilizzando il blocco *synchronized*, la servlet garantisce che il file validato sia lo stesso che viene successivamente elaborato.

Una volta validato, il contenuto del file viene salvato nel database. Il metodo *saveProposalToDatabase* esegue un aggiornamento nella tabella users, associando il contenuto del file al campo *project\_proposals* dell'utente identificato dall'email. Il contenuto del file viene caricato come un flusso di byte (*InputStream*) e salvato utilizzando il metodo *setBlob* del **PreparedStatement**. Questo approccio garantisce che i dati siano memorizzati in modo sicuro e che non vengano manipolati durante il trasferimento.

* **Visualizzazione delle Proposte Progettuali:**
  + **Gli utenti autenticati con sessione valida, possono visualizzare le proposte caricate da altri utenti.** La servlet **ViewProposalsServlet** è divisa in due principali funzionalità: mostrare l'elenco di tutte le proposte disponibili o mostrare il contenuto di una singola proposta specifica.La servlet utilizza il parametro *action* per determinare quale operazione eseguire. Se il valore di action è "*viewProposal*", la servlet tenta di visualizzare una proposta specifica, utilizzando il parametro *proposalId* per identificare la proposta richiesta. Se invece action è nullo o diverso, mostra l'elenco di tutte le proposte disponibili.

Il metodo *showAllProposals* si occupa di recuperare tutte le proposte dal database. Utilizzando una query SQL, estrae l’e-mail dell'utente e il contenuto del campo *project\_proposals*. Per ogni proposta trovata, aggiunge un riferimento all'elenco *proposals*, includendo l’e-mail dell'utente e un link alla servlet con l'azione *viewProposal* e il relativo *proposalId*. Questa lista di proposte viene poi passata alla pagina JSP *viewProposals.jsp* attraverso l'attributo della richiesta. Se si verifica un errore durante il recupero delle proposte, la servlet registra l'errore e reindirizza l'utente alla home page con un messaggio appropriato.

Il metodo *showProposal* gestisce la visualizzazione del contenuto di una proposta specifica. Inizia verificando che il valore di *proposalId* sia valido (non nullo o vuoto) per prevenire input non validi o attacchi di SQL Injection. Successivamente, utilizza una query SQL per recuperare il contenuto della proposta associata a quell'ID. Il contenuto del file viene convertito da byte a stringa e sanitizzato utilizzando il metodo *sanitizeContent*. Questo metodo utilizza la libreria **Apache Commons Text** per escapare i caratteri HTML, prevenendo vulnerabilità di tipo **XSS (Cross-Site Scripting)**. Infine, il contenuto della proposta viene passato alla pagina JSP *viewSingleProposal.jsp* per essere mostrato all'utente.

Il metodo *sanitizeContent* garantisce che qualsiasi contenuto testuale proveniente dall'utente o dal database venga visualizzato in modo sicuro. Utilizzando la funzione **escapeHtml4 di Apache Commons Text**, converte caratteri HTML come <, > e " nei rispettivi codici *HTML (&lt;, &gt;, &quot;)*, impedendo al browser di interpretare il contenuto come codice eseguibile.

* **Logout Sicuro:**
  + **Consentire agli utenti di effettuare il logout in modo sicuro.** La servlet **LogoutServlet** è responsabile della gestione del logout dell'utente, assicurandosi che tutte le tracce dell'autenticazione dell'utente, come la sessione e i cookie, siano rimosse in modo sicuro.

Quando un utente effettua una richiesta per il logout, il primo passo della servlet è ottenere la sessione corrente. Questo viene fatto tramite *request.getSession(false),* che restituisce la sessione esistente senza crearne una nuova. Se esiste una sessione, viene invalidata utilizzando *session.invalidate().* Questo assicura che tutti gli attributi associati alla sessione, inclusi i dati utente, vengano rimossi.

Successivamente, la servlet controlla la presenza del cookie *rememberMe*. I cookie vengono recuperati con il metodo *request.getCookies().* Se il cookie *rememberMe* è presente, la servlet ne imposta il tempo di vita (*MaxAge*) a 0 e definisce il percorso (*Path*) come /, lo stesso utilizzato durante la sua creazione. Questo garantisce che il cookie venga effettivamente eliminato dal browser. Il cookie modificato viene quindi aggiunto alla risposta HTTP con *response.addCookie(cookie*). Dopo aver eliminato la sessione e i cookie, la servlet imposta un messaggio di conferma del logout (*logoutMessage*) come attributo della richiesta. Infine, la servlet reindirizza l'utente alla pagina di login. Questo approccio garantisce che nessun dato di sessione o autenticazione persista dopo il logout, prevenendo potenziali abusi o accessi non autorizzati.

