SQL Injection: le tecniche, i tool ed esempi pratici

Antonio Parata
collaboratore OWASP-Italy
http://www.ictsc.it
antonio.parata@ictsc.it

OWASP

SMAU E-Academy 2006

http://www.owasp.org/index.php/Italy

The OWASP Foundation
http://www.owasp.org

SQL Injection: le tecniche, i tool ed esempi pratici

- Concetti di base di software security
- Introduzione all'SQL Injection
- SQL Injection e metodi di inferenza
- Il tool: Sqlmap
- Rendersi immuni all'SQL Injection
- Conclusioni

Concetti di base di software security

Con il termine *Software Security* si intende la costruzione di software sicuro.

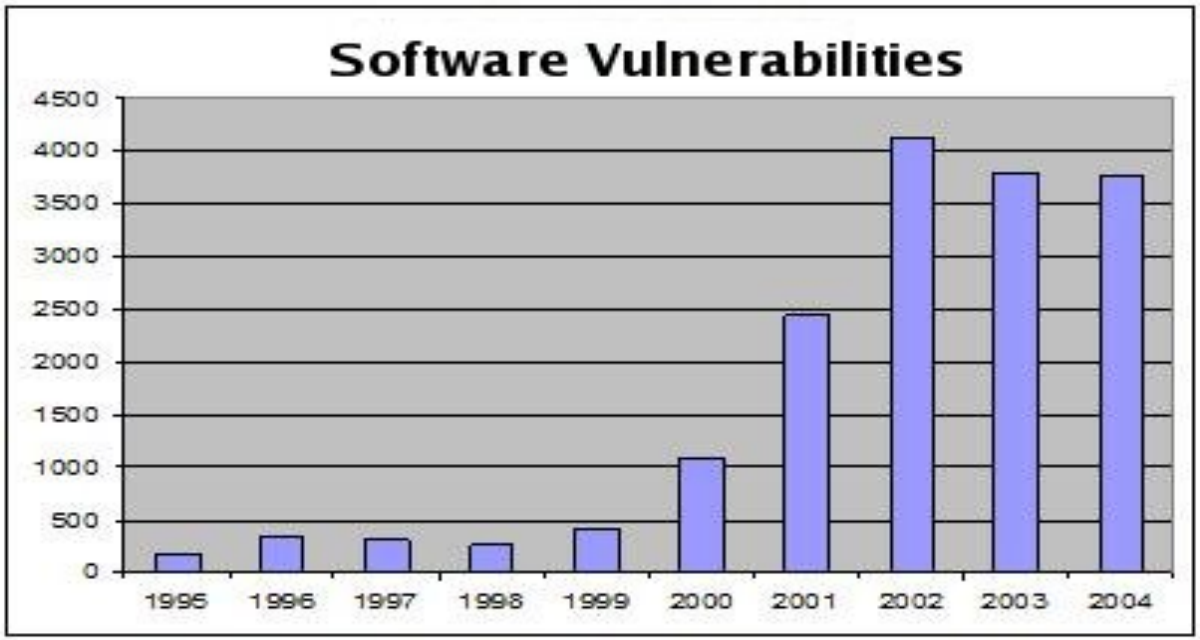
Negli ultimi anni l'interesse verso la *software security* sta crescendo notevolmente. Ciò perchè i software risultano sempre più:

- **Costantemente connessi alla rete**
- **Richiedono una sempre maggiore estendibilità**
- **La loro complessità sta aumentando considerevolmente**

Fare affidamento sulla sola *network security* non basta più.

