Report Tecnico sulle Vulnerabilità Identificate nel Sistema Web

Introduzione

Il presente report fornisce un'analisi approfondita su diverse vulnerabilità rilevate all'interno di un'applicazione web, basandosi sui test di sicurezza effettuati. Sono stati individuati diversi punti deboli che potrebbero compromettere la riservatezza, l'integrità e la disponibilità del sistema e dei dati trattati. Verranno descritti gli attacchi, i rischi associati, prove pratiche, valutazioni di rischio e le relative contromisure.

Matrice di Rischio

ID	Vulnerabilità	Impatto	Probabilità	Livello di Rischio
1	Cross-Site Scripting (XSS)	Alto	Media	Alto
2	Accesso a file via URL FTP	Alto	Media	Alto
3	Esposizione endpoint `/metrics`	Medio	Alta	Medio-Alto
4	Manipolazione parametri HTTP (es. Rating Tampering)	Medio	Alta	Medio-Alto
5	Manipolazione URL con encoding (`%23` per `#`)	Medio	Alta	Medio-Alto
6	SQL Injection (es. login bypass)	Critico	Alta	Critico
7	Accesso pannello admin via URL diretta (`/administration`, IDOR)	Critico	Alta	Critico
8	Email leak tramite endpoint `/whoami` + risposta JSONP	Alto	Media-Alta	Alto

- Critico = impatto potenzialmente distruttivo o compromissione completa del sistema

- Media-Alta = Probabilità concreta ma non costante (es. exploitabilità da utenti autenticati o tramite chaining)

- I livelli di rischio sono determinati tramite l'incrocio tra Probabilità e Impatto

1. Attacco XSS (Cross-Site Scripting)

Descrizione della vulnerabilità

L'attaccante ha testato un'iniezione di codice HTML e JavaScript nel sito. Inizialmente, il codice HTML non veniva eseguito direttamente a causa della natura della Single Page Application (SPA). Tuttavia, l'attaccante ha scoperto un metodo per eseguire codice JavaScript utilizzando la manipolazione dell'attributo `onerror` di un elemento `<img>`, iniettando codice malevolo come:

HTML
(Copia Modifica)

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

o

HTML
(Copia Modifica)

```
<img src=x onerror="window.location='http://attacker.com';">
```

Questa vulnerabilità può essere sfruttata per eseguire codice arbitrario nel browser dell'utente, potenzialmente rubando dati sensibili (cookie, sessioni) o reindirizzandolo a siti dannosi.

Valutazione del rischio: ALTO

- **Impatto**: Elevato (può portare al furto di dati, session hijacking, defacement del sito).
- **Probabilità**: Media (dipende da come vengono gestiti gli input utente e l'output HTML).

Mitigazioni

- **Sanitizzazione e validazione degli input**: Rimuovere o codificare caratteri pericolosi (ad esempio, `< > " ' &`) prima di renderizzarli.

- **Content Security Policy (CSP)**: Implementare una CSP rigorosa per impedire l'esecuzione di script inline non autorizzati.
- **HttpOnly e Secure cookies**: Proteggere i cookie di sessione per prevenire il furto tramite XSS.
- **Escaping lato client**: Assicurarsi che i dati inseriti dagli utenti vengano sempre trattati come testo e non come codice eseguibile.

---

2. Accesso a un file tramite FTP via URL

Descrizione della vulnerabilità

L'attaccante ha identificato che è possibile accedere a file presenti in un server FTP utilizzando un URL diretto. Ciò indica che il servizio FTP potrebbe essere esposto senza adeguate restrizioni di accesso.

Valutazione del rischio: ALTO

- **Impatto**: Elevato (possibile esposizione di file sensibili, accesso non autorizzato, data leakage).
- **Probabilità**: Media (dipende dalla configurazione del server e dalla protezione dell'accesso).

Mitigazioni

- **Disabilitare l'accesso anonimo** al server FTP.
- **Restrizione degli accessi**: Consentire l'accesso solo a IP autorizzati.
- **Cifratura del traffico**: Usare SFTP o FTPS invece di FTP per proteggere i dati in transito.
- **Protezione tramite firewall**: Bloccare porte FTP non necessarie e limitare l'accesso solo ai servizi interni.