* **Feedback per le Operazioni:**
  + **Fornire messaggi chiari per le operazioni:** 
    - **Registrazione**: Conferma di successo o errore (es. "*Email already used*").
    - **Login**: Esito positivo o negativo (es. "*Invalid email or password*").
    - **Caricamento Proposta**: Conferma o errore sul formato file.
    - **Sessione**: Stato della sessione (es. "*Logout completed succesfully*").
* **Gestione dei Cookie:**
  + **Impostare i cookie con opzioni di sicurezza, implementare la cancellazione di cookie non più validi o scaduti, impostare un timeout adeguato per i cookie e rinnovarli quando necessario, salvare i dati sensibili nei cookie utilizzando la crittografia:**

(già spiegato in precedenza nelle varie servlet)

* + **Gestire la ricezione dei cookie:** La classe **RememberMeFilter** è un filtro Java Servlet che gestisce il meccanismo di autenticazione automatica degli utenti tramite cookie. La sua funzione principale è verificare se un utente non autenticato ha un cookie valido per essere autenticato automaticamente.

Quando una richiesta HTTP viene ricevuta, il filtro controlla innanzitutto se esiste già una sessione attiva per l'utente. Questo viene fatto recuperando la sessione corrente tramite *httpRequest.getSession(false*), che restituisce *null* se non c'è una sessione attiva. Se la sessione è valida e contiene un attributo *user*, l'utente è considerato autenticato, e il filtro lascia proseguire la richiesta senza ulteriori controlli.

Se l'utente non è autenticato (cioè, la sessione è nulla o non contiene l'attributo user), il filtro passa a controllare i cookie associati alla richiesta. Utilizzando *httpRequest.getCookies(),* il filtro scansiona tutti i cookie ricevuti alla ricerca di un cookie denominato "*rememberMe*". Se trova questo cookie, procede alla decifratura del suo valore.

Il metodo *decrypt* utilizza **AES** per decifrare il valore del cookie. Prima di tutto, carica una chiave segreta dal **keystore** tramite il metodo *KeyManager.loadSecretKey().* Poi, configura un cifrario *AES* in modalità di decifratura (*DECRYPT\_MODE*) e decodifica i dati codificati *in Base64*. Se la decrittazione riesce, il filtro ottiene l’e-mail memorizzata nel cookie.

Dopo aver recuperato l’e-mail dal cookie, il filtro verifica se l'utente esiste nel database tramite il metodo *isValidUser*. Questo metodo esegue una query SQL per controllare se l’e-mail è presente nella tabella degli utenti. Se l’e-mail è valida e corrisponde a un utente registrato, il filtro crea una nuova sessione per autenticare l'utente.

Se il filtro determina che l'utente è valido, crea una nuova sessione utilizzando. Imposta l'attributo *user* con l’e-mail dell'utente e definisce un *timeout* di inattività per la sessione (15 minuti).

Immagine che contiene testo, software, Software multimediale, schermata

Descrizione generata automaticamente

* **Gestione Sicura delle Chiavi e Crittografia**
  + Questa viene gestito da una classe chiamata **KeyManager**. Essa è responsabile della gestione della configurazione e del caricamento di una chiave segreta da un ***keystore***, utilizzata per operazioni crittografiche come la gestione sicura dei cookie. La classe è progettata per garantire un caricamento sicuro ed efficiente della chiave con controlli rigorosi.

La configurazione necessaria per la classe, come il percorso del keystore, la password e l'alias della chiave, viene caricata da un file di configurazione chiamato *cookie.properties*. Questo avviene nel metodo statico *loadProperties*, che viene eseguito automaticamente una volta che la classe *KeyManager* è caricata in memoria.

In *loadProperties*, il codice apre il file *cookie.properties* utilizzando il class loader e legge i valori di configurazione utilizzando la classe *Properties*.