Concetti di base di software security

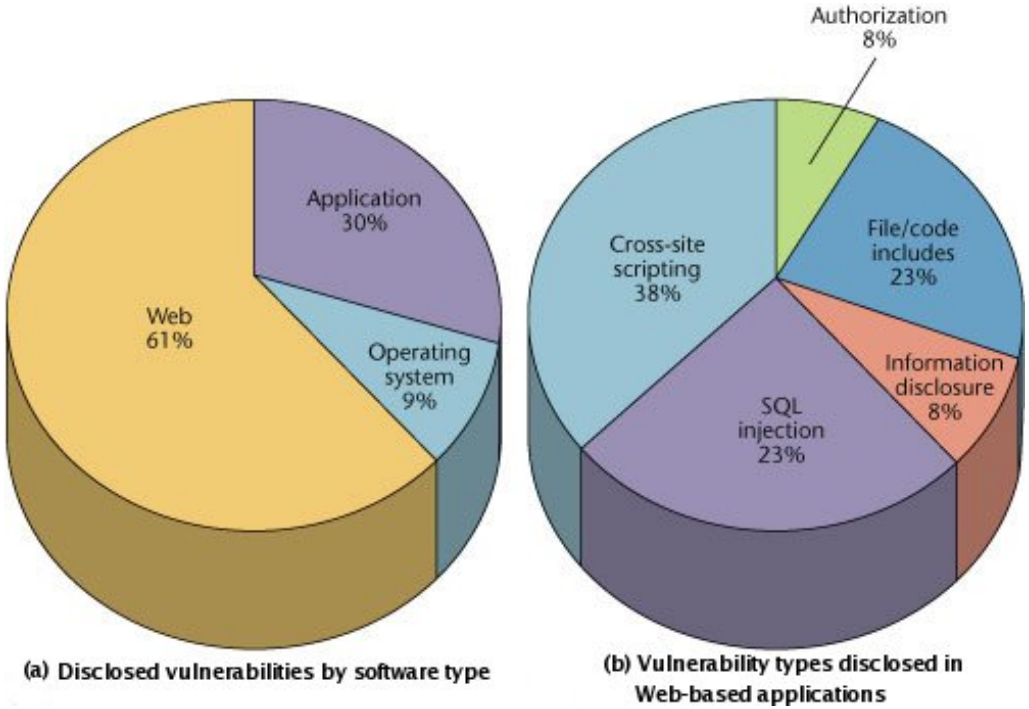
A partire dal 2000/2001 il numero delle vulnerabilità software è cresciuto drasticamente (fonte CERT)



É dunque necessario considerare quali siano le maggiori vulnerabilità software a cui andiamo incontro e in che modo possiamo difenderci.

Introduzione all'SQL Injection

Ma perchè parlare proprio di applicazioni web e dell'attacco SQL Injection?

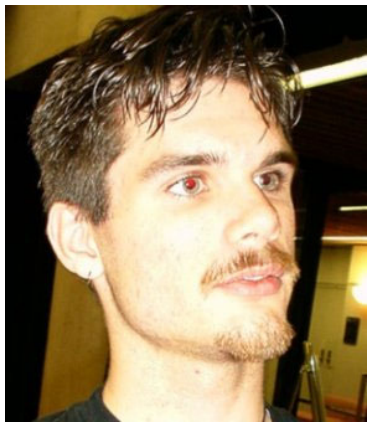


Come si può notare le applicazioni web sono le più colpite e una delle maggiori vulnerabilità è l'SQL Injection (si tenga conto che il Cross-site scripting è un attacco il cui impatto è decisamente minore rispetto all'SQL Injection).

Fonte Securityfocus, maggio 2006

Introduzione all'SQL Injection

Un po' distoria...



Nel 1998 Rain Forest Puppy pubblica su Phrack 54
un articolo dal titolo *“NT Web Technology Vulnerabilities”.*
Anche se il termine SQL Injection non viene menzionato
viene presentato alla comunità uno degli attacchi più
insidiosi che le applicazioni web abbiano mai incontrato.

Nel settembre del 2005 David Litchfield publica un paper
dal titolo *“Data-mining with SQL Injection and Inference”*
basato sulla presentazione fatta al Blackhat europe.
É la nascita di una nuova metodologia di SQL Injection
basata su tecniche di inferenza.
Altro duro colpo per la sicurezza delle applicazioni web.

Introduzione all'SQL Injection

L'SQL Injection è un particolare tipo di attacco il cui scopo è quello di indurre il database ad eseguire query SQL non autorizzate.

Consideriamo la seguente query:

SELECT * FROM Tabella WHERE username='$user' AND password='$pass'

$user e *$pass* sono impostate dall'utente e supponiamo che nessun controllo su di esse venga fatto.
Vediamo cosa succede inserendo i seguenti valori:

$user = ' or '1' = '1
$pass = ' or '1' = '1

La query risultante sarà:

SELECT * FROM Tabella WHERE username='' or '1' = '1' AND password='' or '1' = '1'

Introduzione all'SQL Injection

Tecniche per ottenere il contenuto di una qualsiasi tabella pre-inferenza:

Ciò che in genere veniva fatto è unire alla query originale, un'altra query in modo che tra i valori ritornati venissero inclusi anche i valori di un'altra tabella.

