SQL Injection

Web Application Security

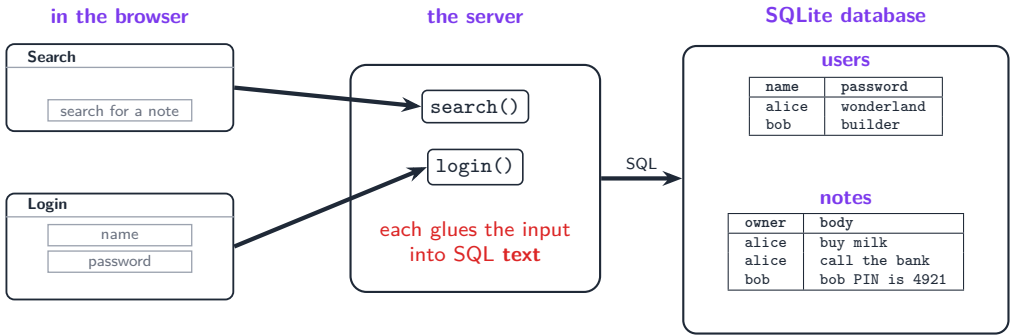
Bernard Nongpoh

Assistant Professor, Department of CSE

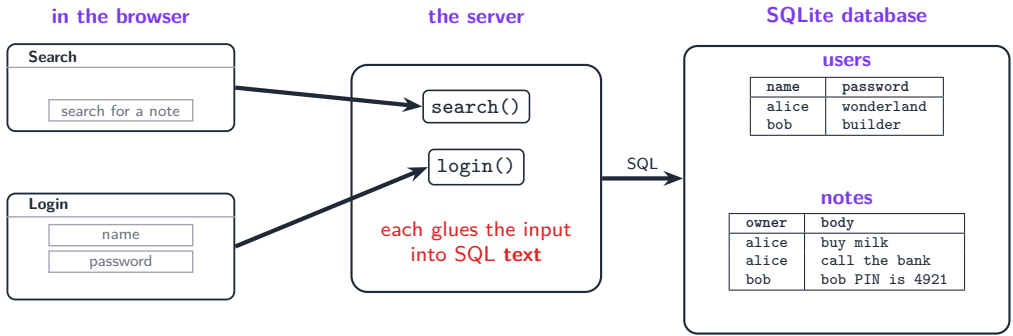


ভাৰতীয় প্ৰযুক্তিবিদ্যা প্ৰতিষ্ঠান গুৱাহাটী
भारतीय प्रौद्योगिकी संस्थान गुवाहाटी
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

The Example Application (1/2)

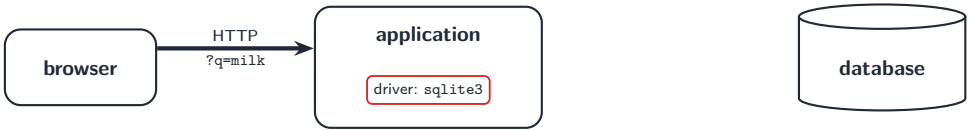


The Example Application (2/2)



Two doors for user input: the **search box** and the **login form**. Both reach the database as text.

Architecture (1/5)



Architecture (2/5)



Architecture (3/5)



Architecture (4/5)



Architecture (5/5)



SQL in Brief (1/3)

How does an application ask its database for data?

table `users`

name	password
alice	wonderland
bob	builder

table `notes`

owner	body
alice	buy milk
alice	call the bank
bob	bob PIN is 4921

SQL in Brief (2/3)

How does an application ask its database for data?

table users

name	password
alice	wonderland
bob	builder

table notes

owner	body
alice	buy milk
alice	call the bank
bob	bob PIN is 4921

a query

```
SELECT body FROM notes
WHERE owner = 'alice'
```

result: **buy milk**, **call the bank**

SQL in Brief (3/3)

How does an application ask its database for data?

table users

name	password
alice	wonderland
bob	builder

table notes

owner	body
alice	buy milk
alice	call the bank
bob	bob PIN is 4921

a query

```
SELECT body FROM notes
WHERE owner = 'alice'
```

result: **buy milk**, **call the bank**

three more pieces of SQL

`'alice'`	quotes start and end a piece of text
`-- anything`	a comment: the rest of the line is ignored
`... UNION SELECT ...`	adds the rows of a second query

Building Queries in Code (1/3)

The query depends on the input. How is the text put together?

```
a = "... name = '" + name + "'"
b = "... name = '%s'" % name
c = "... name = '{}'".format(name)
d = f"... name = '{name}'"
```

concatenation with **+**

the **%** operator

.format()

an **f-string**

Building Queries in Code (2/3)

The query depends on the input. How is the text put together?

