

Linguaggi di Programmazione per il Web – Parte 9

PHP – Hypertext Preprocessor

**SQL injection:
cosa sono e come proteggersi**

Autore

Prof. Rio Chierago
riochierago@libero.it

Siti Utili

<http://www.riochierego.it/mobile>

<http://www.html.it/>

<http://www.mrwebmaster.it>

<http://www.php.net/>

<http://php.html.it/>

SQL Injection: **premessa**

La creazione di **contenuti dinamici** che consentano all'utente di interagire con il nostro sito sono spesso alla base del successo o dell'insuccesso dei servizi che offriamo tramite il web.

Maggiore è l'**importanza dei dati che gestiamo**, maggiore sarà anche il **bisogno di sicurezza** che ruota attorno ai dati stessi.

La **sicurezza** di un sito web non viene garantita soltanto da un web server ben configurato, o da un tunnel SSL, ma deve essere implementata in maniera coscienziosa anche da chi sviluppa l'applicazione web, nel nostro caso il **programmatore PHP**.

In queste slide andremo a conoscere una delle più classiche tipologie di attacco legate al web, molto diffusa ma spesso sottovalutata, che va a colpire il **cuore** dell'applicazione web, **ossia il database**: si tratta **dell'attacco di tipo SQL Injection**.

È importante tenere presente che questo fenomeno può interessare qualsiasi linguaggio di programmazione e qualsiasi DBMS, anche se gli esempi proposti faranno riferimento al linguaggio PHP, con accenni al db server MySQL

by Prof. Rio Chierogo

SQL Injection: **cosa sono**

- La **SQL Injection** è una tecnica molto diffusa di **hacking**, la quale mira a colpire quei siti web **che si appoggiano su un DBMS di tipo SQL**.
- L'attacco si basa principalmente sugli input dell'utente, il quale potrebbe inserire del codice maligno all'interno di una query. Gli effetti sono imprevedibili, ad esempio il cracker potrebbe autenticarsi con privilegi uguali ai vostri se non addirittura superiori e **alterare di conseguenza i dati sensibili all'interno del DB**.
- Il problema è relativamente semplice da capire, ma è anche molto pericoloso: effettuando una query SQL costruita sulla base di input passati da un utente, **senza eseguire un controllo preventivo** sullo stesso input, tale query può essere manipolata a piacimento
- L'input dell'utente nel nostro caso può esserci trasmesso in vari modi: **tramite URL** (query string), **tramite un form HTML** oppure anche **tramite un cookie** costruito su misura.
- Non tutti gli utenti utilizzano il nostro sito in modo "ortodosso", ad esempio cliccando su un link o compilando correttamente un modulo, per cui i dati che ci arrivano potrebbero non rispettare le nostre aspettative.

SQL Injection: **problematiche** **connesse**

Che tipo di problemi può portare una query manipolata arbitrariamente da un utente?

- manipolazione indesiderata dei nostri dati
- accesso indesiderato ad aree riservate
- visualizzazione di dati privati

SQL Injection: **esempio 1**

Supponiamo ad esempio di avere una variabile **\$id** presa in input dalla **query**, teoricamente di tipo intero ma **non correttamente validata e controllata**.

Una query del tipo

```
$sql = "SELECT * FROM Articolo WHERE id=$id;";
```

potrebbe essere manipolata a piacere da un utente smaliziato, causando problemi di varia natura.

Che cosa succede se **\$id** anzichè essere un numero intero viene passata come stringa, ad esempio **"1; DROP TABLE Articolo;"**?

La query verrebbe trasformata in questo modo:

```
$sql = "SELECT * FROM Articolo WHERE id=1; DROP TABLE Articolo;";
```

La prima query magari non ha effetto ma la seconda.....

N.B. Comunque MySQL non supporta gli statement multipli, per cui una query come questa produrrebbe soltanto un messaggio d'errore.