---

3. Esposizione di metriche tramite `/metrics`

Descrizione della vulnerabilità

L'attaccante ha scoperto che è possibile accedere a `/metrics`, il che suggerisce che le metriche di sistema e dell'applicazione potrebbero essere esposte pubblicamente. Questo può rivelare informazioni sensibili sullo stato del sistema, come utilizzo delle risorse, nome dei servizi in esecuzione e potenzialmente credenziali o token di accesso.

Valutazione del rischio: MEDIO-ALTO

- **Impatto**: Medio (potrebbe facilitare attacchi mirati e movimenti laterali).
- **Probabilità**: Alta (se le metriche sono esposte senza autenticazione).

Mitigazioni

- **Restrizione degli accessi**: Proteggere `/metrics` con autenticazione o accesso solo da IP specifici.
- **Rimozione di informazioni sensibili** dalle metriche prima di esporle.
- **Rate limiting e logging**: Monitorare gli accessi e limitare richieste eccessive.

Fonte: FEEDBACK A 0 STELLE

1. Manipolazione dei parametri HTTP (Tampering)

Descrizione della vulnerabilità

L'attaccante ha intercettato una richiesta HTTP utilizzando **Burp Suite**, uno strumento per l'analisi e la modifica del traffico web. Ha quindi alterato un parametro che rappresenta il numero di stelle (probabilmente in un sistema di rating o recensioni). Questo suggerisce che

l’applicazione non valida o protegge adeguatamente i parametri ricevuti dal client, permettendo a un utente malintenzionato di manipolare i dati inviati al server.

Prova pratica:
ho intercettato il traffico con **BURP**, ho intercettato la richiesta **http** ed ho modificato il parametro che indica il numero delle stelle. Vulnerabilita con difficoltà a una stella

Possibili impatti

- **Falsificazione di recensioni e punteggi**: Un utente potrebbe attribuirsi un rating più alto o penalizzare altri utenti/servizi in modo fraudolento.

- **Dati inconsistenti nel database**: Se non viene implementata alcuna verifica server-side, il database potrebbe contenere dati alterati e inattendibili.

- **Possibili attacchi più avanzati**: La mancanza di validazione lato server potrebbe consentire ulteriori attacchi, come **SQL Injection**, se il valore manipolato viene usato in una query senza adeguata protezione.

Valutazione del rischio: MEDIO-ALTO

- **Impatto**: Medio (dipende dalla criticità del sistema di rating e dal suo utilizzo).

- **Probabilità**: Alta (se l’applicazione non implementa controlli sui parametri inviati dal client).

Mitigazioni e contromisure

Per arginare questa vulnerabilità, il provider del servizio dovrebbe adottare le seguenti misure:

1. **Validazione lato server**

- Il server deve verificare che il valore del parametro rientri in un intervallo accettabile (es. da 1 a 5 per un sistema di rating a stelle).

- Non fidarsi mai di dati inviati dal client senza una verifica lato server.

2. **Meccanismi di autenticazione e autorizzazione**

- Se il rating è legato a un utente specifico, assicurarsi che solo utenti autenticati possono modificarlo.

- Impedire che un utente possa modificare il rating di altri utenti senza autorizzazione.

3. **Protezione contro la manipolazione del traffico**

- Implementare **firma digitale o HMAC** sui parametri critici, così eventuali modifiche client-side renderebbero il valore non valido.

- Utilizzare **Token CSRF** per proteggere da richieste non autorizzate.

4. **Logging e monitoraggio**

- Registrare tentativi di modifica sospetti nei log.

- Implementare un sistema di alert per identificare modifiche anomale nei parametri inviati al server.

Fonte: MISSING ENCODING

1. Manipolazione degli URL e Encoding (Bypass delle Restrizioni URL)

Descrizione della vulnerabilità

L'attaccante ha identificato un problema nella gestione degli URL delle immagini all'interno della **photocall.** Il carattere # (hash) viene utilizzato come separatore nell'URL dell'immagine, ma non funziona correttamente. Per aggirare il problema, ha utilizzato un **URLencode** per sostituire # con la sua versione codificata %23, permettendo così il caricamento dell'immagine mancante.