Il metodo principale della classe è *loadSecretKey*, che carica una chiave segreta dal *keystore*. Per fare ciò, verifica prima che le proprietà richieste siano state caricate correttamente. Se i parametri sono validi, il metodo apre il *keystore*, lo carica in memoria e recupera la chiave segreta utilizzando l'alias e la password forniti.

Questo approccio è altamente sicuro perché protegge le informazioni sensibili utilizzando un keystore standard **Java (JCEKS**).

* **Connessione al Database:**
  + La classe **DBConnection** carica le configurazioni dal file *db.properties*, che contiene i dettagli di connessione al database. Oltre ai parametri tradizionali come *db.url, db.user, e db.password*, qui si specifica anche l’utilizzo del **SSL** che garantisce che i dati, dal web browser fino alla loro manipolazione e salvataggio in DB viaggino in modo crittografato e quindi siano più al sicuro da possibili attacchi anche in questa fase.

Quando il metodo *getConnection()* viene chiamato, utilizza il driver *JDBC* per stabilire una connessione sicura con il database. Il driver utilizza **SSL/TLS** per crittografare la comunicazione, garantendo che i dati trasmessi non possano essere intercettati.

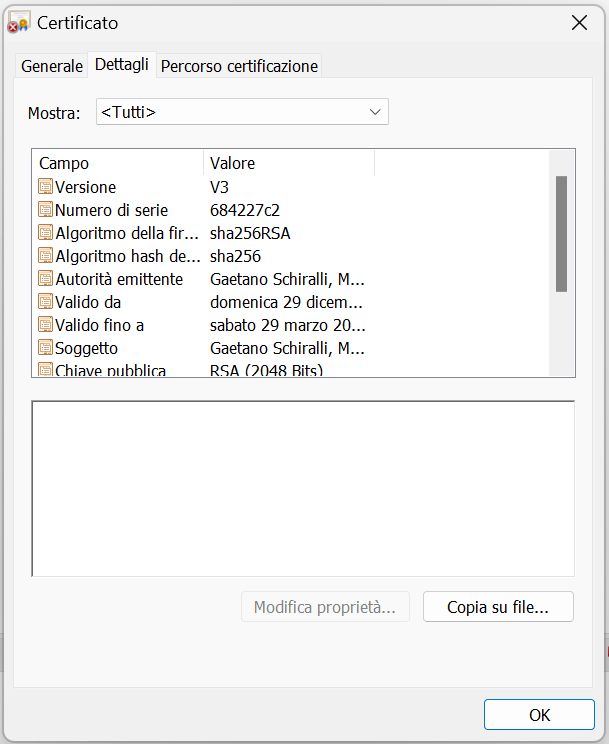
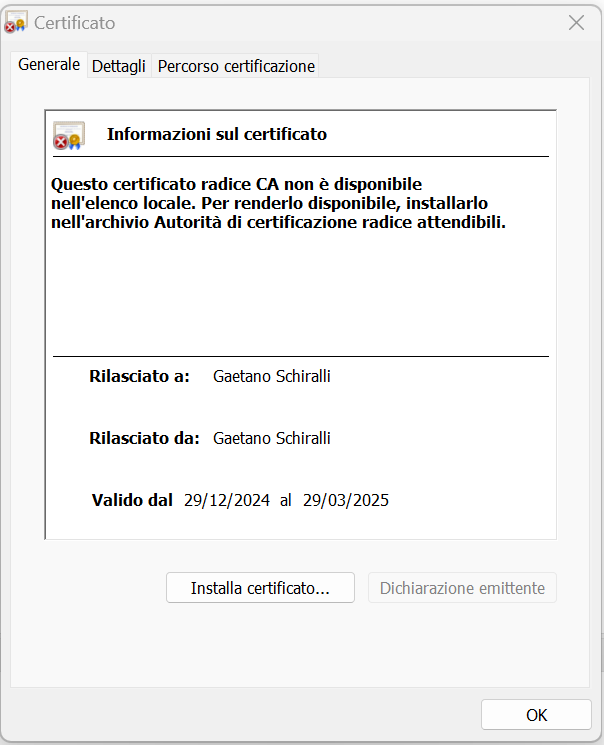
Il metodo *closeConnection()* chiude la connessione al database in modo sicuro, registrando un messaggio in caso di successo o errore.

