CSE 127
Computer Security

Fall 2015

Web Security I: SQL injection

Stefan Savage
The Web creates new problems

- Web sites are *programs*
  - Partially implemented in browser
    » Javascript, Java, Flash
  - Partially implemented in server
    » PHP, SQL, Ruby, ASP*, etc

Browser

OS

Hardware

request

reply

website

Network
Web languages

- Most database and web scripting languages are also untyped.

- Unforeseen user input allows attacker to side-step programmer intent:
  - Server attacks: SQL injection
  - Browser attacks: Cross-site scripting (XSS) [should be called Web page injection]
Structured Query Language – mostly std syntax for sending queries to databases

Examples

```
SELECT * FROM books WHERE price > 100.00 ORDER BY title

INSERT INTO books (title, price) VALUES ("War and Peace", 15.25)
```

Data types are generally either numbers, strings, or field names
How SQL gets used on the Web

- Server-side Web application processes user input (e.g. via PHP)
- Formats it into SQL query
- Sends query to DB
- Formats DB response for Web page

- This is how every e-commerce site you’ve ever used likely works
Simple example

```
dim userName, password, query
dim conn, rs

userName = Request.Form("userName")
password = Request.Form("password")

set conn = server.createObject("ADODB.Connection")
set rs = server.createObject("ADODB.Recordset")

query = "select count(*) from users where userName=\'" & userName & "\' and userPass=\'" & password & "\'
conn.Open "Provider=SQLLOEDB; Data Source=(local); Initial Catalog=myDB; User Id=sa; Password=
rs.activeConnection = conn
rs.open query
if not rs.eof then
    response.write "Logged In SQL world"
else
    response.write "Bad Credentials"
end if
```

Query="select count(*) from users where userName='" & userName & "' and userPass='" & password & "'"
Queries get expressed slightly differently in different languages:

Query="select count(*) from users where userName='" &userName & "' and userPass='" & password & "'"

$username = $_GET['user'];
$password = $_GET['pass'];
$sql = "SELECT count(*) " .
    "FROM users " .
    "WHERE userName='$username'" .
    "AND userPass = '$password'";
What if user provides malicious input that is passed to the database?
This is an **input validation vulnerability**

Unsanitized user input in SQL query to backend database changes the meaning of query

**Specific case of more general command injection**
Uh-oh... user provides input

user: `savage` ;-- (-- is a comment)
password: `anything`

What happens?

Query="select count(*) from users where userName=''
&userName & '' and userPass=''' & password & ''"
Uh-oh... user provides input

user: savage
password: ’--'
What happens?

Query="select count(*) from users where userName='" &userName & "' and userPass='" & password & "'"

Query="select count(*) from users where userName='savage' and userPass='--'"
Uh-oh... user provides input

user: anyone
password: crud’ or 1=1 --
What happens?

Query="select count(*) from users where userName='"&userName & "' and userPass='" & password & "'"

Query="select count(*) from users where userName='anyone' and userPass='crud’ or 1=1 --'"
Some basics of HTTP

URL’s only allow ASCII-US characters. Encode other characters:

\%0A = newline
\%20 = space

Protocol
Hostname
Port
Path
Query


Special characters:
+ = space
? = separates URL from parameters
\% = special characters
/ = divides directories, subdirectories
# = bookmark
& = separator between parameters
Aside: URL encoding

- Frequently input is encoded into url:

- Can still encode spaces, escapes, etc
  - E.g., ‘-> %27, space -> %20, = -> %3D
  - http://website.com/products.asp?user=crud%27%20OR%201%3D1%20-
This isn’t only a problem for authentication

- `any' OR 'x'='x`
  - Entire database is returned
- `any'; DROP TABLE prodinfo; --`
  - Entire database is deleted
- `any'; exec cmdshell('foo.exe') --`
  - Run foo.exe (or any application on system with any arguments)
- This only scratches the surface… UNION lets you append a second query, INSERT INTO lets you add records, etc…

Whenever you see a string in your browser like:
  - it is ripe for injection
This isn’t only a problem for authentication.
What's the problem here?

