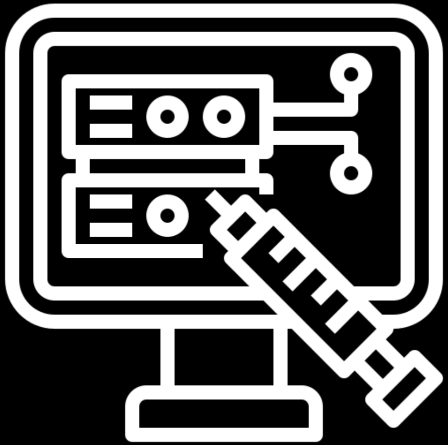


Code Injection Attacks

Lecture 13

Software Security Engineering

Winter 2023
Thompson Rivers University

Server Side of Web Applications

- runs on a web server (application server)
- takes input from remote users via Web server
- interacts with back-end database and other servers
    - side effects: new data stored, functions called
- prepares and outputs results for users
    - dynamically generated HTML
    - content from different sources

Problem: scripting languages allow execution of strings.

Problem: scripting languages allow execution of strings.

Code is Data and Data is Code.

Problem: scripting languages allow execution of strings.

Code is Data and Data is Code.

This is true of C as well: buffer overflow, system()

Problem: scripting languages allow execution of strings.

Code is Data and Data is Code.

This is true of C as well: buffer overflow, system()

But scripting languages makes it easy

e.g., exec(‘a = 4’)

Example: PHP

- PHP: Hypertext Preprocessor (PHP)
- server scripting language, C-like, intermixed with HTML
- e.g., <input value=<?php echo $myvalue; ?>>
- can embed variables in double-quote strings
  - $user=“world”;
  - echo “hello $user”;
  - or echo “hello” . $user;

Command Injection

- server-side PHP calculator
  - $in = USER INPUT VAL
  - eval('$op1 = ' . $in . ';');
- the website only issues HTML calls like
  - http://victim.com/calc.php?val=5
  - it executes: eval('$op1=5;');

But adversary can exhibit arbitrary behaviours!

But adversary can exhibit arbitrary behaviours!

```
http://victim.com/calc.php?val=5 ; system(’rm -rf /’)
```

But adversary can exhibit arbitrary behaviours!

http://victim.com/calc.php?val=5 ; system(’rm -rf /’)

it executes: eval(‘$op1=5; system(’rm -rf /’););

oops!

Another PHP Example

- PHP server-side code for sending email:
  - $email = GET EMAIL
  - system(“mail $email < /tmp/default_email_body”)
- normal call
  - http://victim.com/send_invite/php?email=decent@person.com
- adversarial call
  - http://victim.com/send_invite/php?email=evil@person.com < /usr/passwd; cat
- what happened? why did it happen? how can you stop it?

This is an example of input validation vulnerability

This is an example of input validation vulnerability

Server was expecting a string of a certain form, such as one in the database of users.

This is an example of input validation vulnerability

Server was expecting a string of a certain form, such as one in the database of users.

Assumption string does not have control characters.

This is an example of input validation vulnerability

Server was expecting a string of a certain form, such as one in the database of users.

Assumption string does not have control characters.

Solution is simple: don’t trust any input, and validate all assumptions.

This is an example of input validation vulnerability

Server was expecting a string of a certain form, such as one in the database of users.

Assumption string does not have control characters.

Solution is simple: don’t trust any input,
and validate all assumptions.

Input from users should be treated as hostile.