* + Ogni qual volta nel progetto si interroga il database, vengono utilizzate le **Prepared Statements**. Esse sono utilizzate per garantire la sicurezza e l'efficienza delle interazioni con il database. Proteggono da attacchi di SQL Injection, separando il codice SQL dai dati forniti dall'utente, impedendo che questi ultimi vengano interpretati come comandi SQL. Questo approccio rende il codice più sicuro e robusto, soprattutto quando si gestiscono input non controllati. Inoltre, migliorano le prestazioni grazie al riutilizzo di query precompilate, rendendo le operazioni sul database più veloci e semplificando la manutenzione del codice.

**Gestione della Sessione HTTP**

* + È stato creato un certificato **SSL/TLS self-signed** che è un certificato digitale creato direttamente dall'organizzazione o dal singolo individuo, anziché da un'autorità di certificazione (CA) riconosciuta. Questo tipo di certificato viene utilizzato per abilitare la comunicazione **HTTPS** nel mio caso sul server, crittografando i dati tra il server e il client per garantire la sicurezza e la privacy delle informazioni trasmesse.

Questo è utile in contesti di sviluppo, test o ambienti interni, dove non è necessario acquistare certificati emessi da CA. Il certificato garantisce che la connessione sia crittografata e protegga i dati da intercettazioni o attacchi di tipo **Man-in-the-Middle**. Sebbene non offra la validazione dell'identità del server (poiché non proviene da una CA), fornisce una base per testare e implementare la crittografia SSL/TLS in modo sicuro ed economico.



* **Programmazione Difensiva:**
  + È stato minimizzato lo scope delle variabili per limitare il loro utilizzo al minimo necessario
  + Si è l'accessibilità di classi, metodi e variabili con modificatori di accesso appropriati (private, protected, default).

**TEST D’USO**

* **Registrazione andata a buon fine:**

**Immagine che contiene testo, schermata, Carattere

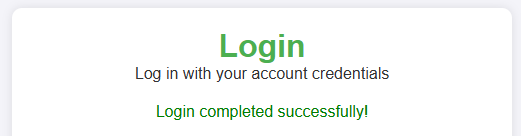
Descrizione generata automaticamente**

* **Login con dati di accesso errati:**

**Immagine che contiene testo, schermata, Carattere, logo

Descrizione generata automaticamente**

* **Login andato a buon fine:**

****

* **Pagina Home una volta loggati:**

**Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente**

* **Caricamento proposta progettuale KO:** la prima in formato .html contenente oggetti come immagini, eventuali link, bolded text (sensibile ad attacchi di tipo XSS). La seconda in formato .pdf

**Immagine che contiene testo, Carattere, logo, design

Descrizione generata automaticamente**

* **Caricamento proposta progettuale OK:**

**Immagine che contiene testo, Carattere, schermata, design

Descrizione generata automaticamente**

* **Visualizzazione lista delle proposte progettuali caricate:**

**Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente**

* **Visualizzazione contenuto di una specifica proposta progettuale caricata:**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

* **Istante del Logout:**

Immagine che contiene testo, Carattere, schermata, design

Descrizione generata automaticamente

* **Per la session timeout e il cookie scaduto, così come per l’accesso diretto alla home o alla pagina di visualizzazione delle proposte progettuali senza autenticazione o cookie il sito mostra la pagina di login:**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

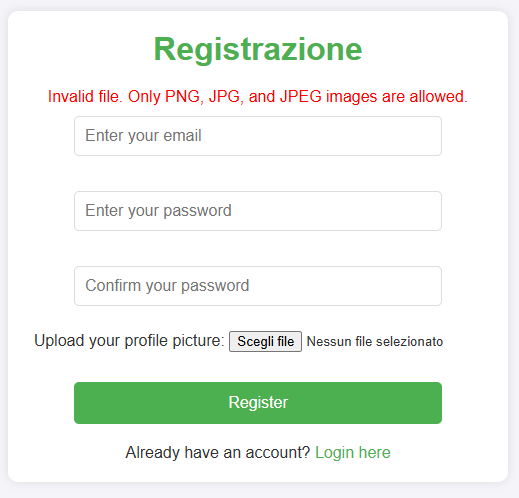
**TEST ABUSO**

* **Registrazione con file non valido (.exe) e registrazione con file non valido (fake .jpeg):** Un file .exe è un file eseguibile, che potrebbe essere usato per distribuire malware o eseguire codice malevolo sul server o sui client. Se l'applicazione non controllasse il tipo di file caricato, un attaccante potrebbe caricare un file .exe camuffato come parte del processo di registrazione.