Prova pratica:
ho fatto apparire l'immagine mancante nella **photohall** facendo ispeziona mi sono accorto che il # era utilizzato come separatore nel url dell'immagine, ma non funzionava quindi ho usato un url encoder per decifrarlo e sostituire gli # con il risultato cioe %23

Questa vulnerabilità suggerisce che:

- Il sistema **non valida correttamente gli input negli URL**, permettendo all'attaccante di modificarli manualmente per caricare risorse non previste.

- Potrebbe esserci **un potenziale rischio di directory traversal o information disclosure**, se la manipolazione dell'URL consente l'accesso a file non destinati all'utente.

- Se la gestione dell'URL non è robusta, potrebbe essere possibile caricare immagini da **server esterni**, con il rischio di attacchi **SSRF (Server-Side Request Forgery)** o esposizione a contenuti malevoli.

Possibili impatti

- **Accesso a risorse non autorizzate**: Un utente potrebbe manipolare l'URL per accedere a immagini o file riservati.

- **Attacchi SSRF**: Se il sistema consente di caricare immagini da URL arbitrari, potrebbe essere sfruttato per attaccare servizi interni.

- **Phishing o attacchi XSS**: Se gli utenti possono caricare immagini arbitrarie, potrebbero essere indotti a caricare immagini da server malevoli per attacchi di phishing o esecuzione di codice dannoso.

Valutazione del rischio: MEDIO-ALTO

- **Impatto**: Medio (può portare a disclosure di informazioni, SSRF o attacchi basati su contenuti esterni).

- **Probabilità**: Alta (se il sistema non valida gli URL o consente di modificarli facilmente).

Mitigazioni e contromisure

Per proteggere il servizio da questa vulnerabilità, il provider dovrebbe implementare le seguenti misure:

1. **Validazione degli URL lato server**

- Consentire solo URL che puntano a risorse autorizzate (es. immagini caricate dall'utente o da un CDN sicuro).

- Rifiutare URL con caratteri speciali (`#`, `23`, `..`, `//`) che potrebbero essere usati per attacchi di **directory traversal**.

2. **Whitelist di URL consentiti**

- Se l'applicazione consente il caricamento di immagini da URL esterni, dovrebbe essere implementata una whitelist di domini affidabili.

3. **Sanitizzazione e encoding degli input**

- Assicurarsi che gli input dell'utente vengano **sanitizzati e normalizzati** prima di essere utilizzati negli URL.

- Evitare di interpretare caratteri speciali (`%23`, `..`, `?`, `&`) in modo pericoloso.

4. **Protezione contro SSRF**

- Se il sistema accetta URL esterni, implementare restrizioni per impedire richieste verso **server interni** o servizi cloud privati.

- Limitare il tipo di richieste che il server può eseguire.

5. **Logging e monitoraggio**

- Registrare tentativi sospetti di modifica degli URL.

- Implementare alert per richieste insolite o URL manipolati.

Fonte:LOGIN ADMIN

1. SQL Injection (SQLi)

Descrizione della vulnerabilità

L'attaccante ha testato la vulnerabilità all'**SQL Injection** inserendo il payload:

SQL
(Copia Modifica)

```
' OR true--
```

nel campo email del modulo di login. Questo suggerisce che il sistema **non filtra adeguatamente gli input utente prima di eseguire le query SQL**.

La query risultante potrebbe essere simile a:

SQL
(Copia Modifica)
```
SELECT * FROM users WHERE email = '' OR true--' AND password = 'xyz'
```

L'operatore `OR true` rende la condizione sempre vera, permettendo all'attaccante di bypassare l'autenticazione e accedere senza conoscere la password.

L'attaccante ha inoltre scoperto che il sistema seleziona l'utente con **l'ID più basso** in caso di più corrispondenze, deducendo che il primo account creato è quello **admin**.