SQL Injection: **esempio 2**

Supponiamo, ora, di avere un **form** per l'autenticazione di un utente per effettuare il **login** in un'area protetta dell'applicazione web.

Probabilmente il **form di login** avrà un campo per inserire la **username** e la **password**.

Una volta inseriti i dati, l'utente clicca su un pulsante e i dati vengono inviati all'applicazione la quale usando questi dati costruirà una query per interrogare il database al fine di controllare se le credenziali inserite sono corrette o meno.

SQL Injection: **esempio 2**

Probabilmente il **form di login** sarà costruito in questa maniera (file **login.htm**)

```
<form action='login.php' method='POST' target='_self' />
```

```
<label>Inserisci Userid:</label>
```

```
<input type='text' id='userid' name='userid' />
```

```
<label>Inserisci Password:</label>
```

```
<input type='password' id='pwd' name='pwd' />
```

```
<input type='submit' value='OK' />
```

```
</form>
```


SQL Injection: **esempio 2**

Premendo OK arriveranno alla pagina **login.php** le seguenti variabili in POST

- **userid**, con la userid inserita dall'utente
- **pswd**, con la password inserita dall'utente

Queste variabili saranno usate per costruire una query string in PHP del tipo

```
$sql = "SELECT *  
      FROM Utente  
      WHERE userid= ' " . $_POST['userid' ] . " '  
      AND password = ' " . $_POST['pswd' ] . " ' ;"
```

Se l'utente ha quindi inserito userid = "pippo" e pswd = "paperino" la query sarà

```
$sql = "SELECT *  
      FROM Utente  
      WHERE userid= 'pippo' AND password = 'paperino';"
```

La query restituirà i dati degli utenti con username **pippo** e password **paperino** presenti nel nostro database (magari nessuna). **Fin qui nulla di strano.**

SQL Injection: **esempio 2**

Supponiamo, ora, che l'utente inserisca

- userid = "pippo"
- pswd = ' (un apice singolo)

La query eseguita ora sarà

```
SELECT *  
FROM Utente  
WHERE userid= 'pippo' and password= ' ' ;
```

Non notate ancora nulla di strano?

Fate attenzione al campo password e alla presenza di **ben tre apici**.....

Ebbene supponiamo che l'utente ora inserisca

- userid = "pippo"
- pswd = ' OR '1' = '1 (0 OR 1 = 1 se numerico)

Cosa accadrà?

SQL Injection: **esempio 2**

La query string ottenuta sarà:

```
SELECT *  
FROM Utente  
WHERE userid= 'pippo' and password= " OR '1' = '1' ;
```

L'esecuzione fornirà i dati di **TUTTI I RECORD** presenti nella tabella utenti.

Ecco...abbiamo appena effettuato un attacco di sql injection.

Questo accade perchè la seconda parte della query e cioè **OR '1'='1'** è sempre vera, ed è quella che abbiamo inserito noi, deliberatamente, nel campo di input.

Tutto ciò è stato reso possibile dal primo apice della stringa inserita che ha chiuso prematuramente la prima parte della query, permettendoci di inserire una clausola OR che è sempre vera e che mira a rendere vero tutto l'enunciato composto così ottenuto.

Lo stesso risultato si sarebbe ottenuto usando una qualsiasi altra espressione vera.

Lo stesso vettore di attacco si può provare sostituendo i singoli apici con i doppi apici.

SQL Injection: **esempio 2**

Supponiamo, ora, che l'utente inserisca (perché lo conosce) esclusivamente

- userid = `pippo' OR 1=1 --'`
- pswd = `nulla`

La query eseguita ora sarà

```
SELECT *  
FROM Utente  
WHERE userid= ' pippo' OR 1=1 --' AND password= ' ' ;
```

Ecco...abbiamo appena effettuato un altro attacco di sql injection

N.B. In questo modo posso ottenere la password dell'utente con userid pippo senza conoscerla sfruttando una debolezza del codice implementato che non ha tenuto in considerazione la possibilità di **sql injection**

