- Una buona soluzione è quella di utilizzare una libreria ORM, come EntityFramework, Hibernate o iBatis.
- Limitare l'accesso agli oggetti e alle funzionalità di database, in base al principio del minimo privilegio.

Esempio - Javascript per Client Second Order Sql Injection.
Forma non corretta:

```
var userId = 5;
var query = connection.query('SELECT * FROM users WHERE id = ?', [userId],
function(err, results) {
 //query.sql returns SELECT * FROM users WHERE id = '5'
});
```

Forma corretta:

```
var post = {id: 1, title: 'Hello MySQL'};
var query = connection.query('INSERT INTO posts SET ?', post, function(err,
result) {
 //query.sql returns INSERT INTO posts SET `id` = 1, `title` = 'Hello MySQL'
});
```

7.11.7 Client Sql Injection

Come riconoscerla

Utilizzando questa vulnerabilità un attaccante potrebbe utilizzare i canali di comunicazione tra l'applicazione e il suo database, ossia modificando ad arte una query SQL testuale. Ciò è reso possibile nei casi in cui l'applicazione costruisce dinamicamente le query concatenandole con l'input dell'utente. Se non a questo non sono stati applicati i controlli di validità, l'attaccante potrebbe modificare i comandi SQL nel senso da lui desiderato.

Come difendersi

Valgono le considerazioni e le contromisure esposte nel punto precedente.

Esempio - Javascript per Client SQL Injection
Forma non corretta:

```
var info = {
 userid: message.author.id
}

connection.query("SELECT * FROM table WHERE userid = '" + message.author.id + "'",
info, function(error) {
 if (error) throw error;
});
```

Forma corretta:

```
var sql = "SELECT * FROM table WHERE userid = ?";
var inserts = [message.author.id];
sql = mysql.format(sql, inserts);
```

Per ulteriori informazioni vedere: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.11.8 Cross-Site Request Forgery (CSRF)

Come riconoscerla

Il Cross-Site Request Forgery , abbreviato CSRF o anche XSRF, è una vulnerabilità a cui sono esposti i siti web dinamici quando sono progettati per ricevere richieste da un client senza meccanismi per controllare

se la richiesta sia stata inviata intenzionalmente oppure no. Diversamente dal cross-site scripting (XSS), che sfrutta la fiducia di un utente in un particolare sito, il CSRF sfrutta la fiducia di un sito nel browser di un utente.
Nelle applicazioni Web 2.0 Ajax comunica con i servizi Web di back-end tramite XML-RPC, SOAP o REST. È possibile invocarli tramite interrogazioni di tipo GET e POST che effettuano chiamate cross-site ai servizi web. La tecnologia di tipo Cross-Site Request Forgery permette di manipolare queste chiamate indebolendo la sicurezza del sistema.
Un attaccante induce la sua vittima a inviare inconsapevolmente una richiesta HTTP dal suo browser al sistema web dove è attualmente autenticato. Il sistema, vulnerabile al CSRF, avendo la certezza che la richiesta provenga dall'utente già precedentemente autenticato la esegue senza sapere che in realtà dietro la richiesta si cela un'azione pensata dall'attaccante come ad esempio un trasferimento di fondi, un acquisto di beni, una richiesta di dati o qualsiasi altra funzione offerta dall'applicazione vulnerabile. Ci sono innumerevoli modi con i quali un utente può essere ingannato nell'inviare una richiesta pensata da un attaccante: per esempio nascondendola in un elemento HTML di un'immagine, una XMLHttpRequest o un URL.

Come difendersi

Usare framework, librerie, moduli e in generale codice fidato che permettano allo sviluppatore di evitare l'introduzione di questa vulnerabilità. L'uso di un token antifalsificazione è di solito la scelta migliore.
Nei form che permettono operazioni importanti inserire un campo hidden valorizzandolo con una stringa random. La stessa stringa, va impostata come variabile di sessione, in questo modo non è rintracciabile lato client ed è nota solo al server. Una volta compiuta la submit del form, se il valore della variabile di sessione corrisponde alla value del sopracitato campo hidden, la richiesta è da considerarsi valida.
Identificare le operazioni che possano risultare pericolose e quando un utente genera un'operazione di questo tipo inviare una richiesta addizionale di conferma all'utente, per esempio, la richiesta di una password, che deve essere verificata prima di eseguire l'operazione.
Non utilizzare il metodo GET per il passaggio di parametri da una pagina web all'altra soprattutto per quelle richieste che comportano un cambiamento di stato come ad esempio la modifica di dati. Controllare il campo di intestazione HTTP referer per vedere se la richiesta è stata generata da una pagina valida.
Verificare che il sistema sia esente da vulnerabilità di tipo cross-site scripting poiché la difesa da CSRF può essere rafforzata da queste contromisure.
Dal lato utente è buona abitudine eseguire sempre il logout da siti web sensibili prima di visitare altre pagine web.

Per ulteriori informazioni vedere: http://cwe.mitre.org/data/definitions/352.html,
CWE-352: Cross-Site Request Forgery (CSRF).

Esempio:
Viene introdotto per un campo in input un token antifalsificazione:

```
<form action="/Home/Test" method="post">
  <input name="__RequestVerificationToken" type="hidden"
      value="6fGBtLZmVBZ59oUad1Fr33BuPxANKY9q3Srr5y[...]" />
  <input type="submit" value="Submit" />
</form>
```

Per evitare l'invio del token in JSON, da parte dello script Ajax, lo si può includere in un'intestazione http dettagliata:

```
<script>
  @functions{
    public string TokenHeaderValue()
    {
      string cookieToken, formToken;
      AntiForgery.GetTokens(null, out cookieToken, out formToken);
      return cookieToken + ":" + formToken;
    }
  }
```