Structured Query Language (SQL)

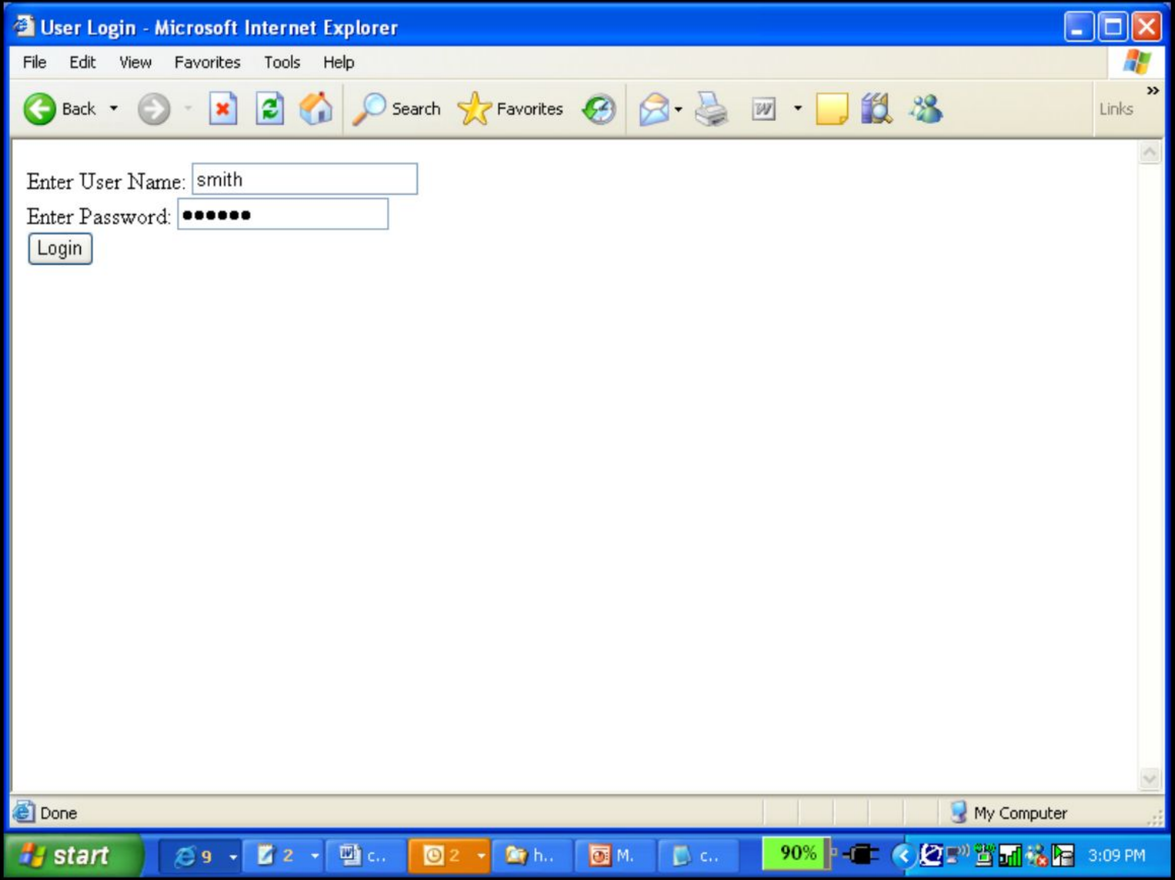
- widely used database query language
- fetch data: SELECT * FROM table WHERE something=‘value’
- add data: INSERT INTO table (col1, col2) VALUES (val1, val2)
- modify, delete, etc.
- syntax is standardized, independent of the database

Typical Query Generation Code

- $selected_user = (get user input)
- $sql_query = “SELECT username, key FROM keys WHERE username=‘$selected_user’ ”;
- $result = $db->executeQuery($sql);

What if ‘user’ is a malicious string that changes the meaning of the query?