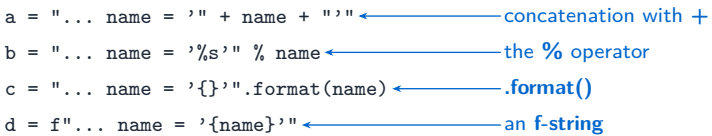
```
a = "... name = '" + name + "'"      ← concatenation with +
b = "... name = '%s'" % name         ← the % operator
c = "... name = '{}'".format(name)   ← .format()
d = f"... name = '{name}'"           ← an f-string
```

the example data

```
name = alice
  all four the same: True
  query text: SELECT * FROM users WHERE name = 'alice'
  database: ok
```

Building Queries in Code (3/3)

The query depends on the input. How is the text put together?



the example data

```
name = alice
  all four the same: True
  query text: SELECT * FROM users WHERE name = 'alice'
  database: ok
```

```
name = O'Brien
  all four the same: True
  query text: SELECT * FROM users WHERE name = 'O'Brien'
  database: near "Brien": syntax error
```

SQL Injection (1/6)

Your text goes into a query. What if the text is SQL?

```
42 def search(owner, q):
43     sql = ("SELECT body FROM notes"
44            " WHERE owner = '" + owner + "'"
45            " AND body LIKE '%" + q + "%'")
```

rain.forest.puppy, Phrack 54, 1998; Stanford CS253

SQL Injection (2/6)

Your text goes into a query. What if the text is SQL?

```
42 def search(owner, q):
43     sql = ("SELECT body FROM notes"
44            " WHERE owner = '" + owner + "'"
45            " AND body LIKE '%" + q + "%'")
```

`q = milk`

```
SELECT ... WHERE owner = 'alice' AND body LIKE '%milk%'
```

rain.forest.puppy, Phrack 54, 1998; Stanford CS253

SQL Injection (3/6)

Your text goes into a query. What if the text is SQL?

```
42 def search(owner, q):
43     sql = ("SELECT body FROM notes"
44            " WHERE owner = '" + owner + "'"
45            " AND body LIKE '%" + q + "%'")
```

`q = milk` `SELECT ... WHERE owner = 'alice' AND body LIKE '%milk%'`

`q = ' OR 1=1--` `SELECT ... WHERE owner = 'alice' AND body LIKE '%' OR 1=1--%'`

rain.forest.puppy, Phrack 54, 1998; Stanford CS253

SQL Injection (4/6)

Your text goes into a query. What if the text is SQL?

```
42 def search(owner, q):
43     sql = ("SELECT body FROM notes"
44            " WHERE owner = '" + owner + "'"
45            " AND body LIKE '%" + q + "%'")
```

`q = milk` `SELECT ... WHERE owner = 'alice' AND body LIKE '%milk%'`

`q = ' OR 1=1--` `SELECT ... WHERE owner = 'alice' AND body LIKE '%' OR 1=1--%'`

ends the text

rain.forest.puppy, Phrack 54, 1998; Stanford CS253

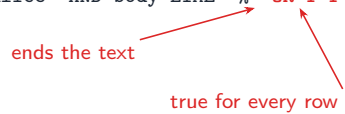
SQL Injection (5/6)

Your text goes into a query. What if the text is SQL?

```
42 def search(owner, q):
43     sql = ("SELECT body FROM notes"
44            " WHERE owner = '" + owner + "'"
45            " AND body LIKE '%" + q + "%'")
```

`q = milk` `SELECT ... WHERE owner = 'alice' AND body LIKE '%milk%'`

`q = ' OR 1=1--` `SELECT ... WHERE owner = 'alice' AND body LIKE '%' OR 1=1--%'`



rain.forest.puppy, Phrack 54, 1998; Stanford CS253

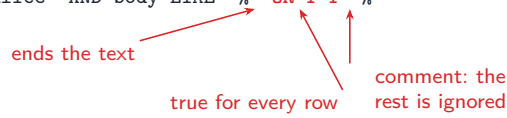
SQL Injection (6/6)

Your text goes into a query. What if the text is SQL?

```
42 def search(owner, q):
43     sql = ("SELECT body FROM notes"
44            " WHERE owner = '" + owner + "'"
45            " AND body LIKE '%" + q + "%'")
```

`q = milk` `SELECT ... WHERE owner = 'alice' AND body LIKE '%milk%'`

`q = ' OR 1=1--` `SELECT ... WHERE owner = 'alice' AND body LIKE '%' OR 1=1--%'`

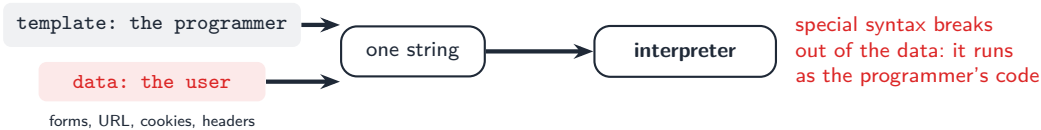


rain.forest.puppy, Phrack 54, 1998; Stanford CS253

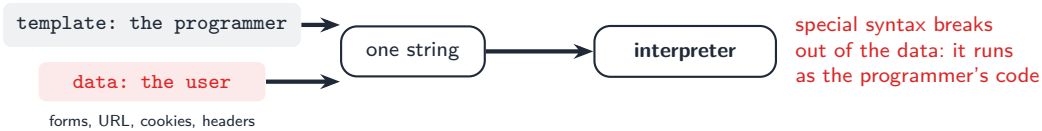
Code Injection (1/6)



