Web Security: Injection Attacks

CS 161: Computer Security
Ruta Jawale and Rafael Dutra

July 29, 2019

Slides credit: Raluca Ada Popa, David Wagner, Dan Boneh
Announcements

• Project 2 deadline extended!
  – Due tomorrow (7/30)
  – Autograder gives partial feedback
• Homework 2 deadline extended!
  – Due this Friday (8/2)
• Midterm 2 is next Monday (8/5)
  – Attend lectures and discussions
What happens if a web server is compromised?

- Steal sensitive data (e.g., data from many users)
- Change server data (e.g., affect users)
- Gateway to enabling attacks on clients
- Impersonation (of users to servers, or vice versa)
- Others
Common Attacks

• SQL Injection
  – Browser sends malicious input to server
  – Bad input checking leads to malicious SQL query

• XSS – Cross-site scripting
  – Attacker inserts client-side script into pages viewed by other users, script runs in the users’ browsers

• CSRF – Cross-site request forgery
  – Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site
Today’s focus: injection attacks
Historical Overview

• 1998: first public discussions of SQL injection

In the Phrack magazine, first magazine for hacking community.


• Hundreds of proposed fixes and solutions
Top 10 Web Vulnerabilities

Learn from the mistakes of others!!!
General Code Injection

- Attacker user provides bad input
- Web server does not check input format
- Enables attacker to execute arbitrary code on the server
PHP Code Injection: Eval

- \$_GET[‘A’]: gets the input with value A from a HTTP GET request

1. User visits calculator and writes 3+5 ENTER
2. User’s browser sends HTTP request
   http://site.com/calc.php?exp="3+5"
3. Script at server receives http request and runs
   \$_GET(“exp”) = “3+5”

- \$_POST[‘B’]: gets the input with value B from a HTTP POST request
PHP Code Injection: Eval

- **eval** allows a web server to evaluate a string as code
  - e.g. `eval('result = 3+5')` produces 8


```
$exp = $_GET['exp'];
eval('$result = $exp');
```

Example: PHP server-side code for sending email

```
$email = $_POST["email"]
$subject = $_POST["subject"]
system("mail $email -s $subject < /tmp/joinmynetwork")
```

Attacker can post

```
http://yourdomain.com/mail.php?
email="hacker@hackerhome.net" &
subject="foo < /usr/passwd; ls"
```
Structured Query Language (SQL)
How is SQL related to the web?
Structure of Modern Web Services

Browser

URL / Form
command.php?arg1=x&arg2=y

Web server

Database server
Structure of Modern Web Services

URL / Form
command.php?arg1=x&arg2=y

Web server

Database query built from x and y

Database server
Structure of Modern Web Services

Browser

Web server

Custom data corresponding to x & y

Database server
Structure of Modern Web Services

Web page built using custom data

Browser

Web server

Custom data corresponding to x & y

Database server
Structure of Modern Web Services

Browser

Web server

Database server
## Databases

- Structured collection of data
  - Often storing tuples/rows of related values
  - Organized in tables

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>35.7</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>79.2</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Databases

• Widely used by web services to store server and user information
• Database runs as separate process to which web server connects
  – Web server sends queries or commands derived from incoming HTTP request
  – Database server returns associated values or modifies/updates values
• SQL is commonly used to manage the database
What are some examples of what SQL can do?
SQL: SELECT

- Widely used database query language
  - (Pronounced “ess-cue-ell” or “sequel”)

- Fetch a set of rows:
  
  \texttt{SELECT column FROM table WHERE condition}
  
  - returns: the value(s) of the given column in the specified table, for all records where condition is true.

