

Progetto Network Security

Arnone Riccardo M63001225

Flagiello Mattia M63001540

Ilardi Lucio M63001505

Sommario

Introduzione	1
CSRF	2
SQL Injection	4
XSS	6
Reflected	6
Stored	8
DOM Based	10

Introduzione

Il nostro progetto consiste di una Web Application sviluppata in Spring, progettata in maniera vulnerabile al fine di mostrare attacchi di tipo XSS (reflected, stored e DOM based), CSRF e SQL Injection. Oltre all'app vulnerabile è stata creata una seconda versione “safe” che include le contromisure necessarie per prevenire tali problemi.

Avviare l'applicazione con `$ docker-compose up -d`: verranno creati 3 container su porto 8080 (app vulnerabile), 8081 (app safe), 8082 (sito malevolo).

```
1 version: "3.8"
2
3 services:
4
5   WebAppVulnerable:
6     image: ilardi/webappvulnerable
7     ports:
8       - "8080:8080"
9     networks:
10      - ns_network
11
12   WebAppSafe:
13     image: ilardi/webappsafe
14     ports:
15       - "8081:8081"
16     networks:
17       - ns_network
18
19   evil:
20     image: ilardi/evil
21     ports:
22       - "8082:8082"
23     networks:
24       - ns_network
25
26 networks:
27   ns_network:
```

Per accedere all'app vulnerabile visitare localhost:8080/ ed effettuare il login (sito safe stessa cosa ma su porto 8081).

Utenti del sistema (username, password):

- alice, alicepass
- bob, bobpass
- carl, carlpass

CSRF

Un attacco CSRF (Cross Site Request Forgery) cerca di sfruttare una sessione esistente fra client e server, tipicamente tramite cookie di sessione, per creare delle richieste che non sono riconoscibili dalle legittime richieste effettuate dal client. Questo consente quindi di effettuare una qualsiasi azione presso il server come se fossimo loggati con quel particolare account (Es. cambiare password).

Nella nostra applicazione la sessione viene mantenuta tramite cookie JSESSIONID. Per provare l'attacco CSRF è necessario per prima cosa effettuare il login al sistema, dunque visitiamo localhost:8080/ ed accediamo ad un account (username: alice, password: alicepass). La funzionalità che analizziamo è quella di trasferimento del denaro, ossia un form che consente di trasferire parte del proprio bilancio ad altri utenti del sistema:



Trasferimento fondi

Username destinatario:

Importo:

Trasferisci denaro

Attacco CSRF

Torna alla home

Non essendoci alcuna protezione contro CSRF è possibile far partire tale richiesta da un sito esterno senza l'effettiva volontà dell'utente; difatti cliccando sul link fornito nella pagina si arriva sul sito malevolo, nel quale viene automaticamente inviata una richiesta POST verso localhost:8080/transfer, trasferendo i fondi verso l'account di bob.

Tornando in home possiamo controllare il bilancio di alice e bob e notare come l'attacco sia andato a buon fine: (bilancio iniziale 1000 e 2000)

Ciao alice ! Ciao bob !

Bilancio: 600

Bilancio: 2400

La prima contromisura contro attacchi CSRF è l'utilizzo di SameSite cookies:

- SameSite=strict: il browser manda il cookie solo se la richiesta proviene dallo stesso dominio della destinazione, dunque non viene inviato per richieste cross-site.
- SameSite=lax: il cookie viene inviato anche in richieste cross-site ma solo in caso di GET iniziate dall'utente, quindi ad esempio se si clicca su un link, mentre viene bloccato se la richiesta parte in automatico (Es. GET tramite il caricamento di un'immagine) o se si tratta di una POST.

```
server.servlet.session.cookie.same-site=strict
```

Oltre alle policy SameSite è opportuno l'utilizzo di token per prevenire attacchi CSRF. Un token viene randomicamente generato dal backend e settato come cookie relativo al documento restituito all'utente; il token viene quindi recuperato e incluso nelle richieste effettuate verso il server. Questo meccanismo previene attacchi CSRF perché l'attaccante non ha accesso al token.

Nel codice che segue il token viene recuperato tramite document.cookie e inviato nella richiesta POST verso il backend:

```
const csrfToken = document.cookie.replace(/(?:?:(?:^|.*;)\s*)XSRF-TOKEN\s*=\s*([^\s]*)\.?$/|^.*$/, '$1');

document.getElementById("transferForm").addEventListener("submit", function(event) {
    event.preventDefault();

    var destinationUsername = document.getElementById("destinationUsername").value;
    var amount = document.getElementById("amount").value;

    var xhr = new XMLHttpRequest();
    xhr.open("POST", "/transfer", true);
    xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    xhr.setRequestHeader('X-XSRF-TOKEN', csrfToken);
```

Con queste contromisure, riprovando l'attacco sul sito safe otteniamo un 403 dato che l'attaccante tenta di far partire la richiesta ma non ha a disposizione il token:

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun May 05 16:19:14 CEST 2024

There was an unexpected error (type=Forbidden, status=403).