- Unvalidated input make difficult by
  - Untyped variables
  - Multiple levels of interpretation

What to do:
- Input validation
- Static queries
- Proxying
Input Validation

- Strip out comments and (if possible) SQL keywords from input
  - Can we just strip out quotes?

Probably not in general
- What about names like O'Brien?
Better escaping
- Remove ambiguity about whether the quote is a string literal or syntactic sugar
- Replace ‘ in user input with token that indicates it should be treated as a string literal
  » E.g. Convert ‘ into \'
- Need to cover all cases (e.g. ‘)
  » mysql_real_escape_string();
- php provides mode to deal with this: magic_quotes sanitizes input automatically
  » However, still allows quoting problems: canonicalization
Input Validation (3)

- Integer field validation
  - Quoting doesn’t solve problem
    - SELECT fields from TABLE where id = 52 OR 1=1
  - Need to validate that integers are integers
  - Validate that integers are in expected range
Integer field validation issues

- Many languages, such as Perl and PHP, will convert a string into a number as long as the string begins with digits.
- Each language and database has different quoting rules.

```php
if ($category > 0) {
    $categ = "AND catid=$category ";
} elseif ($category == 0) {
    ....
}

Return true even if $category = "2 union ...";
```

“AND catid=2 union...”
Second-Order SQL Injection

- Input parameters not the only risk
- Second-order SQL injection: data stored in database is later used to conduct SQL injection
- For example, user manages to set username to admin’ --
  - This vulnerability could exist if string escaping is applied inconsistently (e.g., strings not escaped)
  - UPDATE USERS SET passwd='cracked' WHERE username='admin' --’
Solution: treat all parameters as dangerous
Another solution: prepared statements

- Create query as a parameterized statement
  - `q = "SELECT * FROM scenarios WHERE username=? AND password=?";`
  - `PreparedStatement ps = conn.prepareStatement(q);`
  - `ps.setObject(1, user);`

- Create static parameterized query and explicitly fill in parameters rather than creating a dynamic query
  - Bind variables (indicated by ?) can also be explicitly typed (e.g., `ps.setInt()``

- Somewhat awkward syntax and can’t be used in some cases
Mitigation strategies

- Be careful not to leak database schema
  - Make attacker guess about field names
- Some databases allow certain commands to be executed to privilege of requestor (typically using user login as key to privs)
  - E.g. database can only be read not written
  - E.g. external programs cannot be invoked by Web-originated request
- Encrypt sensitive data in DB (e.g. credit cards)
  - Even if data stolen, data has little value
  - Tricky issue: key management (key can’t be in DB)
Web Application Gateway (WAG)

- WAG works like a reverse web proxy
- It can interpret more information in the HTTP protocol and HTML content, it checks
  - Form inputs are within our expectation
  - Hidden fields and cookies are unmodified
  - Multiple choice (select/radio) input is one of the allowed option
  - etc.
- Potentially protects more than just SQL injection
The downside of WAG

- Can be challenging to configure it precisely, especially for protecting SQL injection attacks on free-format text input.
- Use heuristics to resolve in practice (does this look like an injection attack or not?)

A new user register to a web portal application

<table>
<thead>
<tr>
<th>Name</th>
<th>user01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email Address</td>
<td><a href="mailto:use01@domain.com">use01@domain.com</a></td>
</tr>
<tr>
<td>Telephone</td>
<td>800-12345678</td>
</tr>
<tr>
<td>Address</td>
<td>115 Admin's Street</td>
</tr>
<tr>
<td>Zip Code</td>
<td>CT1234</td>
</tr>
<tr>
<td>Country</td>
<td>United States</td>
</tr>
</tbody>
</table>

Noted the apostrophe

But should we block this?
CardSystems Attack (June 2005)

- CardSystems was a major credit card processing company
- Put out of business by a SQL injection attack
  - Credit card numbers stored unencrypted
  - Data on 263,000 accounts stolen
  - 43 million identities exposed
Hold Security (2014)

- Uncovered widespread SQL injection attacks from Russia against online websites with user registrations
- Identified theft of username and password hashes from 420,000 websites
  - And this is why you shouldn’t use the same password everywhere
- SQLI remains the most common form of successful online attack against Web servers
Next time

- Problems on the browser
  - Cross site scripting
  - Cross site request forwarding
  - Click jacking, etc