- Example:
  
  \texttt{SELECT Balance FROM Customer WHERE Username='bgates'}
  
  - returns: the value 79.2
## SQL: INSERT INTO

- Can add data to the table (or modify):

```
INSERT INTO Customer
VALUES (8477, 'oski', 10.00)
```

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>35.7</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>79.2</td>
</tr>
<tr>
<td>8477</td>
<td>oski</td>
<td>10.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
SQL: DROP TABLE

- Can delete entire tables:

```sql
DROP TABLE Customer
```
SQL: Multiple Queries

• Issue multiple commands, separated by semicolon “;”:

  INSERT INTO Customer VALUES (4433, 'vladimir', 70.0); SELECT AcctNum FROM Customer WHERE Username='vladimir'

  – returns: 4433
SQL: Subquery

• Issue subcommand using parentheses:

```sql
SELECT AcctNum, Balance FROM (SELECT * FROM Customer WHERE Username='vladimir')
```

- subquery runs first and returns a table
- outer query selects specific columns from this table
SQL: Comment

• Comments can be made using “--”
• SQL Parser ignores comments
• `SELECT AcctNum, Balance -- Comment
  FROM Customer`
  – There is a line break between `Balance` and `FROM`
  – `Comment` is commented out
  – This is a valid query
• `SELECT AcctNum, Balance -- Comment FROM Customer`
  – All SQL code is on the same line
  – `Comment FROM Customer` is commented out
  – This is an invalid query
Get familiar with SQL:

https://www.w3schools.com/sql/sql_examples.asp
How does SQL parse its code?
SELECT AcctNum FROM Customer
WHERE Balance < 100 AND Username='Bob'
Break Time: Ryan Lehmkuhl

- San Diego, CA
- Enjoys systems security (Proj 2)
- A cappella group member

- Got lost in the Sahara Desert
- Found staring into the eyes of a kiwi bird IRL life changing
SQL Injection
How can an attacker use SQL?
Hi, this is your son's school. We're having some computer trouble.

Oh, dear - did he break something? In a way...
SQL Injection Scenario #1

- Suppose web server runs the following code:

  ```php
  $recipient = $_POST['recipient'];
  $sql = "SELECT AcctNum FROM Customer WHERE Username='\$recipient';
  $rs = $db->executeQuery($sql);
  ```

- Web server stores URL parameter “recipient” in variable `$recipient`

- Web server sends `$sql` query to database server to get recipient’s account number from database
SQL Injection Scenario #1

- Suppose web server runs the following code:
  
  ```php
  $recipient = $_POST['recipient'];
  $sql = "SELECT AcctNum FROM Customer WHERE Username='\$recipient' ";
  $rs = $db->executeQuery($sql);
  ```

  - Normal use case: If HTTP URL request contains "?recipient=Bob", then the SQL query will be
    
    ```sql
    $sql = " SELECT AcctNum FROM Customer WHERE Username='Bob' "
    ```
**SQL Injection Scenario #1**

1. post malicious form

   $\text{recipient}$ specified by attacker

2. unintended SQL query

3. receive valuable data

**How an attacker use $\text{recipient}$ to cause trouble here?**
$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer
WHERE Username='$recipient' ";
$rs = $db-&gt;executeQuery($sql);

Untrusted user input 'recipient' is embedded directly into SQL command

Attack: $recipient = " alice'; SELECT *
FROM Customer-- "

Returns the entire contents of the Customer!
SQL Injection Scenario #2

```sql
set ok = execute("SELECT * FROM Users WHERE user=' " & form("user") & " ' AND pwd=' " & form("pwd") & " ' "");

if not ok.EOF
    login success
else fail;
```
SQL Injection Scenario #2

Normal Query
set ok = execute("SELECT * FROM Users WHERE user='" & form("user") & "' AND pwd='" & form("pwd") & "'");

if not ok.EOF
    login success
else fail;
SQL Injection Scenario #2

- Suppose user = "' OR 1=1 -- " (URL encoded)
- Then scripts does:
  - `ok = execute("SELECT * FROM Users WHERE user= '" OR 1=1 -- ...")`
  - The "--" causes rest of line to be ignored.
  - Now `ok.EOF` is always false and login succeeds.
- The bad news: easy login to many sites this way.