Le vulnerabilità possono essere numerose (dipende dalla fantasia dell'attaccante) e derivano solitamente da una **distrazione del programmatore** o comunque da una **errata implementazione**.

by Prof. Rio Chierogo

SQL Injection: **pericolosità**

- Sfruttando quindi questo tipo di vulnerabilità, **detta **sql injection** in quanto realizzata iniettando codice sql malevolo**, potremmo loggarci senza conoscere le credenziali degli utenti.
- Una volta compromesso il database, i danni che si possono causare possono essere incalcolabili, dalla sottrazione di dati sensibili alla distruzione di record, di tabelle o dell'intero database con l'aggiunta di stringhe tipo

" **'; DROP TABLE <nome_della_tabella> –**"

Si possono anche eseguire dei programmi sul server usando stringhe del tipo

" **'; exec master..xp_cmdshell 'notepad.exe'–** "

In questo caso verrebbe eseguita l'applicazione notepad.exe, ma potrebbero anche essere caricati sul server dei programmi per sniffare il traffico e sottrarre informazioni, il server, quindi, sarebbe completamente compromesso.

SQL Injection: **altri esempi**

Ecco una breve lista di attacchi che possono far scoprire eventuali vulnerabilità di tipo **sql injection**

- '--
- ' OR ' 1'=' 1
- " OR " 1 " = " 1
- ' /*
- " /*
- '//
- "--
- ' OR 1=1--
- " OR 1=1--
- ') OR ('1'='1
- ') OR (1=1

N.B. Per maggiori informazioni sull'**Sql Injection** e sulla sicurezza delle applicazioni web in generale si consiglio di dare uno sguardo al sito dell'[OWASP](https://www.owasp.org/)

SQL Injection: **come proteggersi**

La **SQL injection** è una delle tecniche di hacking più pericolose perchè facilmente riproducibili e dagli effetti catastrofici. Per difendersi basta affidarsi ad un principio generale “Non fidarsi mai”.

L'errore commesso nell'implementazione presentata prima è proprio quello di fidarsi che i dati inseriti dagli utenti sono sicuri, senza effettuare alcun controllo.

Non esiste una soluzione assoluta, che sia sicura e portabile allo stesso tempo.

Lo stesso concetto di sicurezza è relativo e viene influenzato da numerosi parametri. Il nostro approccio al problema dovrà essere abbastanza elastico e dovremo adattarci alle situazioni, scegliendo di volta in volta la soluzione più opportuna.

SQL Injection: **come proteggersi**

È stato detto che il punto fondamentale è **saper gestire l'input dell'utente**, e sarà qui che andremo a focalizzare la nostra attenzione.

A seconda dei dati che andremo a trattare, possiamo adottare più strategie per elevare il livello di sicurezza effettuando:

- **Controlli sul tipo di dato**
- **Creazione di filtri tramite espressioni regolari**
- **Eliminazione di caratteri potenzialmente dannosi**
- **Escape di caratteri potenzialmente dannosi**

SQL Injection: controlli sul tipo di dato

Il type casting è un'operazione che forza una variabile ad essere valutata come appartenente ad un certo tipo.

Nell'**esempio 1** di query manipolata, ci aspettiamo che la variabile **\$id** sia di tipo intero, ma per evitare sorprese, è opportuno essere sicuri di ciò:

```
$id = (int) $_GET['id']; // Cast del tipo .... $_GET e $_POST sono array di valori stringa  
$sql = "SELECT * from Articolo WHERE id=$id"; \\ In questo modo $id avrà sicuramente un valore intero.
```

Oppure

```
settype($_GET['id'], 'int'); // forzo la variabile ad essere di tipo intero  
$id = $_GET['id'];
```

Oppure

```
$id = intval($_GET['id']); //solo per gli interi
```

SQL Injection: **controlli sul tipo di dato**

Un approccio leggermente diverso ma sempre legato al tipo di variabili in gioco consiste nel verificare, mediante opportune funzioni come

[is_int\(\)](#)

[is_numeric\(\)](#)

[gettype\(\)](#)

che una variabile appartenga ad un determinato tipo e comportarsi di conseguenza:

```
if (is_numeric($_GET['id']))
{
    // ho un valore numerico, posso procedere
}
```

Nota: le variabili che arrivano via GET o POST sono sempre **stringhe numeriche**, per cui in questi casi **is_numeric()** va preferito rispetto ad **is_int()**.

