Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/94.html, Improper Control of Generation of Code ('Code Injection') CWE-94.

7.11.2 Client DOM Stored Code Injection

Come riconoscerla

L'utente malintenzionato può attraverso questo tipo di vulnerabilità causare la riscrittura di pagine web e l'inserimento di script dannosi per la sicurezza.
Una vulnerabilità persistente (o stored) come la "Client DOM Stored Code Injection" è una variante più pericolosa di cross-site scripting con manipolazione di codice: si verifica quando i dati forniti dall'attaccante vengono salvati sul server, e quindi visualizzati in modo permanente sulle pagine normalmente fornite agli utenti durante la normale navigazione.

Come difendersi

Occorre evitare qualsiasi esecuzione dinamica del codice. Se è proprio necessaria, anziché utilizzare i dati sul lato client, inclusi i dati precedentemente memorizzati nella cache dalla stessa applicazione, utilizzare solo dati attendibili provenienti dal server.
Evitare di chiamare dinamicamente una funzione senza averne prima bonificato l'input.
Nel caso debba essere impostato del codice Javascript per delle chiamate dinamiche, vanno utilizzati solo metodi predefiniti o codice Javascript non influenzabile da variabili dinamiche o non dipendente da routine tipo "eval()" non particolarmente sicure.

Esempio di codice Javascript sicuro:

```
window.setInterval("timedFunction();", 1000);
```

Per ulteriori informazioni ed esempi si veda: http://cwe.mitre.org/data/definitions/94.html,
Improper Control of Generation of Code ('Code Injection') CWE-94.

7.11.3 Client Dom Stored XSS

Come riconoscerla

Un malintenzionato può utilizzare l'accesso legittimo all'applicazione per inviare dati ingegnerizzati al database dell'applicazione. Quando un altro utente accede in seguito, le pagine Web potrebbero essere riscritte con i dati salvati e potrebbero essere attivati script dannosi.
L'applicazione crea pagine web che includono dati provenienti dal database, incorporati direttamente nell'HTML della pagina. Il browser, quindi, li visualizza come parte della pagina.
Il problema nasce quando questi dati salvati sono stati immessi da un altro utente. Se i dati includono frammenti HTML o Javascript malevoli, anche questi vengono visualizzati (o eseguiti), sebbene la vittima non si accorga dell'inganno sottostante. La vulnerabilità è perciò il risultato dell'incorporazione di dati arbitrari provenienti dal database, senza prima codificarli. La codifica trasforma i caratteri malevoli in normale testo, e il browser non può più trattarli come codice valido HTML/Javascript.

Come difendersi

I parametri devono essere limitati a un set di caratteri consentito e l'input non convalidato deve essere eliminato. Oltre ai caratteri, occorre controllare il tipo di dati, la loro dimensione, l'intervallo di validità, il formato e l'eventuale corrispondenza all'interno dei valori previsti (white list). Sconsigliata invece la black list, ossia una lista di valori non consentiti: l'elenco sarebbe sempre troppo limitato, rispetto ai casi che potrebbero verificarsi.

L'account con il quale l'applicazione viene avviata deve avere molte restrizioni e non deve godere di privilegi non necessari.
La convalida non sostituisce la codifica (encoding), ossia la neutralizzazione di tutti i caratteri potenzialmente eseguibili. Tutti i dati dinamici, indipendentemente dall'origine, devono essere codificati prima di incorporarli nell'output. La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
Nell'intestazione della risposta HTTP Content-Type, definire esplicitamente la codifica dei caratteri (set di caratteri) per l'intera pagina.
Impostare il flag httpOnly sul cookie di sessione, per impedire agli exploit XSS di rubarlo.

Esempio di HTML richiamato nel codice Javascript: la stringa in uscita è codificata nella pagina Html prima che venga visualizzata nell'etichetta relativa:

```
public class StoredXssFixed
{
    public string foo(Label lblOutput, SqliteConnection connection,
HttpServerUtility Server, string id)
    {
        SqliteConnection connection = new SqliteConnection(connectionString)
        string sql = "select email from CustomerLogin where customerNumber = " +
id;
        SqliteCommand cmd = new SqliteCommand(sql, connection);
        string output = (string)cmd.ExecuteScalar();
        lblOutput.Text = String.IsNullOrEmpty(output) ? "Customer Number does not
exist" : Server.HtmlEncode(output);
    }
}
```

Esempio Javascript per Client Dom Stored XSS.

```
Forma non corretta (routine completa):
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8">
 <title>XSS Example</title>
 <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.4/jquery.min.js"></script>
 <script>
  $(function() {
   $('#users').each(function() {
    var select = $(this);
    var option = select.children('option').first();
    select.after(option.text());
    select.hide();
   });
  });
 </script>
</head>
<body>
 <form method="post">
  <p>
   <select id="users" name="users">
    <option
value="bad">&lt;script&gt;alert(&#x27;xss&#x27;);&lt;/script&gt;</option>
   </select>
  </p>
 </form>
</body>
</html>
```

Forma corretta (fix relativa alle stringa modificata):