SQL Injection

Un attacco di tipo SQL Injection sfrutta errati meccanismi di creazione delle query per effettuare operazioni su un database che non dovrebbero essere consentite ad un normale utente. Questo accade solitamente quando l'applicazione crea delle query a partire da dati inseriti dall'utente che non vengono sanitizzati nel modo corretto.

Nella nostra applicazione la funzionalità vulnerabile a SQL Injection è quella di modifica dell'account; la query infatti viene creata attraverso una semplice concatenazione di stringhe senza alcun controllo sui dati inseriti:

```
// Fix per SQL injection: query parametrica e
// input validation nel controller
@Override
public void updateAccount(String email, String address, String description,
    String username) {
    String queryString = "UPDATE Account SET email='" + email + "', "
        + "description='" + description + "', "
        + "address='" + address + "' "
        + "WHERE username='" + username + "'";
    em.createNativeQuery(queryString).executeUpdate();
}
```

Nella pagina sono suggeriti alcuni payload di attacco, ad esempio per modificare il proprio bilancio o quello di altri utenti. Attraverso il "--" viene commentato il WHERE della query e dunque si possono inserire le condizioni che si vuole:

Modifica il tuo account

Email:

Indirizzo:

Descrizione:

Conferma

Account modificato con successo

Esempi di payload per SQL Injection:

', balance=9999 where username='alice' --

', balance=1 where username='carl' --

Torna alla home

Tornando nella home possiamo vedere come l'attacco abbia avuto successo:

Ciao alice !

Bilancio: 9999

Email:

La contromisura a questo tipo di attacco è gestire opportunamente i dati inseriti dall'utente; invece che concatenare gli input si può usare una query parametrica:

```
// Soluzione a SQL Injection: uso di query parametrica JQPL
// Inoltre si può fare input validation nel controller
// (non implementato per mostrare l'effetto della query parametrica)
@Override
public void updateAccount(String email, String address, String description, String username) {
    String queryString = "UPDATE Account SET email = :email, description = :description, "
        + "address = :address WHERE username = :username";
    Query query = em.createQuery(queryString);
    query.setParameter("email", email);
    query.setParameter("description", description);
    query.setParameter("address", address);
    query.setParameter("username", username);
    query.executeUpdate();
}
```

Con questa modifica l'utente non può modificare la struttura della query, per cui riprovando l'attacco (sul sito safe) il risultato è il seguente:

Ciao alice !

Bilancio: 1000

Email: ', balance=9999 where username='alice' --

Inoltre sarebbe opportuno fare input validation in modo che l'utente non possa inserire caratteri speciali, ma nel progetto non è stata implementata in modo da mostrare l'effetto della query parametrica.

XSS

Gli attacchi di tipo XSS (Cross Site Scripting) sfruttano vulnerabilità di siti web per cercare di eseguire codice malevolo lato client: questo può avere conseguenze molto varie come session hijacking tramite furto di cookies, redirect verso siti malevoli e altro ancora. Nella nostra applicazione useremo come script di attacco un semplice alert() per mostrare che è stato eseguito codice Javascript.

La prima contromisura contro Cross Site Scripting è gestire opportunamente i valori inseriti in input dall'utente ed evitare ad esempio che appaiano tag <script> all'interno del documento; nelle sezioni che seguono sono discusse soluzioni specifiche a seconda del contesto.

Esistono anche rimedi più generali. Il primo consiste nell'abilitare la protezione XSS built-in del browser settando l'header X-XSS-Protection. Il secondo riguarda la configurazione della Content Security Policy che consente il caricamento e l'esecuzione solo delle risorse specificate; con la direttiva "script-src 'self'" stiamo indicando che possono essere eseguiti solo gli script aventi la stessa origine della pagina web.

```
//XSS protection
http.headers(headers -> headers
    .xssProtection(xss -> xss
        .headerValue(XXssProtectionHeaderWriter.HeaderValue.ENABLED_MODE_BLOCK))
    .contentSecurityPolicy(cps -> cps.
        policyDirectives("script-src 'self';")))
);
```