Typical Login Prompt

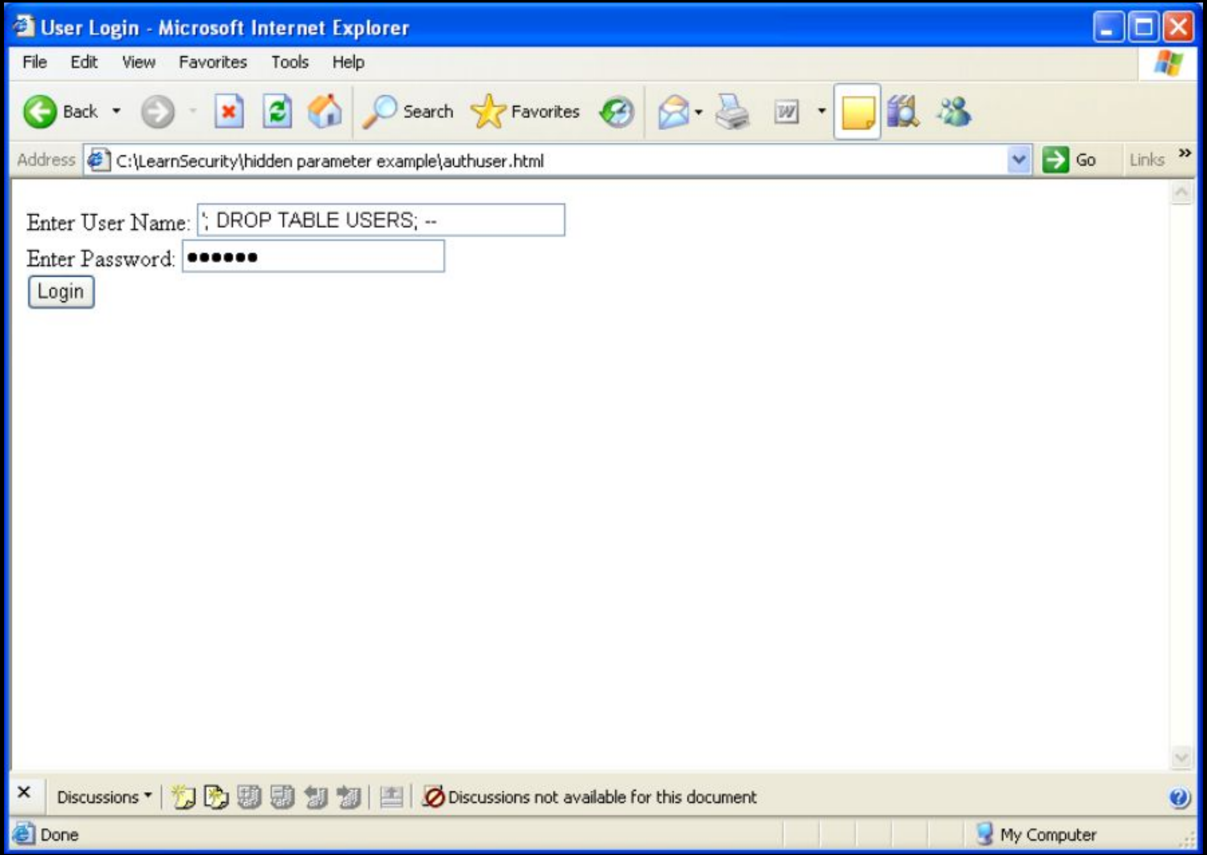


Enter User Name: smith

Enter Password: ●●●●●●●

Browser sends ‘user’,
web server creates SQL,
DB executes SQL

Malicious Login



User Login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
Go
Links
Discussions
Discussions not available for this document
Done
My Computer

C:\LearnSecurity\hidden parameter example\authuser.html

Enter User Name: '; DROP TABLE USERS; --

Enter Password: ●●●●●●●

Login

SQL Injection Attack

- provided input is:
    - ‘foo’; DROP TABLE USERS; ––’
- executed query is
    - SELECT username, key FROM keys WHERE username=foo’; DROP TABLE USERS; ––
- this deletes the table name USERS
- oops.