Prova pratica:
L'applicativo è vulnerabile a injection di codice di conseguenza provo a iniettare del codice sql, vado in login e nella mail e scrivo ' OR true–. Inserisco caratteri casuali nel campo password e provo a loggare, a quanto pare la logica del sistema quando vengono presi più utenti prende quello con il ID più basso ed a sto punto presumo che l'account admin admin@juice-sh.op è il primo account mai creato

Possibili impatti

- **Accesso non autorizzato**: Un attaccante può autenticarsi come qualsiasi utente, incluso l'**amministratore**.
- **Compromissione del database**: Se sfruttata oltre il login, l'SQL Injection potrebbe permettere **dump di dati sensibili**, cancellazione di tabelle o **escalation di privilegi**.
- **Possibile esecuzione di codice remoto (RCE)**: Se il database consente **query per l'esecuzione di comandi di sistema**, l'attaccante potrebbe ottenere il pieno controllo del server.

Valutazione del rischio: CRITICO

- **Impatto**: **Estremamente alto** (compromissione totale del sistema).
- **Probabilità**: **Alta** (se il sistema è vulnerabile e non utilizza query parametrizzate).

Mitigazioni e contromisure

Per arginare questa vulnerabilità, il provider del servizio deve implementare le seguenti misure:

1. **Utilizzare query parametrizzate (Prepared Statements)**

- Le query SQL devono essere scritte in modo sicuro, utilizzando parametri al posto di concatenazioni di stringhe.

Esempio corretto con Prepared Statements in Python (usando SQLite):

PYTHON
(Copia Modifica)

```
cursor.execute("SELECT * FROM users WHERE email = ? AND password = ?", (email, password))
```

- Questo impedisce l'esecuzione di input malevoli come SQL Injection.

2. **Sanitizzazione e validazione degli input**

- Verificare e filtrare tutti gli input dell'utente, **non consentendo caratteri speciali** o parole chiave SQL (`'`, `--`, `OR`, `AND`, etc.).

- Utilizzare **whitelist** per accettare solo formati validi (es. email con regex).

3. **Limitare i privilegi del database**

- Gli utenti del database dovrebbero avere **permessi minimi necessari**.

- Separare gli utenti del database tra **lettura**, **scrittura**, e **amministrazione** per ridurre il rischio di compromissione totale.

4. **Implementare l'Autenticazione Sicura**

- **Hashing delle password**: Utilizzare algoritmi sicuri come **bcrypt**, **Argon2** o **PBKDF2**.

- **Rate Limiting**: Limitare il numero di tentativi di login per prevenire attacchi automatizzati.

5. **Logging e Monitoraggio**

- Registrare i tentativi di login sospetti e gli errori SQL.

- Implementare un **sistema di allerta** per query anomale.

6. **Utilizzare Web Application Firewall (WAF)**

- Un WAF può rilevare e bloccare **pattern noti di SQL Injection** prima che raggiungano l'applicazione.

Fonte:ADMIN SECTION

1. Accesso non autorizzato al pannello amministratore (Insecure Direct Object Reference - IDOR)

Descrizione della vulnerabilità

L'attaccante ha analizzato il file **main.js**, che contiene la configurazione delle rotte dell'applicazione, identificando la route **"administration"**. Successivamente, ha inserito manualmente `/administration` nell'URL ed è riuscito ad accedere al pannello amministratore, **senza autenticazione o autorizzazione adeguata**.

Questo suggerisce che:

- L'accesso al pannello amministratore **non è protetto adeguatamente lato server**.
- Il controllo di accesso è affidato unicamente al client-side (`canActivate` nel file JavaScript), che può essere facilmente aggirato.

Prova pratica:

Per accedere al pannello amministratore visiono il file main.js poichè all'interno ci sono listate tutte le pagine e cercando administration trovo questo:

{ path: 'administration',
component: Ki,
canActivate: [te] },
ora inserisco administration nel url e sono nel pannello amministratore

Possibili impatti

- **Accesso non autorizzato alle funzioni di amministrazione**: Un utente malintenzionato potrebbe modificare configurazioni, gestire utenti o accedere a dati

sensibili.