Reflected

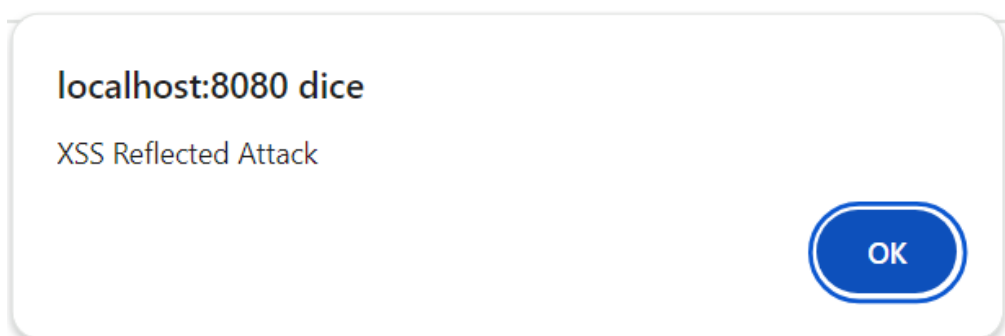
Un attacco XSS di tipo Reflected si basa sul fatto che parte dell'input inserito dall'utente viene riflesso nella risposta. Se i dati inseriti non vengono gestiti nel modo corretto è possibile iniettare nella pagina di risposta dei tag <script> e dunque far eseguire codice lato client.

Nella nostra applicazione la funzionalità vulnerabile è quella della recensione: l'utente ha modo di scrivere una recensione, e dopo averla inviata questa viene visualizzata a schermo.



The screenshot shows a web interface for leaving a review. It has a title "Lascia una recensione:", a text input field, a green "Invia recensione" button, and a section titled "La tua recensione:" with the text "Questa è la mia recensione". Below the form, there is an orange banner that says "Attacco Reflected XSS" with an information icon, and a blue button that says "Torna alla home".

Dato che il testo inserito non viene controllato in alcun modo, inserendo un payload contenente tag `<script>` è possibile effettuare un attacco Reflected XSS (visitare il link sulla pagina per far partire l'attacco da sito malevolo).



The screenshot shows a modal dialog box with the text "localhost:8080 dice" and "XSS Reflected Attack". There is a blue "OK" button in the bottom right corner.

Il motivo della vulnerabilità è l'utilizzo della funzione `th:utext` di Thymeleaf per mostrare a schermo la recensione inserita dall'utente; questa è una funzione vulnerabile dato che inserisce il testo unescaped:

```
<!-- Soluzione: usare th:text -->
<p th:utext="${review}"></p>
```

Per risolvere il problema è dunque necessario fare l'escaping della recensione in modo che i tag `<script>` non siano interpretati come Javascript; questo è stato realizzato mediante l'utilizzo di `th:text`:

```
<!-- th:text al posto di th:utext -->
<p th:text="${review}"></p>
```

Proviamo quindi l'attacco dal sito safe (Nota: essendo abilitata la protezione contro CSRF il link malevolo non funziona, quindi lanciare l'attacco inserendo come recensione il payload `<script>alert("Reflected XSS");</script>`):

La tua recensione:

`<script>alert("Reflected XSS");</script>`

La stringa viene solo visualizzata a schermo ma non eseguita come script.

Stored

Un attacco Stored XSS (o persistent) salva sul server lo script malevolo; questo poi viene visualizzato da altri utenti e dunque eseguito dai loro browser.

Un esempio classico, implementato nella nostra applicazione, è quello di una bacheca di commenti; è possibile scrivere e salvare sul server dei commenti e, se questi non vengono gestiti in maniera appropriata, tutti gli utenti che visualizzano la bacheca saranno soggetti allo script malevolo inserito.

Bacheca dei commenti

- alice: Ciao a tutti!
- bob: Mandatemi dei soldi :(
- carl: Bel tempo oggi.

Ordinato per:

Ordinamento: [Ordina](#)

Aggiungi un commento

[Posta il commento](#)

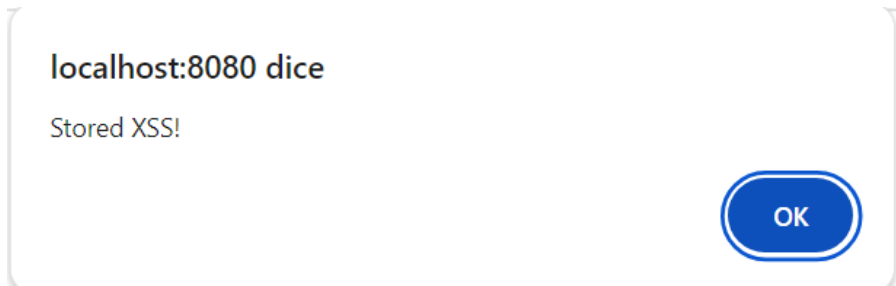
Esempio di payload per Stored XSS:
`<script>alert("Stored XSS!");</script>`
(refreshare la pagina dopo averlo inserito)

[Attacco DOM XSS](#) ⓘ

[Torna alla home](#)

Inseriamo come commento il payload suggerito nella pagina: al refresh lo script verrà eseguito, sia da noi che da qualsiasi altro utente che visita la bacheca (in uno scenario

reale lo script può essere più complesso in modo che non abbia effetto sull'attaccante stesso).