Code Injection (2/6)

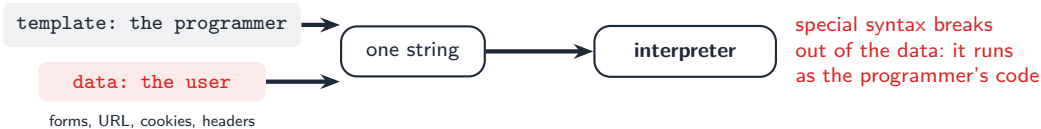


Code Injection (3/6)



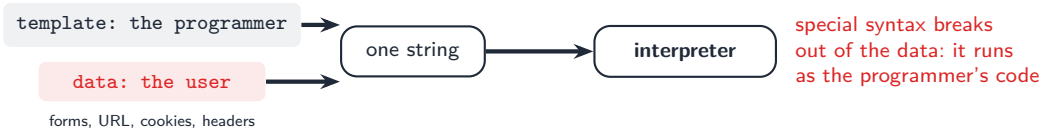
interpreter	unsafe: glue into a string	safe: data passed apart	attack
OS shell	`execSync(‘cat ${filename}‘)`	`spawnSync('cat', [filename])`	command injection

Code Injection (4/6)



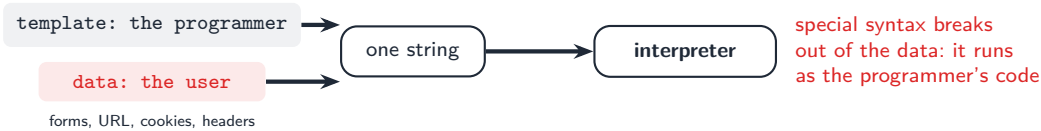
interpreter	unsafe: glue into a string	safe: data passed apart	attack
OS shell	`execSync(‘cat ${filename}‘)`	`spawnSync('cat', [filename])`	command injection
database	`"... WHERE name = '" + name + "'"`	`"... WHERE name = ?", (name,)`	SQL injection

Code Injection (5/6)



interpreter	unsafe: glue into a string	safe: data passed apart	attack
OS shell	`execSync(‘cat ${filename}‘)`	`spawnSync('cat', [filename])`	command injection
database	`"... WHERE name = '" + name + "'"`	`"... WHERE name = ?", (name,)`	SQL injection
browser	`"<p>" + q + "</p>"`	`"<p>" + escape(q) + "</p>"`	XSS

Code Injection (6/6)



interpreter	unsafe: glue into a string	safe: data passed apart	attack
OS shell	`execSync(‘cat ${filename}‘)`	`spawnSync('cat', [filename])`	command injection
database	`"... WHERE name = '" + name + "'"`	`"... WHERE name = ?", (name,)`	SQL injection
browser	`"<p>" + q + "</p>"`	`"<p>" + escape(q) + "</p>"`	XSS

Rule: never build code from strings; pass the data **separately**.

Root Causes of SQL Injection (1/5)

Why does this keep happening?

1. Code and data travel in **one string**

PortSwigger Web Security Academy: SQL injection; OWASP SQL Injection Prevention Cheat Sheet

Root Causes of SQL Injection (2/5)

Why does this keep happening?

1. Code and data travel in **one string**
2. Input is **trusted**: URL, form, cookie, header, *even the database*

PortSwigger Web Security Academy: SQL injection; OWASP SQL Injection Prevention Cheat Sheet

Root Causes of SQL Injection (3/5)

Why does this keep happening?

1. Code and data travel in **one string**
2. Input is **trusted**: URL, form, cookie, header, *even the database*
3. Fixes by hand: **escaping** or **blocklists**

PortSwigger Web Security Academy: SQL injection; OWASP SQL Injection Prevention Cheat Sheet

Root Causes of SQL Injection (4/5)

Why does this keep happening?

1. Code and data travel in **one string**
2. Input is **trusted**: URL, form, cookie, header, *even the database*
3. Fixes by hand: **escaping** or **blocklists**
4. Bigger damage if the database login **has full rights**, or errors **show the query**

PortSwigger Web Security Academy: SQL injection; OWASP SQL Injection Prevention Cheat Sheet

Root Causes of SQL Injection (5/5)

Why does this keep happening?