- **Esecuzione di azioni privilegiate**: Se il pannello consente operazioni critiche (es. eliminazione account, gestione pagamenti), l'attaccante potrebbe compromettere l'intero sistema.

- **Data leakage**: Possibile accesso a informazioni riservate, log di sistema, credenziali o altre risorse sensibili.

Valutazione del rischio: CRITICO

- **Impatto**: **Molto alto** (compromissione totale del sistema se il pannello di amministrazione non è protetto).

- **Probabilità**: **Alta** (se l'accesso si basa solo su controlli lato client, è facilmente aggirabile).

Mitigazioni e contromisure

Per proteggere il pannello di amministrazione, il provider del servizio deve implementare le seguenti misure:

1. **Implementare controlli di accesso lato server**

- Ogni richiesta a `/administration` deve essere verificata lato server, assicurandosi che l'utente abbia i permessi adeguati.

- Se l'utente non è autorizzato, il server deve restituire un **errore 403 (Forbidden)**.

Esempio in Express.js (Node.js):

JAVASCRIPT
(Copia Modifica)

```
app.get('/administration', (req, res) => {
  if (!req.user || req.user.role !== 'admin') {
    return res.status(403).send('Accesso negato');
  }
  res.sendFile('/admin-panel.html');
});
```

2. **Rimuovere informazioni sensibili dai file client-side**

- Evitare di listare direttamente le rotte sensibili nei file JavaScript accessibili al client.

- Se necessario, offuscare il codice o **caricare dinamicamente** le rotte dal server solo agli utenti autorizzati.

3. **Autenticazione e autorizzazione robusta**

- Implementare **token JWT** o **sessioni con ruoli e permessi** per controllare l'accesso.

- Usare **OAuth 2.0** o **RBAC (Role-Based Access Control)** per gestire i privilegi utente.

4. **Protezione tramite middleware**

- In framework come Angular o React, utilizzare **guardie di autenticazione** lato client, ma sempre abbinate a controlli lato server.

Esempio di protezione con middleware in Express.js:

javascript
(CopiaModifica)

```
function isAdmin(req, res, next) {
  if (!req.user || req.user.role !== 'admin') {
    return res.status(403).json({ message: 'Accesso negato' });
  }
  next();
}
app.use('/administration', isAdmin);
```

5. **Logging e monitoraggio**

- Registrare **tutti i tentativi di accesso** al pannello amministrativo.

- Implementare **sistemi di alert** per segnalare tentativi di accesso sospetti.

6. **Rate limiting e protezione anti-brute force**

- Limitare il numero di richieste per prevenire **tentativi ripetuti di accesso non autorizzato**.

FONTE:Email Leak

Vulnerabilità Identificate

1. Esposizione di Informazioni Sensibili tramite Endpoint non Protetto (`/rest/user/whoami`)

- **Descrizione**: L'endpoint `/rest/user/whoami` restituisce informazioni sull'utente attualmente autenticato, incluso l'email e l'immagine del profilo, senza ulteriori meccanismi di protezione o limitazione (es. CORS, SameSite, autenticazione lato server).

- **Dati esposti**:

- `id`

- `email`

- `lastLoginIp`

- `profileImage`

- **Rischio Associato**: **Medio-Alto**

    - Sebbene l'endpoint restituisca solo informazioni dell'utente autenticato, in contesti dove l'applicazione è vulnerabile ad attacchi come **Cross-Site Request Forgery (CSRF)** o **Cross-Site Script Inclusion (XSSI)**, un attaccante può sfruttare questa funzionalità per **esfiltrare dati sensibili** di utenti autenticati.

---

2. Supporto non Intenzionale a JSONP (`/rest/user/whoami?callback=test`)

- **Descrizione**: Il supporto alla tecnica JSONP (JSON with Padding) permette il caricamento cross-domain di dati JSON tramite tag `<script>`. In questo caso, il server restituisce una risposta eseguibile contenente i dati utente, rendendola vulnerabile ad attacchi **di tipo XSSI (Cross-Site Script Inclusion)**.

- **Rischio Associato**: **Alto**

- Se un utente autenticato visita un sito malevolo mentre è connesso al servizio, lo script su quel sito può **ottenere le sue informazioni personali**, sfruttando la risposta JSONP generata dal server target.