Ad esempio, inseriamo il seguente valore di *id:*

id = 1' UNION ALL SELECT * FROM crediCard WHERE '1' = '1

Otterremo la seguente query:

SELECT * FROM Tabella WHERE id='1' UNION ALL SELECT * FROM creditCard WHERE '1' = '1'

Unica accortezza, dobbiamo fare in modo che la query aggiunta contenga un numero di parametri uguale a quella originale. Compito non molto difficile, basta aggiungere progressivamente parametri (anche senza significato come 'a') fino a quando non otteniamo una query corretta.

Introduzione all'SQL Injection

Alcuni problemi…

La presenza di parentesi potrebbe rendere più complicato rendere la query corretta, comunque nulla che non si possa risolvere.

La presenza di ulteriori parametri dopo quello vulnerabile potrebbero non permettere di effettuare l'unione, ad esempio:

SELECT * FROM Tabella WHERE id='$id' AND tipo='tipo1'

Si potrebbe inserire un simbolo di commento alla fine della query per fare in modo che l'ultima sentenza non venga considerata, comunque questa soluzione potrebbe creare errori sintattici.

Non è detto che il sito web contenga una sezione in cui vengano visualizzati tutti gli elementi prodotti dalla query.

SQL Injection e metodi di Inferenza

Con il termine *Inferenza* si intende una conclusione tratta da un insieme di fatti o circostanze.

In pratica ciò che viene fatto è eseguire alcuni test di verità sui parametri vulnerabili e in base al risultato dedurre il valore del parametro.

I test effettuati sono di tipo binario (vero/falso).

Per poter eseguire con successo l'attacco è dunque necessario saper distinguere il significato di vero da quello di falso.

SQL Injection e metodi di Inferenza

Supponiamo che nell'applicazione che stiamo testando, vi sia un parametro vulnerabile, ad esempio *id*.

Per prima cosa creiamo una query sintatticamente corretta che restituisca un valore falso, ad esempio:

id = aaa' and '1' = '2

Con questa query otteremo sicuramente un valore falso. In particolare il valore falso è rappresentato dal contenuto della pagina web ritornata.

SQL Injection e metodi di Inferenza

Diamo inizio alle danze…

Ciò che andremo a fare è cercare di determinare il valore di ogni singolo carattere del parametro di cui vogliamo scoprire il valore.

In pratica, ci chiederemo:

"Il carattere 1 del parametro è uguale ad a?" => NO
"Il carattere 1 del parametro è uguale a b?" => NO
"Il carattere 1 del parametro è uguale a c?" => NO
"Il carattere 1 del parametro è uguale a d?" => SI

Conosciamo il valore del primo carattere, procediamo con il secondo

"Il carattere 2 del parametro è uguale ad a?" => NO
"Il carattere 2 del parametro è uguale a b?" => SI

Conosciamo il valore del secondo carattere, procediamo in questo modo fino a scoprire i valori di tutti i caratteri che compongono il valore del parametro.

SQL Injection e metodi di Inferenza

Come facciamo a capire quando abbiamo esaminato tutta la stringa?

Abbiamo accennato al fatto che la funzione *substring* ritorna un valore null (pari al valore ASCII 0) quando l'indice da cui cominciare a considerare la sottostringa è maggiore della lunghezza della stringa stessa.

A questo punto basta verificare che il valore del carattere su cui stiamo facendo inferenza sia uguale a 0 per dedurre che abbiamo esaminato tutta la stringa.

Ma...
e se il valore della variabile su cui stiamo facendo inferenza contenesse il carattere ASCII 0?

Generalmente ciò non accade con gli input inseriti dall'utente, ma invece può accadere nel caso volessimo ricavare il valore di un file (binario).

SQL Injection e metodi di Inferenza

Soluzione: fare inferenza sulla lunghezza.

In pratica quando incontriamo il carattere *null*, andremo a fare inferenza sulla lunghezza della stringa, è inoltre necessario tenere memoria di quanti caratteri ho fino ad ora esaminato.

Le fasi da intraprendere sono:

Detto *n* il numero di caratteri fino ad ora analizzati, mi chiedo:

La lunghezza della stringa rappresentante il valore è uguale a n?

SI => ho finito di fare inferenza

NO => il valore contiene il carattere ASCII 0, continuo a fare inferenza sul prossimo carattere.

SQL Injection e metodi di Inferenza

Un breve esempio.