1. Code and data travel in **one string**
2. Input is **trusted**: URL, form, cookie, header, *even the database*
3. Fixes by hand: **escaping** or **blocklists**
4. Bigger damage if the database login **has full rights**, or errors **show the query**

Where input can land in a query:

`SELECT`	`WHERE` clause, most often
`INSERT`	the inserted values
`UPDATE`	new values, `WHERE`
`SELECT`	table, column, `ORDER BY`

PortSwigger Web Security Academy: SQL injection; OWASP SQL Injection Prevention Cheat Sheet

How UNION Works (1/4)

`UNION` stacks the rows of a *second* query under the first.

```
SELECT body FROM notes WHERE ...  UNION SELECT password FROM users
```

the app's query — the attacker adds this

PortSwigger: SQL injection UNION attacks

How UNION Works (2/4)

`UNION` stacks the rows of a *second* query under the first.

```
SELECT body FROM notes WHERE ...  UNION SELECT password FROM users
```

the app's query — the attacker adds this

rows from `notes`

body
(no match)

+

rows from `users`

password
builder
wonderland

How UNION Works (3/4)

`UNION` stacks the rows of a *second* query under the first.

```
SELECT body FROM notes WHERE ...  UNION SELECT password FROM users
```

the app's query

the attacker adds this

rows from `notes`

body
(no match)

+

rows from `users`

password
builder
wonderland

=

what the page shows

body
builder
wonderland

passwords, shown as "notes"

How UNION Works (4/4)

`UNION` stacks the rows of a *second* query under the first.

```
SELECT body FROM notes WHERE ...  UNION SELECT password FROM users
```

the app's query — the attacker adds this

rows from `notes`

body
(no match)

+

rows from `users`

password
builder
wonderland

=

what the page shows

body
builder
wonderland

passwords, shown as "notes"

Two rules: the **same number of columns** in both queries, and **matching types**. So step one of a UNION attack is to find the column count.

PortSwigger: SQL injection UNION attacks

Logging In Without a Password (1/4)

The login builds its query the same way.

```
29 def login(name, password):
30     sql = ("SELECT name FROM users"
31            " WHERE name = '" + name + "'"
32            " AND password = '" + password + "'")
```

Logging In Without a Password (2/4)

The login builds its query the same way.

```
29 def login(name, password):
30     sql = ("SELECT name FROM users"
31            " WHERE name = '" + name + "'"
32            " AND password = '" + password + "'")
```

`name = bob'--, pw = x`

```
SELECT name FROM users WHERE name = 'bob'--' AND password = 'x'
```

Logging In Without a Password (3/4)

The login builds its query the same way.

```
29 def login(name, password):
30     sql = ("SELECT name FROM users"
31            " WHERE name = '" + name + "'"
32            " AND password = '" + password + "'")
```

name = bob'--, pw = x

```
SELECT name FROM users WHERE name = 'bob'--' AND password = 'x'
```

← comment: the password check is gone

Logging In Without a Password (4/4)

The login builds its query the same way.

```
29 def login(name, password):
30     sql = ("SELECT name FROM users"
31            " WHERE name = '" + name + "'"
32            " AND password = '" + password + "'")
```

name = bob'--, pw = x

```
SELECT name FROM users WHERE name = 'bob'--' AND password = 'x'
```

← comment: the password check is gone

```
login form: name = bob'--, password = x
the page shows: welcome bob
```

output

Quiz: Who Gets In? (1/4)

Same login query. Who is logged in?

1. `name = bob` `pw = ' OR '1'='1` ☐
2. `name = bob'#` `pw = x` ☐
3. `name = x' OR name='bob'--` `pw = x` ☐

Quiz: Who Gets In? (2/4)

Same login query. Who is logged in?

1.	`name` = `bob`	`pw` = `' OR '1'='1`	alice
2.	`name` = `bob'#`	`pw` = `x`	
3.	`name` = `x' OR name='bob'--`	`pw` = `x`	

Quiz: Who Gets In? (3/4)

Same login query. Who is logged in?

1.	`name = bob`	`pw = ' OR '1'='1`	alice
2.	`name = bob'#`	`pw = x`	nobody
3.	`name = x' OR name='bob'--`	`pw = x`	

Quiz: Who Gets In? (4/4)

Same login query. Who is logged in?

1.	`name = bob`	`pw = ' OR '1'='1`	alice
2.	`name = bob'#`	`pw = x`	nobody
3.	`name = x' OR name='bob'--`	`pw = x`	bob

How to Detect SQL Injection (1/3)

No source code? Send a crafted input and watch *how the answer changes*.