La vulnerabilità deriva dall'uso di funzioni non safe per mostrare a video i commenti, in particolare l'uso di `innerHTML` per creare gli elementi della lista e di `th:utext` per mostrarli:

```
window.commentsData.forEach(comment => {
    const listItem = document.createElement('li');

    // Using 'textContent' or 'innerText' instead of 'innerHTML' will
    // automatically escape any HTML tags.
    listItem.innerHTML = comment.username + ': ' + comment.comment;
    commentsList.appendChild(listItem);
});

<div id="comments">
  <ul id="comments-list">
    <!-- Render initial comments using Thymeleaf -->
    <!-- Usare th:text al posto di th:utext per risolvere Stored XSS -->
    <li th:each="comment : ${comments}" th:utext="${comment.username + ': ' + comment.comment}"></li>
  </ul>
</div>
```

Il problema è stato risolto mediante l'impiego di funzioni più sicure (`textContent` e `th:text`):

```
window.commentsData.forEach(comment => {
    const listItem = document.createElement('li');

    // textContent usato al posto di innerHTML
    listItem.textContent = comment.username + ': ' + comment.comment;
    commentsList.appendChild(listItem);
});

<div id="comments">
  <ul id="comments-list">
    <!-- Render initial comments using Thymeleaf -->
    <!-- th:text al posto di th:utext -->
    <li th:each="comment : ${comments}" th:text="${comment.username + ': ' + comment.comment}"></li>
  </ul>
</div>
```

Con tali modifiche questo è il risultato dell'attacco sull'app safe (la stringa viene visualizzata come testo ma non eseguita):

Bacheca dei commenti

- alice: Ciao a tutti!
- bob: Mandatemi dei soldi :(
- carl: Bel tempo oggi.
- alice: `<script>alert("Stored XSS!");</script>`

DOM Based

Il DOM Based XSS è chiamato in questo modo perché sfrutta il Document Object Model, ossia una rappresentazione strutturata del documento HTML. In particolare, negli attacchi XSS DOM Based i dati malevoli non sono elaborati dal server ma vengono riflessi dal codice JavaScript interamente sul lato client, dunque sia source che sink sono contenuti nel browser. Uno scenario tipico è quello in cui un query parameter contenuto nell'URL viene utilizzato in maniera non safe, ad esempio attraverso funzioni come `eval()` oppure `document.write()`, causando dunque l'esecuzione di un eventuale script.

Nella nostra applicazione questo tipo di vulnerabilità è presente sempre nella bacheca dei commenti; difatti il query parameter "sortField" viene utilizzato per aggiungere nuove opzioni di ordinamento, e il parametro stesso viene visualizzato a schermo nel menu a tendina tramite un `document.write()`:

```
if (!existingOption) {  
    // If the sort field doesn't exist in the dropdown options, add it dynamically  
    document.write('<option value="' + sortField + '">' + sortField + '</option>');  
}
```

In assenza di controlli basta inserire lo script come valore del parametro sortField per effettuare l'attacco (visitare il link sulla pagina per far partire l'attacco):

localhost:8080/board?sortField= `<script>alert(%27DOM%20XSS%20Attack!%27);</script>`

localhost:8080 dice

DOM XSS Attack!

OK

La funzione `document.write()` rappresenta un sink vulnerabile perché introduce i tag `<script>` all'interno del documento. La soluzione consiste nell'utilizzare funzioni più sicure che trattino il query parameter come testo; nell'app safe è stato rimosso il

document.write() ed utilizzato il metodo add() (dell'elemento <select>) per aggiungere la nuova opzione di ordinamento:

```
if (!existingOption) {  
  // If the sort field doesn't exist in the dropdown options, add it dynamically  
  var option = document.createElement("option");  
  option.text = sortField;  
  option.value = sortField;  
  dropdown.add(option);  
}
```

Facendo così il query parameter non viene eseguito ma è considerato come una semplice stringa:

localhost:8081/board?sortField= <script>alert(%27DOM%20XSS%20Attack!%27);</script>



The screenshot shows a web application interface. At the top, a browser address bar displays the URL: localhost:8081/board?sortField= <script>alert(%27DOM%20XSS%20Attack!%27);</script>. The main content area has a light gray background. On the right side, there is a white rounded rectangle titled "Bacheca dei commenti" in blue. Below the title, there is a list of comments: "alice: Ciao a tutti!", "bob: Mandatemi dei soldi :(", and "carl: Bel tempo oggi.". Below the comments, there are two sorting options. The first is "Ordinato per:" followed by a dropdown menu showing "<script>alert('DOM XSS Attack!');</script>". The second is "Ordinamento:" followed by a dropdown menu showing "Ascendente" and a blue button labeled "Ordina".

Come sempre sarebbe opportuno fare input validation ed evitare di accettare stringhe contenenti caratteri speciali.