Authentication to DB

```
set user found = execute(“SELECT * FROM users WHERE username=‘ ” &
form(“user”) & “’ AND password=‘ ” & form(“pwd”) & “’ ”);
if (size(user found) != 0)
return AUTHENTICATE SUCCESS
```

Authentication to DB

```
set user found = execute(“SELECT * FROM users WHERE username=‘ ” &
form(“user”) & “’ AND password=‘ ” & form(“pwd”) & “’ ”);
if (size(user found) != 0)
return AUTHENTICATE SUCCESS
```

user provides username and password,
this query looks up the combination

Authentication to DB

```
set user found = execute(“SELECT * FROM users WHERE username=‘ ” &
form(“user”) & “’ AND password=‘ ” & form(“pwd”) & “’ ”);
if (size(user found) != 0)
return AUTHENTICATE SUCCESS
```

user provides username and password,
this query looks up the combination

if there is one row in user found,
authentication is correct!

Attack on Authentication

- user gives username: ’ OR 1=1 ––
- web server executes SELECT * FROM users WHERE username=“ OR 1=1 –– blahblah
    - now everything matches (why?)
    - user is found (why?)
    - authentication successful (why?)

Another Example

- SELECT * WHERE user=‘name’ AND pwd=‘passwd’
- user gives for both name and passwd:
  - ’OR WHERE pwd LIKE ‘%
- server runs:
  - SELECT * WHERE user=‘’ OR WHERE pwd LIKE ‘%’ AND pwd = ‘’ OR WHERE pwd LIKE ‘%’
  - the % is a wildcard, it matched anything

Result of this:

logs into the database with the credentials of the first person in DB

Result of this:

logs into the database with the credentials of the first person in DB

this is usually the administrator!

Result of this:

logs into the database with the credentials of the first person in DB

this is usually the administrator!

PRIVILEGE ESCALATION

Pull Data from other Database

username: ’ AND 1 = 0 UNION SELECT cardholder, number, exp_month, exp_year FROM creditcards

Pull Data from other Database

username: ’ AND 1 = 0 UNION SELECT cardholder, number, exp_month, exp_year FROM creditcards

results of both queries are combined and returned

Create User

username: ’; INSERT INTO USERS (...) VALUES (...);

Create User

username: ’; INSERT INTO USERS (...) VALUES (...);

WHERE email=victim@tru.ca

Second-Order SQL Injection

- code as data can be stored now but executed later
  - inconsistency in checking
- user sets username to: admin’ ––
  - suppose that DB builds the query correctly
  - the quote in the username does not terminate the query but the username is set as above
    - i.e., it is properly escaped at the time
- user then changes their password
  - perhaps not through a web frontend
  - UPDATE USERS SET passwd=‘evil’ WHERE uname=‘admin’ ––’

Preventing SQL Injection

- validate all inputs
  - filter out any character that has special meaning
    - apostrophes, semicolons, percents, hyphens, underscores
  - check the data type
    - all assumptions must be checked
  - use libraries designed to do this instead of doing it yourself
- FULL MEDIATION

Preventing SQL Injection

- allow list permitted characters
    - block listing bad ones doesn’t work
    - safe defaults
    - set well-defined set of safe values
    - match with regular expressions

Escaping Quotes

- special characters like ’ blur code and data
- but can occur in names: O’Brian
- these must be escaped in the input
  - functions to do this: escape(o’connor) → o\’connor
  - don’t do this ad hoc
  - don’t just replace ’ with \’ (why?)

Prepared Statements

- SQL injection comes about because queries are created by string concatenations
- this elevates user-provided input to the importance level of backend code written by trusted engineers
  - both strings are equal components to the resulting query
  - both strings can be data or code
  - user-provided input should be only data, not code

Prepared Statements

- bind variables
  - placeholders guaranteed to be data
- prepared statements
  - static scaffolds of SQL with bind variables to be filled in

Prepared Statements Example (pseudo syntax)

- String query = “SELECT * FROM table WHERE userid=?”;
- PreparedStatement ps = db.prepareStatement(query);
- ps.setInt(1, session.getCurrentUserId());
- ResultSet = ps.executeQuery();