1. a quote `'` → **error** (reached the SQL)
2. `OR 1=1` vs `OR 1=2` → **differ** (ran as SQL)
3. a **time delay** → slower answer
4. **out-of-band** (OAST) → the DB calls your server

vulnerable app: page says yes / no

```
'             -> error (the query broke)
x' OR 1=1--   -> yes
x' OR 1=2--   -> no     answers DIFFER
```

PortSwigger: How to detect SQL injection

How to Detect SQL Injection (2/3)

No source code? Send a crafted input and watch *how the answer changes*.

1. a quote `'` → **error** (reached the SQL)
2. `OR 1=1` vs `OR 1=2` → **differ** (ran as SQL)
3. a **time delay** → slower answer
4. **out-of-band** (OAST) → the DB calls your server

vulnerable app: page says yes / no

```
'             -> error (the query broke)
x' OR 1=1--   -> yes
x' OR 1=2--   -> no     answers DIFFER
```

fixed app (parameterized)

```
'             -> no
x' OR 1=1--   -> no
x' OR 1=2--   -> no     answers the SAME
```

PortSwigger: How to detect SQL injection

How to Detect SQL Injection (3/3)

No source code? Send a crafted input and watch *how the answer changes*.

1. a quote `'` → **error** (reached the SQL)
2. `OR 1=1` vs `OR 1=2` → **differ** (ran as SQL)
3. a **time delay** → slower answer
4. **out-of-band** (OAST) → the DB calls your server

vulnerable app: page says yes / no

```
'             -> error (the query broke)
x' OR 1=1--   -> yes
x' OR 1=2--   -> no     answers DIFFER
```

fixed app (parameterized)

```
'             -> no
x' OR 1=1--   -> no
x' OR 1=2--   -> no     answers the SAME
```

Answer changes with the input ⇒ injectable. **Answer never changes** ⇒ safe. Test every input; only on systems you may test.

PortSwigger: How to detect SQL injection

What Error Messages Reveal (1/3)

The single quote broke the query. What does the error page say?

the error message

```
--- debug error page (development setting) ---
sqlite3.OperationalError: unrecognized token: "'"
query: SELECT body FROM notes WHERE owner = 'alice' AND body LIKE '%'%'
```

PortSwigger: Blind SQL injection, error-based

What Error Messages Reveal (2/3)

The single quote broke the query. What does the error page say?

the error message

```
--- debug error page (development setting) ---
sqlite3.OperationalError: unrecognized token: "'"
query: SELECT body FROM notes WHERE owner = 'alice' AND body LIKE '%'%'
```

- The **database product**, the **query**, and where the input lands
- Some databases even put **data values** into the message: **error-based** SQL injection

PortSwigger: Blind SQL injection, error-based

What Error Messages Reveal (3/3)

The single quote broke the query. What does the error page say?

the error message

```
--- debug error page (development setting) ---
sqlite3.OperationalError: unrecognized token: "'"
query: SELECT body FROM notes WHERE owner = 'alice' AND body LIKE '%'%'
```

```
--- plain error page (production setting) ---
Something went wrong. Reference: 7f3a
```

- The **database product**, the **query**, and where the input lands
- Some databases even put **data values** into the message: **error-based** SQL injection
- Fix: a plain page for the user, the details in the **server log**

PortSwigger: Blind SQL injection, error-based

Blind SQL Injection (1/4)

The page only says yes or no. Is the data safe?

what the page /exists answers

```
GET /exists?q=milk   ->  yes
GET /exists?q=zebra  ->  no
```

Blind SQL Injection (2/4)

The page only says yes or no. Is the data safe?

what the page /exists answers

```
GET /exists?q=milk   ->  yes
GET /exists?q=zebra  ->  no
```

as the database received it

```
SQL: SELECT body FROM notes WHERE owner = '
      alice' AND body LIKE '%x' OR (SELECT
      substr(password,1,1) FROM users WHERE
      name='bob')='b'--%'
```

Blind SQL Injection (3/4)

The page only says yes or no. Is the data safe?

what the page /exists answers

```
GET /exists?q=milk   ->  yes
GET /exists?q=zebra  ->  no
```

as the database received it

```
SQL: SELECT body FROM notes WHERE owner = '
      alice' AND body LIKE '%x' OR (SELECT
      substr(password,1,1) FROM users WHERE
      name='bob')='b'--%'
```

output (shortened)

```
$ ask "(SELECT substr(password,1,1) ...)='a'"
no
$ ask "(SELECT substr(password,1,1) ...)='b'"
yes
```

Blind SQL Injection (4/4)

The page only says yes or no. Is the data safe?

what the page /exists answers

```
GET /exists?q=milk   ->  yes
GET /exists?q=zebra  ->  no
```

as the database received it

```
SQL: SELECT body FROM notes WHERE owner = '
     alice' AND body LIKE '%x' OR (SELECT
     substr(password,1,1) FROM users WHERE
     name='bob')='b'--%'
```

output (shortened)

```
$ ask "(SELECT substr(password,1,1) ...)='a'"
no
$ ask "(SELECT substr(password,1,1) ...)='b'"
yes
```

```
letter 1: b   so far: b
letter 2: u   so far: bu
...
letter 7: r   so far: builder
bob's password: builder   (97 questions)
```