Besides logging in, what else can attacker do?
SQL Injection Scenario #2

• Suppose user = "' ; DROP TABLE Users--"

• Then script does:

```python
ok = execute("SELECT * FROM Users
WHERE user= ' ; DROP TABLE Users-- ...
")
```
SQL Injection Scenario #2

• Add query to create another account with password, or reset a password
  - `user = "'"; INSERT INTO TABLE Users ('attacker', 'attacker secret')`

• And pretty much everything that can be done by running a query on the DB!
SQL Injection Demo
CardSystems Attack

- CardSystems
  - credit card payment processing company
  - SQL injection attack in June 2005
  - put out of business

- The Attack
  - 263,000 credit card #s stolen from database
  - credit card #s stored unencrypted
  - 43 million credit card #s exposed
The hbgaryfederal.com CMS was susceptible to a kind of attack called SQL injection. In common with other CMSes, the hbgaryfederal.com CMS stores its data in an SQL database, retrieving data from that database with suitable queries. Some queries are fixed—an integral part of the CMS application itself. Others, however, need parameters. For example, a query to retrieve an article from the CMS will generally need a parameter corresponding to the article ID number. These parameters are, in turn, generally passed from the Web frontend to the CMS.

It has been an embarrassing week for security firm HBGary and its HBGary Federal offshoot. HBGary Federal CEO Aaron Barr thought he had unmasked the hacker hordes of Anonymous and was preparing to name and shame those responsible for co-ordinating the group's actions, including the denial-of-service attacks that hit MasterCard, Visa, and other perceived enemies of WikiLeaks late last year.

When Barr told one of those he believed to be an Anonymous ringleader about his forthcoming exposé, the Anonymous response was swift and humiliating. HBGary's servers were broken into, its e-mails pillaged and published to the world, its data destroyed, and its website defaced. As an added bonus, a second site owned
SQL Injection Prevention

- Sanitize user input: check or enforce that value/string that does not have commands of any sort
  - Blacklisting: disallow special characters
  - Whitelisting: only allow certain types of characters
  - Escape input string

SELECT PersonID FROM People WHERE Username=’alice\’; SELECT * FROM People’
SQL Escape Input

You “escape” the SQL parser

Web Server -> query -> Parser -> commands -> DB
SQL Escape Input

- The input string should be interpreted as a string and not as a special character.
- To escape the SQL parser, use backslash in front of special characters, such as quotes or backslashes.
**SQL Parser**

- If it sees ‘ it considers a string is starting or ending
- If it sees \ it considers it just as a character part of a string and converts it to ‘

Example:

```sql
SELECT PersonID FROM People WHERE Username='alice\'; SELECT * FROM People
```

The username will be matched against `alice'; SELECT * FROM People`

and no match will be found

- Different parsers have different escape sequences or API for escaping
What is the string username gets compared to (after SQL parsing), and when does it flag a syntax error? (syntax error appears at least when quotes are not closed)

```sql
[..] WHERE Username='alice'
```

```sql
[..] WHERE Username='alice\'
```

```sql
[..] WHERE Username='alice''
```

```sql
[..] WHERE Username='alice\\'
```

because `\` gets converted to `\` by the parser
SQL Injection Prevention

• Avoid building a SQL command based on raw user input, use existing tools or frameworks

• E.g. (1): the Django web framework has built in sanitization and protection for other common vulnerabilities
  – Django defines a query abstraction layer which sits atop SQL and allows applications to avoid writing raw SQL
  – The execute function takes a sql query and replaces inputs with escaped values

• E.g. (2): Or use parameterized/prepared SQL
SQL Prepared Statement

• Builds SQL queries by properly escaping args: ‘ ’ → \'
  – Example: Parameterized SQL (ASP.NET 1.1)
  – Ensures SQL arguments are properly escaped

SqlCommand cmd = new SqlCommand(
    "SELECT * FROM UserTable WHERE username = @User AND password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request["user"]);

cmd.Parameters.Add("@Pwd", Request["pwd"]);

cmd.ExecuteReader();
Fix structure of SQL parse tree. Only allow user input (?)'s at leaves, not internal nodes.
What happens to the input "Bob"; DROP TABLE Customer --?
General Injection Prevention

Similarly to SQL injections:

- Sanitize input from the user!
- Use frameworks/tools that already check user input
Summary

- Injection attacks were and are the most common web vulnerability

- It is typically due to malicious input supplied by an attacker that is passed without checking into a command; the input contains commands or alters the command

- Can be prevented by sanitizing user input