*Goal: sappiamo che il parametro **id** è vulnerabile all'SQL Injection. Supponendo che il server sia eseguito con i privilegi di amministratore (root), sfruttiamo questa vulnerabilità per riuscire a ricavare il contenuto del file **/etc/shadow**. Inoltre supponiamo che il DBMS sia MySql.*

A questo punto sappiamo come operare. La richieste che andremo ad eseguire sono della seguente forma:

http://www.mysite.com/index.php?id=20'%20and%20ASCII(SUBSTRING(LOAD_FILE("/etc/*shadow*"),1,1))=97/*

Supponendo di aver ricavato **8** caratteri, e di aver incontrato il valore **null**, la query da eseguire per fare inferenza sul valore della lunghezza è:

http://www.mysite.com/index.php?id=20'%20and%20CHAR_LENGTH(LOAD_FILE("/etc/*shadow*"))=8/*

Il tool: Sqlmap (by Daniele Bellucci)

Al momento supporta solo database Mysql (versione 5)
Permette di ottenere in automatico il valore di intere tabelle tramite inferenza

```
./sqlmap.py -u http://localhost/blind/showcontent.php?id=TOKEN -t TOKEN
-d 1 –database blind –table news –dump -v

[*] starting at: 02:37:53
[02:37:53] fetching total attributes for db:blind table:news
[02:37:53] retrieved: 3
...
DATABASE: blind TABLE: news
+-----------------------+----+-----------+
|news                   |id  |highlight  |
+-----------------------+----+-----------+
|ciao da daniele        |1   |numero 1   |
|sempre ciao da daniele |2   |numero 2   |
+-----------------------+----+-----------+

[*] shutting down at: 02:37:58
```

Rendersi immuni all'SQL Injection

Due metodologie di difesa:

Approccio blacklist: ho una lista di caratteri non permessi.

Approccio whitelist: ho una lista di caratteri permessi.

Rendersi immuni all'SQL Injection

Approccio blacklist

L'origine di tutti i mali è il carattere **'**.

1) Per cui cominciamo ad includere ' nella blacklist.

Problemi:

- il carattere potrebbe essere codificato (eg. %27)

- il carattere ' necessita di essere incluso tra i caratteri validi

Rendersi immuni all'SQL Injection

Approccio blacklist

2) Utilizziamo la funzione **addslashes** di PHP per mettere davanti al carattere ' il carattere / (o in alternativa possiamo usare **mysql_real_escape_string** per le versioni più recenti di PHP).

Problemi:

- Innanzitutto dobbiamo ricordarci di utilizzare la funzione **stripslashes** quando preleviamo i dati dal database, altrimenti ci ritroveremo caratteri indesiderati.

- Questa soluzione non sempre funziona, consideriamo la seguente query:

SELECT * FROM Tabella WHERE id=$parametroIntero

se come $**parametroIntero** mettiamo il valore **2 or 1 = 1** la query diventa:

SELECT * FROM Tabella WHERE id=2 or 1 = 1

Rendersi immuni all'SQL Injection

Approccio blacklist

3) Allora inseriamo nella blacklist anche il carattere di spazio.

Problemi:

- In genere il carattere di *spazio* non è inserito in tale liste, proprio perchè è probabilmente il carattere più utilizzato, per cui negarlo sarebbe un grosso problema per l'usabilità del sito.

- Questa soluzione non sempre funziona, consideriamo la query:

  SELECT * FROM Tabella WHERE id='$parametro'

  se come **$parametro** mettiamo il valore **a'or(1=1)#** la query diventa:

  SELECT * FROM Tabella WHERE id='a'or(1=1)#'

  che in MySql è perfettamente legale.

Rendersi immuni all'SQL Injection

Approccio whitelist

Dagli esempi precedenti abbiamo potuto capire che il filtraggio dei caratteri non validi non è un'operazione semplice. Senza contare che in futuro potrebbero apparire nuovi pericolosi caretteri non inizialmente inseriti nella nostra lista.

Nell'approccio **whitelist** abbiamo un insieme di caratteri validi. Ad ogni richiesta fatta, se l'input ricevuto contiene dei caratteri non presenti in tale lista, allora segnaleremo un errore.

Ciò comporta una attenta definizione della lista in fase di definizione dei requisiti dell'applicazione oltre che una corretta gestione dei caratteri permessi (se permetto il carattere ' e poi non lo gestisco adeguatamente non risolvo nulla).

Rendersi immuni all'SQL Injection

Alcune considerazioni...

Nelle applicazioni web esistono due tipologie di parametri:

Parametri il cui valore deve essere fornito dall'utente (si pensi ad esempio alle form di autenticazione).

Parametri il cui valore è fisso e indipendente dalla sessione di navigazione (ad esempio i parametri che identificano l'oggetto nei vari negozi on-line).