Impact and Data Exposure (1/6)

What can one injectable input cost?

read other users' data, passwords, card details

PortSwigger: SQL injection; OWASP Top 10:2025, A05 Injection

Impact and Data Exposure (2/6)

What can one injectable input cost?

read	other users' data, passwords, card details
change or delete	`UPDATE`, `DELETE`: wrong balances, lost rows

PortSwigger: SQL injection; OWASP Top 10:2025, A05 Injection

Impact and Data Exposure (3/6)

What can one injectable input cost?

read	other users' data, passwords, card details
change or delete	`UPDATE`, `DELETE`: wrong balances, lost rows
log in as anyone	the login bypass we saw

PortSwigger: SQL injection; OWASP Top 10:2025, A05 Injection

Impact and Data Exposure (4/6)

What can one injectable input cost?

read	other users' data, passwords, card details
change or delete	`UPDATE`, `DELETE`: wrong balances, lost rows
log in as anyone	the login bypass we saw
take over the server	files, commands (MOVEit: a web shell)

PortSwigger: SQL injection; OWASP Top 10:2025, A05 Injection

Impact and Data Exposure (5/6)

What can one injectable input cost?

read	other users' data, passwords, card details
change or delete	`UPDATE`, `DELETE`: wrong balances, lost rows
log in as anyone	the login bypass we saw
take over the server	files, commands (MOVEit: a web shell)
stay unnoticed	a backdoor, a long-term compromise

PortSwigger: SQL injection; OWASP Top 10:2025, A05 Injection

Impact and Data Exposure (6/6)

What can one injectable input cost?

read	other users' data, passwords, card details
change or delete	`UPDATE`, `DELETE`: wrong balances, lost rows
log in as anyone	the login bypass we saw
take over the server	files, commands (MOVEit: a web shell)
stay unnoticed	a backdoor, a long-term compromise

The limit is the **database account**, not the page that has the bug.

PortSwigger: SQL injection; OWASP Top 10:2025, A05 Injection

Parameterized Queries (1/3)

Keep the code and the data apart.

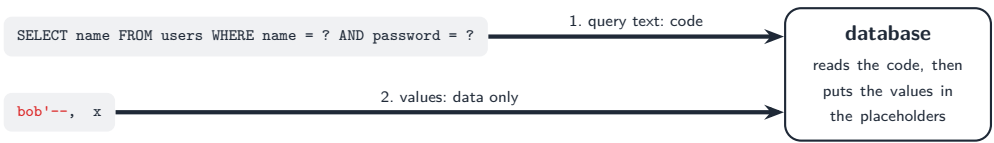
```
36 def login_safe(name, password):
37     sql = ("SELECT name FROM users"
38            " WHERE name = ? AND password = ?")
39     return run(sql, (name, password))
```

OWASP SQL Injection Prevention Cheat Sheet; Stanford CS253

Parameterized Queries (2/3)

Keep the code and the data apart.

```
36 def login_safe(name, password):
37     sql = ("SELECT name FROM users"
38            " WHERE name = ? AND password = ?")
39     return run(sql, (name, password))
```

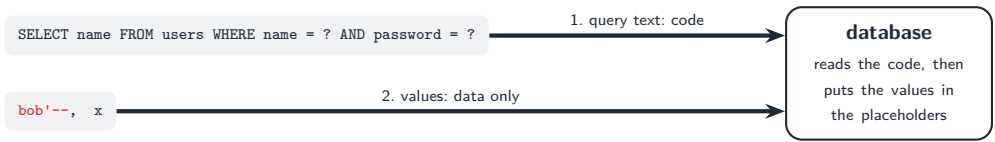


OWASP SQL Injection Prevention Cheat Sheet; Stanford CS253

Parameterized Queries (3/3)

Keep the code and the data apart.

```
36 def login_safe(name, password):
37     sql = ("SELECT name FROM users"
38            " WHERE name = ? AND password = ?")
39     return run(sql, (name, password))
```



OWASP SQL Injection Prevention Cheat Sheet; Stanford CS253

What Cannot Be a Parameter (1/4)

A user may choose the sort order. Can ? hold a column name?

sorting by a column name

```
6 col = "body"  # from the URL: ?sort=body
7 sql = "SELECT body FROM notes ORDER BY ?"
8 print(db.execute(sql, (col,)).fetchall())
```

OWASP SQL Injection Prevention Cheat Sheet

What Cannot Be a Parameter (2/4)

A user may choose the sort order. Can ? hold a column name?