- L’attacco non richiede vulnerabilità XSS all’interno del dominio dell’applicativo vittima, rendendolo particolarmente pericoloso.

Come Mitigare queste Vulnerabilità

1. Disabilitare JSONP

- JSONP è una tecnologia obsoleta, superseded da **CORS (Cross-Origin Resource Sharing)**. Il server **non dovrebbe accettare parametri `callback`** né restituire dati come codice JavaScript.

- **Soluzione**: Rimuovere il supporto a `?callback=...` o configurare il framework/backend per rifiutare risposte JSONP.

In Express.js, ad esempio:

JS
(Copia Modifica)
```
app.set('jsonp callback name', null);
```

2. Applicare Header di Sicurezza e Protezione Cross-Origin

- Impostare correttamente i seguenti header HTTP:

  - `Content-Type: application/json`

  - `X-Content-Type-Options: nosniff`

  - `X-Frame-Options: DENY`

  - `Content-Security-Policy: script-src 'self'`

  - `Access-Control-Allow-Origin: [dominio specifico]`

3. Verificare l’Autenticazione e Autorizzazione Lato Server

- Sebbene l’endpoint sembri restituire solo i dati dell’utente loggato, è importante assicurarsi che non siano accessibili da terzi tramite sessioni condivise o intercettazioni.

- **Sessioni sicure**, **token CSRF**, **flag HttpOnly e Secure sui cookie**, e **header SameSite** sono essenziali.

4. Logging e Monitoraggio

- Tenere traccia degli accessi all'endpoint `/rest/user/whoami` per rilevare pattern anomali di accesso, come richieste da domini esterni o picchi improvvisi.

Prova pratica:
Utilizzando **burp** per analizzare il traffico di rete durante un normale utilizzo dell'applicativo e dopo aver "mappato" le cartelle, ho notato che durante il login l'applicativo interpellava la cartella **/rest/***, vedendo che all'interno di essa c'è la cartella user in cui ho visto solo whoami ho deciso di darci un'occhiata. Ho inserito nel url **/rest/user/whoami** e mi restituisce le informazioni dell'account con il quale sono loggato.

ESEMPIO: se loggassi come admin vedrei:
{"user":{"id":1,"email":"admin@juice-sh.op","lastLoginIp":"undefined","profileImage":"assets/public/images/uploads/defaultAdmin.png"}}, a questo punto sono sicuro di aver trovato il mio endpoint dove posso estrapolare informazioni senza venir bloccato, dopo di che posso vedere se l'applicativo è vulnerabile ad uno script **jsonp** quindi eseguo:

/rest/user/whoami?callback=test e mi restituisce // typeof test === 'function' && test({"user":{"id":1,"email":"admin@juice-sh.op","lastLoginIp":"undefined","profileImage":"assets/public/images/uploads/defaultAdmin.png"}})**

dopo aver analizzato il tutto confermo che è vulnerabile ad un attacco jsonp.

SPIEGAZIONE COMANDO:Quando si fa una richiesta a */rest/user/whoami?callback=test*, il parametro callback indica al server che la risposta deve essere incapsulata in una funzione di callback JavaScript. Questo è ciò che rende la risposta una JSONP. Il server avvolgerà i dati JSON in una funzione che verrà eseguita dal browser.

Conclusioni

Il sistema esaminato presenta una vasta gamma di vulnerabilità, dalle più comuni (XSS, SQLi) a quelle legate alla cattiva configurazione (IDOR, JSONP, FTP). È urgente implementare misure di sicurezza sia tecniche (CSP, query parametrizzate, autenticazione lato server) che organizzative (logging, monitoraggio, policy di accesso) per mitigare i rischi identificati.

Raccomandazioni:

- Eseguire code review e penetration test periodici.
- Adottare framework sicuri e best practice OWASP.
- Formare gli sviluppatori su temi di sicurezza.

Con l'adozione delle contromisure proposte, è possibile ridurre sensibilmente il rischio di attacchi e migliorare la sicurezza dell'intera infrastruttura.