Potremmo chiamare i primi **parametri dinamici** e i secondi **parametri statici.**

Per quanto riguarda i parametri dinamici, la loro validazione deve essere fatta ad-hoc dal progettista (ad esempio implementando una funzione di validazione basata su whitelist).

Concentriamoci invece sui parametri statici.

Rendersi immuni all'SQL Injection

Spesso sono i parametri statici a dare più problemi.

Gli sviluppatori sono consci del fatto che devono controllare l'input dell'utente nel caso di parametri dinamici (verificare se la password è correttà, verificare se lo username è corretto, e via dicendo).

I parametri statici invece sono blandamente controllati fondamentalmente perchè:

1) si suppone che non siano modificabili, non essendo richiesto all'utente di inserirne un valore.

2) sono in numero decisamente maggiore rispetto ai parametri dinamici, e il loro controllo diventa difficoltoso.

Fortunatamente è abbastanza facile verificare se i parametri statici sono stati alterati.

Vediamo in che modo.

Rendersi immuni all'SQL Injection

Ciò che si fa è inserire un ulteriore parametro il quale identifica univocamente i restanti parametri della richiesta (sia che sia di tipo GET che di tipo POST).

Vediamo come fare in PHP (versione 5).

Utilizzando la class **PEAR::Crypt_HMAC**, possiamo utilizzare la funzione HMAC per creare un *hash* dei valori dei parametri. Per creare l'hash è necessaria una chiave segreta.

A questo punto tutti i parametri delle varie richieste saranno associati al rispettivo HMAC. Ogni volta che viene invocata una richiesta è necessario controllare il relativo HMAC per verificare che i parametri non sono stati alterati. Se sono stati alterati segnalare l'errore.

In questo modo non è possibile per un attacker modificare i parametri senza che l'applicazione se ne accorga.

Rendersi immuni all'SQL Injection

```
<?php
require_once('Crypt/HMAC.php');

Function create_parameters($array) {
	$clearText = '';
	$arrayParameters = array();

	/* Costruisco la stringa con la coppia chiave valore */
	foreach($array as $paramName => $paramValue) {
		$clearText = $clearText.$paramName.$paramValue;
		$arrayParameters[ ] = “$paramName=$paramValue”;
	}

	$hmac = new Crypt_HMAC(SECRET_KEY, 'sha1');
	$hashValue = $hmac->hash($clearText);
	$arrayParameters[ ] = “hash=$hashValue”;
	$queryString = join (&amp;', $arrayParameters);

	return $queryString;
}
```

Rendersi immuni all'SQL Injection

```
<?php
require_once('Crypt/HMAC.php');

Function verify_parameters($array) {
        $clearText = '';
        $arrayParameters = array();

        if (!in_array('hash',$array) {
                return FALSE;
        }

        $hash = $array['hash'];
        unset($array['hash']);
        foreach($array as $paramName => $paramValue) {
                $clearText = $clearText.$paramName.$paramValue;
                $arrayParameters[ ] = “$paramName=$paramValue”;
        }

        $hmac = new Crypt_HMAC(SECRET_KEY, 'sha1');
        if ($hash != $hmac->hash(SECRET_KEY) {
                return FALSE;
        }
        else {
                return TRUE;
        }
}
```

Rendersi immuni all'SQL Injection

Vediamo come creare un nuovo link:

```
<?php

define('SECRET_KEY', ”un6r34k4b13”); // Inserire la propria chiave segreta

echo '<a href=”script.php?'.create_parameters(array('name' => 'value')).'”>link</a>
?>
```

Per la verifica invece:

```
<?php

define('SECRET_KEY', ”un6r34k4b13”); // Inserire la propria chiave segreta

if (!verify_parameters($arrayParametriRichiesta) {
      die(“VIOLAZIONE DELLA SICUREZZA!”);
}
// altro...
?>
```

Conclusioni

In questa presentazione abbiamo appreso che:

- Le applicazioni web, rappresentano la tipologia di software più sviluppata, ma anche la più colpita.

- L'SQL Injection con metodi di inferenza, permette di ottenere non solo il valore dei campi di una qualsiasi tupla di una qualsiasi tabella, ma di ricavare anche il contenuto di file presenti sul filesystem.

- Possiamo classificare i parametri come **statici** e **dinamici**. I parametri dinamici necessitano di un'opportuna validazione (basata su whitelist), mentre il controllo dei parametri statici può essere facilmente risolto ricorrendo alla funzione HMAC.

Grazie per l'attenzione.

Domande?