sorting by a column name

```
6 col = "body"  # from the URL: ?sort=body
7 sql = "SELECT body FROM notes ORDER BY ?"
8 print(db.execute(sql, (col,)).fetchall())
```

output

```
$ run it
[('zoo',), ('milk',)]
```

OWASP SQL Injection Prevention Cheat Sheet

What Cannot Be a Parameter (3/4)

A user may choose the sort order. Can ? hold a column name?

sorting by a column name

```
col = "body"  # from the URL: ?sort=body
sql = "SELECT body FROM notes ORDER BY ?"
print(db.execute(sql, (col,)).fetchall())
```

```
SORTABLE = {"body", "owner"}  # an allow-list
col = col if col in SORTABLE else "body"
sql = "SELECT body FROM notes ORDER BY " + col
print(db.execute(sql).fetchall())
```

output

```
$ run it
[('zoo',), ('milk',)]
```

```
[('milk',), ('zoo',)]
```

OWASP SQL Injection Prevention Cheat Sheet

What Cannot Be a Parameter (4/4)

A user may choose the sort order. Can ? hold a column name?

sorting by a column name

```
col = "body"  # from the URL: ?sort=body
sql = "SELECT body FROM notes ORDER BY ?"
print(db.execute(sql, (col,)).fetchall())
```

```
SORTABLE = {"body", "owner"}  # an allow-list
col = col if col in SORTABLE else "body"
sql = "SELECT body FROM notes ORDER BY " + col
print(db.execute(sql).fetchall())
```

output

```
$ run it
[('zoo',), ('milk',)]
```

```
[('milk',), ('zoo',)]
```

Also: give the database account **only the rights it needs**.

OWASP SQL Injection Prevention Cheat Sheet

SQL Injection in Juice Shop's Login (1/3)



A real, popular codebase. Can you spot the mistake?

routes/login.ts, OWASP Juice Shop v20.4

```
34 models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.email || ''}' AND
       password = '${security.hash(req.body.password || '')}' AND deletedAt IS NULL`, { model
       : UserModel, plain: true })
```

Juice Shop: routes/login.ts, challenge *Login Admin*

SQL Injection in Juice Shop's Login (2/3)



A real, popular codebase. Can you spot the mistake?

routes/login.ts, OWASP Juice Shop v20.4

```
34 models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.email || ''}' AND
       password = '${security.hash(req.body.password || '')}' AND deletedAt IS NULL`, { model
       : UserModel, plain: true })
```

our example application

```
30      sql = ("SELECT name FROM users"
31             " WHERE name = '" + name + "'"
32             " AND password = '" + password + "'")
```

Juice Shop: routes/login.ts, challenge *Login Admin*

SQL Injection in Juice Shop's Login (3/3)



A real, popular codebase. Can you spot the mistake?

routes/login.ts, OWASP Juice Shop v20.4

```
34 models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.email || ''}' AND
       password = '${security.hash(req.body.password || '')}' AND deletedAt IS NULL`, { model
       : UserModel, plain: true })
```

our example application

```
30     sql = ("SELECT name FROM users"
31            " WHERE name = '" + name + "'"
32            " AND password = '" + password + "'")
```

`${...}` in a JavaScript template string = `+` in Python: the input is **glued into the text**.

Juice Shop: routes/login.ts, challenge *Login Admin*

Lab: Logging In Without a Password (1/3)



Your turn, on your own Juice Shop. Put the payload in the *email* field; the password can be anything.

Challenge	**Email field**	**You become**
Login Admin (2⋆)	`' OR 1=1--`	admin

Juice Shop challenges *Login Admin*, *Login Jim*, *Database Schema*

Lab: Logging In Without a Password (2/3)



Your turn, on your own Juice Shop. Put the payload in the *email* field; the password can be anything.

Challenge	Email field	You become
Login Admin (2⋆)	`' OR 1=1--`	admin
Login Jim (3⋆)	`jim@juice-sh.op'--`	Jim
Login Bender (3⋆)	`bender@juice-sh.op'--`	Bender

Juice Shop challenges *Login Admin*, *Login Jim*, *Database Schema*

Lab: Logging In Without a Password (3/3)



Your turn, on your own Juice Shop. Put the payload in the *email* field; the password can be anything.

Challenge	Email field	You become
Login Admin (2⋆)	`' OR 1=1--`	admin
Login Jim (3⋆)	`jim@juice-sh.op'--`	Jim
Login Bender (3⋆)	`bender@juice-sh.op'--`	Bender

Stretch: in the **address bar**, not the search box. Open `localhost:3000/rest/products/search?q=`

```
qwert')) UNION SELECT sql,2,3,4,5,6,7,8,9,10 FROM sqlite_master--
```

dumps every table: *Database Schema*. The search box mangles it; the `/rest` API does not.