SQL Injection: **espressioni regolari**

Spesso i dati che aspettiamo in input possono essere descritti da una **espressione regolare**.

E' possibile fare ciò in due modi:

a) Usando i **pattern** nel tag **input** del **form HTML** (vedi **PPT-7**)

b) Utilizzando la funzione [preg_match\(\)](#) oppure la funzione [ereg\(\)](#) del PHP:

```
if (preg_match("/^[a-z0-9]{4,12}$/i ", $login))
{
    // $login rispetta il parametro, per cui posso effettuare la query
}
else
{
    // errore
}
```

SQL Injection: **eliminazione caratteri pericolosi**

Quando il type casting non ci può servire e non troviamo un'espressione regolare che possa filtrare adeguatamente un input, possiamo decidere di eliminare eventuali caratteri pericolosi o sostituirli con codici innocui.

In sostanza i caratteri potenzialmente dannosi sono quelli che hanno un significato all'interno di una query SQL, come ad esempio gli apici singoli e doppi, la virgola, il punto e virgola, e così via.

In alcune situazioni però tali caratteri devono essere ammessi: se un nostro utente si chiama ad esempio "Paperon de' Paperoni", l'apice fa parte del suo cognome, e sarà opportuno concederne l'inserimento in un ipotetico form di registrazione.

In altri casi invece questi caratteri possono essere eliminati o sostituiti senza problemi, sfruttando una tra le funzioni [str_replace\(\)](#), [preg_replace\(\)](#), [ereg_replace\(\)](#) o [strtr\(\)](#), ad esempio:

SQL Injection: **eliminazione caratteri pericolosi**

```
$input = str_replace("'", "", $input); // attenzione ai caratteri inseriti.
```

- Il primo parametro di `str_replace()` è la stringa da cercare (nell'esempio, un apice singolo);
- Il secondo è la stringa di sostituzione (nell'esempio, una stringa vuota);
- Il terzo è la stringa di partenza (nell'esempio, una ipotetica variabile `$input`).

N.B. Un approccio del genere può comunque risultare difficile da applicare, in quanto i caratteri da controllare possono essere svariati ed i DBMS si comportano in maniera differente, per cui sarà facile dimenticare qualche particolare.

SQL Injection: **escape delle stringhe**

Si possono “escapare” gli apici e i doppi apici, e questo può essere fatto nel codice tramite la funzione **addslashes()** e **stripslashes()** oppure usando la direttiva nel file **php.ini** **magic_quotes_gpc**.

Quando lasciate all’utente la possibilità di inserire un testo o un commento, che poi andrà a salvarsi nel vostro database, dovete **assicurarvi che alcuni caratteri particolari (come gli apostrofi) non vadano a danneggiare la query**, provocando degli errori sconcertanti.

Questo può avvenire intenzionalmente (l’utente, cioè, è consapevole del danno e sta cercando di attaccare volontariamente il vostro database; il termine tecnico è *sql injections*), ma anche casualmente: in tal caso la colpa sarà solo nostra.

Per evitare simili disastri, ci vengono in soccorso due funzioni native del PHP:

- **addslashes()** aggiungerà il backslash (\) davanti agli apici (‘), doppi apici (“) e ai backslash (\) presenti nella stringa passata. Dovrà essere usata prima di salvare le stringhe nel database.
- **stripslashes()** farà il procedimento inverso, togliendo i backslash davanti ai suddetti caratteri e dovrà essere usata una volta estratta la stringa del database e prima di stamparla a video.