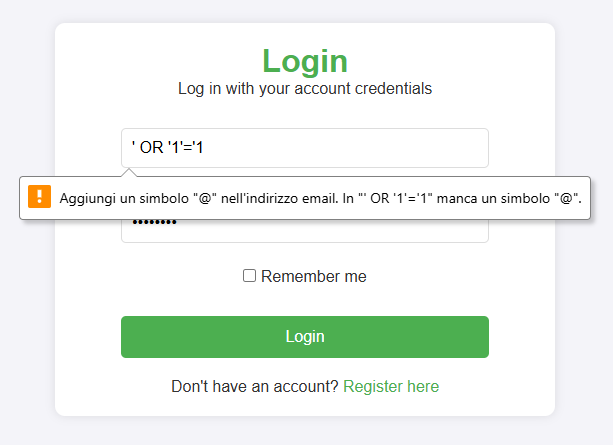
Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

L'applicazione blocca tutti i file non consentiti. In questo caso lo stesso messaggio di errore viene mostrato sia quando il malintenzionato prova a caricare un file .exe, e sia quando prova a caricare una fake.jpeg, in fase di registrazione.



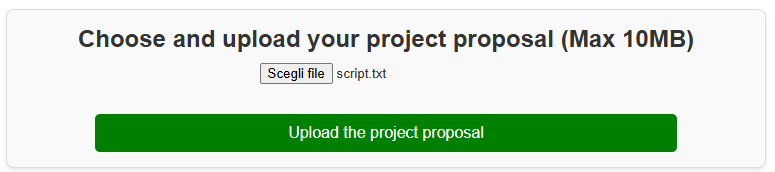
* **Login bypass via SQL Injection su e-mail:** L'inserimento di una stringa come ' OR '1'='1 in un campo di input (ad esempio, un campo e-mail) è un tentativo di SQL Injection. Questo tipo di attacco sfrutta query SQL vulnerabili per manipolare o bypassare i controlli di autenticazione o per eseguire operazioni non autorizzate sul database.



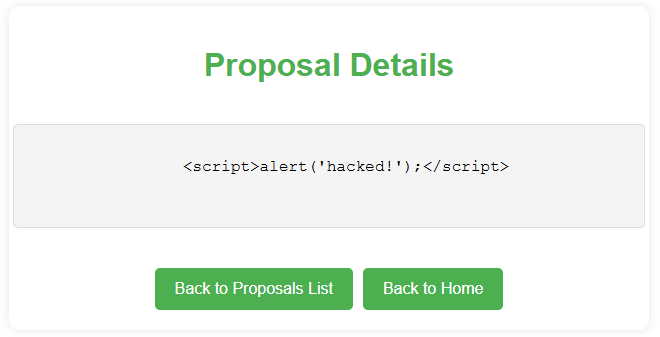
Anche se le PreparedStatement sono sicure, è buona pratica validare e sanitizzare gli input utente per evitare input non validi come in questo caso.

* **Caricamento di un file .txt di proposta progettuale e contenente script/codice per attacco stored XSS (ad esempio <script> alert(‘hacked!’) </script>):**

Lo Stored XSS (Cross-Site Scripting Memorizzato) è un tipo di vulnerabilità in cui un attaccante inietta codice malevolo (tipicamente JavaScript) che viene memorizzato permanentemente sul server dell'applicazione web. Questo codice viene poi eseguito nei browser degli utenti che visitano le pagine colpite.



Quando un altro utente accede a una pagina che carica il contenuto memorizzato, il browser interpreta ed esegue il codice iniettato come parte della pagina web. Il codice malevolo può leggere i cookie di sessione dell'utente, token di autenticazione, o altre informazioni sensibili e inviarle all'attaccante



La pagina Web come si vede in foto, quando mostra i dati memorizzati nel browser dell'utente, esegue **l’escaping** dei caratteri HTML e previene l'interpretazione del codice.

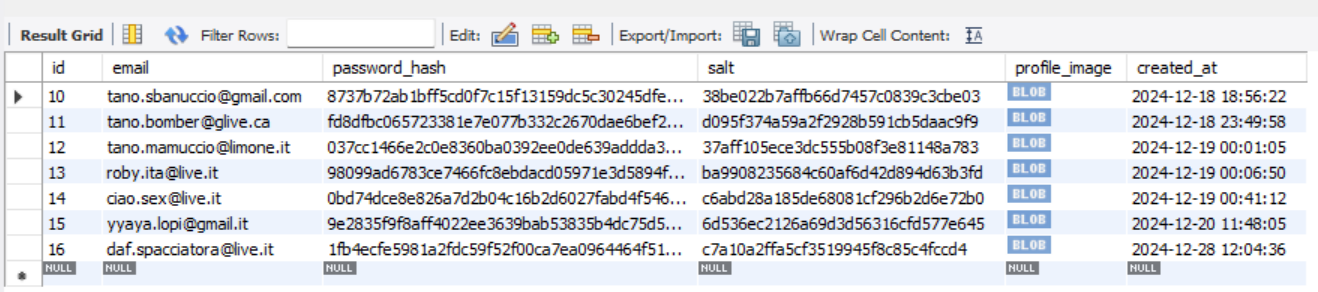
**SCELTE PROGETTUALI**

**Cambio da SHA-256 + Salt a BCrypt per il salvataggio delle password nel database:** Se si utilizza **BCrypt**, non è necessario gestire manualmente il **salt**. Questo perché BCrypt genera automaticamente un salt univoco per ogni password, lo incorpora nell'hash generato e lo salva insieme all'hash stesso. Di conseguenza, il campo salt nel database diventa superfluo.

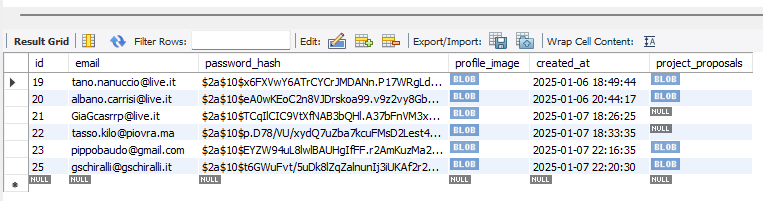
**Confronto con SHA-256 + Salt**

| **Aspetto** | **SHA-256 + Salt** | **BCrypt** |
| --- | --- | --- |
| **Salt** | Deve essere generato manualmente. | Generato automaticamente e incluso nell'hash. |
| **Complessità del codice** | Elevata | Molto ridotta |
| **Resistenza agli attacchi brute force** | Dipende dal design | Molto elevata (es. forza lavoro incorporata) |
| **Adattabilità** | Difficile da aggiornare | Supporta l'aumento del costo computazionale. |

* **Struttura Database con SHA-256 + Salt:**

****

* **Struttura Database finale con BCrypt:**

****

**Come funziona BCrypt?**

1. **Generazione del Salt**: Quando chiami BCrypt.hashpw(), viene generato un salt casuale automaticamente.
2. **Incorporazione del Salt**: Il salt viene incluso direttamente nell'hash risultante.
3. **Verifica**: Durante il controllo della password (BCrypt.checkpw()), il salt viene estratto dall'hash e utilizzato per ricreare l'hash della password fornita, rendendo il confronto affidabile e sicuro.

**Struttura di un hash BCrypt**

Un hash generato da BCrypt ha una struttura simile a questa:

**$2b$10$E9j1.GtQ2zH60Qo3J3sqPuC/R1cY99aV/DiO1IeqKz2Fd5hjBGDsm**

* **$2b**: Indica l'algoritmo BCrypt.
* **10**: Indica il costo computazionale (in questo caso, 10).
* **E9j1.GtQ2zH60Qo3J3sqPu**: Salt generato automaticamente.
* **R1cY99aV/DiO1IeqKz2Fd5hjBGDsm**: Hash della password con il salt.

Nella realtà se avessi già registrato tutti gli utenti con un approccio basato su SHA-256 + Salt, avrei dovuto migrare gradualmente gli utenti esistenti aggiornando l'hash al primo login o forzando un reset della password. Questo avrebbe garantito che tutte le password utilizzino lo standard più sicuro (BCrypt).