SQL Injection: **escape delle stringhe**

Si possono “escapare” gli apici e i doppi apici, e questo può essere fatto nel codice tramite la funzione **addslashes()** e **stripslashes()**

Esempio:

```
$stringa = "Questa è una stringa di prova con l'apostrofo";
```

```
$stringa2 = addslashes($stringa);
```

```
echo $stringa2; // darà come risultato "Questa è una stringa di prova con l\'apostrofo"
```

```
$stringa = stripslashes($stringa2);
```

```
echo $stringa; //darà come risultato la stringa iniziale
```

SQL Injection: **escape delle stringhe**

Si possono “escapare” gli apici e i doppi apici, e questo può essere fatto usando la direttiva nel file `php.ini` **magic_quotes_gpc**.

Il file di configurazione `php.ini` del nostro webserver Apache ha al suo interno la direttiva **magic_quotes_gpc** (la sigla “gpc” sta per “get post cookies”).

Quando questa è impostata sul valore “ON”, non avremo bisogno di usare la funzione `addslashes()` nel salvare le nostre stringhe nel database, perché **ci penserà autonomamente il server ad aggiungere i backslashes** in tutti i valori GET, POST o COOKIES.

Comunque è da **sconsigliare l'uso di questa direttiva per due motivi:**

- 1) In genere i server hanno di default la direttiva a ON, ma può capitare in caso di trasferimento del sito di trovarci di fronte a un server con la direttiva impostata a *OFF*. In questo caso, dovremmo aggiungere manualmente la funzione `addslashes()` a tutte le stringhe presenti nel nostro database, perdendo inutilmente del tempo.
- 2) La direttiva è deprecata in PHP5 e sarà del tutto rimossa con PHP6 per ragioni di efficienza.

SQL Injection: **conclusioni**

Abbiamo visto come sia possibile trasformare una innocente query SQL in una istruzione potenzialmente distruttiva per il nostro database, ed abbiamo visto quanto sia relativamente semplice effettuare una serie di controlli che hanno lo scopo di limitare i problemi.

Le soluzioni che proposte, type casting, controllo del tipo di dato, espressioni regolari, eliminazione o escape delle stringhe, possono comunque essere integrate tra di loro.

Quando ci si trova davanti ad input diversi da quelli attesi ci sono più vie da scegliere, a seconda delle situazioni e della propria opinione personale: è possibile utilizzare dei valori predefiniti per sostituire gli input non regolari, oppure si può scegliere di predisporre un messaggio d'errore, o anche di costruire un sistema di logging che registri tutti i tentativi di **exploit**.

Indipendentemente dalla via che si decide di intraprendere, l'unico punto che dobbiamo avere ben chiaro è: **controllare ogni input**.

SQL Injection: **link**

Un **exploit** ("/ɪk 'splɔɪt/", tradotto dall'inglese "sfruttare") è un termine usato in informatica per identificare una tipologia di script, virus, worm, porzione di dati o binario che sfrutta un bug o una vulnerabilità per creare comportamenti non previsti in software, hardware, o in sistemi elettronici (solitamente computerizzati), ad es. ottenere l'accesso a sistemi informatici, permettere l'acquisizione dei privilegi amministrativi, o attacchi denial of service (DoS o il correlato DDoS) (fonte wikipedia)

- [Manuale PHP – database security](#) (inglese); una pagina del manuale PHP che descrive alcuni problemi legati alla sicurezza del database.
- [SQL Injection by OpenBeer](#) (italiano, PDF); interessante paper in italiano che tratta l'argomento delle SQL injection e propone alcuni esempi per PHP/MySQL, PHP/PostgreSQL ed anche ASP/MDB
- [Espressioni regolari](#) – le basi per poter utilizzare le espressioni regolari e creare quindi degli appositi filtri per i nostri dati.