Juice Shop challenges *Login Admin*, *Login Jim*, *Database Schema*

Input Validation versus Parameterization (1/3)

input validation

```
 7 def valid_name(s):              # allow letters only
 8     return re.fullmatch(r"[A-Za-z]+", s) is not None
 9
10 def find(name):                 # parameterized
11     return db.execute("SELECT name FROM users WHERE name = ?", (name,)).fetchall()
```

OWASP SQL Injection Prevention Cheat Sheet; OWASP Input Validation Cheat Sheet

Input Validation versus Parameterization (2/3)

input validation

```
 7 def valid_name(s):              # allow letters only
 8     return re.fullmatch(r"[A-Za-z]+", s) is not None
 9
10 def find(name):                 # parameterized
11     return db.execute("SELECT name FROM users WHERE name = ?", (name,)).fetchall()
```

output

```
input           valid?  found
alice           True    [('alice',)]
O'Brien         False   [("O'Brien",)]
x' OR '1'='1    False   []
```

OWASP SQL Injection Prevention Cheat Sheet; OWASP Input Validation Cheat Sheet

Input Validation versus Parameterization (3/3)

input validation

```python
def valid_name(s):                # allow letters only
    return re.fullmatch(r"[A-Za-z]+", s) is not None

def find(name):                   # parameterized
    return db.execute("SELECT name FROM users WHERE name = ?", (name,)).fetchall()
```

output

```
input           valid?  found
alice           True    [('alice',)]
O'Brien         False   [("O'Brien",)]
x' OR '1'='1    False   []
```

	validation	parameterization
checks	the input's shape	nothing: it stays data
free text	cannot be restricted	works
role	**second layer**	**the defence**

OWASP SQL Injection Prevention Cheat Sheet; OWASP Input Validation Cheat Sheet

Quiz: Is This Query Safe? (1/5)

`name`, `q` and `col` come from the request.

1. `run(f"SELECT * FROM users WHERE name = '{name}'")`

2. `run("SELECT body FROM notes WHERE body LIKE ?", ("%"+q+"%",))`

3. `run("SELECT body FROM notes ORDER BY " + col)`

4. `s.scalars(select(User).where(User.name == name))`

OWASP SQL Injection Prevention Cheat Sheet

Quiz: Is This Query Safe? (2/5)

`name`, `q` and `col` come from the request.

1. `run(f"SELECT * FROM users WHERE name = '{name}'")`

   NO the f-string glues `name` into the query text

2. `run("SELECT body FROM notes WHERE body LIKE ?", ("%"+q+"%",))`

3. `run("SELECT body FROM notes ORDER BY " + col)`

4. `s.scalars(select(User).where(User.name == name))`

Quiz: Is This Query Safe? (3/5)

`name`, `q` and `col` come from the request.

1. `run(f"SELECT * FROM users WHERE name = '{name}'")`
   NO the f-string glues `name` into the query text
2. `run("SELECT body FROM notes WHERE body LIKE ?", ("%"+q+"%",))`
   YES `%` is added to the *value*, not the query text
3. `run("SELECT body FROM notes ORDER BY " + col)`
4. `s.scalars(select(User).where(User.name == name))`

OWASP SQL Injection Prevention Cheat Sheet

Quiz: Is This Query Safe? (4/5)

`name`, `q` and `col` come from the request.

1. `run(f"SELECT * FROM users WHERE name = '{name}'")`
   NO the f-string glues `name` into the query text
2. `run("SELECT body FROM notes WHERE body LIKE ?", ("%"+q+"%",))`
   YES `%` is added to the *value*, not the query text
3. `run("SELECT body FROM notes ORDER BY " + col)`
   NO `col` is glued into the text; use an allow-list
4. `s.scalars(select(User).where(User.name == name))`

OWASP SQL Injection Prevention Cheat Sheet

Quiz: Is This Query Safe? (5/5)

`name`, `q` and `col` come from the request.

1. `run(f"SELECT * FROM users WHERE name = '{name}'")`
   NO the f-string glues `name` into the query text
2. `run("SELECT body FROM notes WHERE body LIKE ?", ("%"+q+"%",))`
   YES `%` is added to the *value*, not the query text
3. `run("SELECT body FROM notes ORDER BY " + col)`
   NO `col` is glued into the text; use an allow-list
4. `s.scalars(select(User).where(User.name == name))`
   YES the ORM sends `name` as a parameter

OWASP SQL Injection Prevention Cheat Sheet

Summary (1/2)

Attack	What the attacker gets
`' OR 1=1--`	every row: all users' notes
`UNION SELECT ...`	any table: the passwords
`bob'--` in the login	a session as bob
error messages	the database, the query, sometimes data
yes or no questions (blind)	the data, one letter at a time

OWASP Top 10:2025, A05 Injection; OWASP SQL Injection Prevention Cheat Sheet

Summary (2/2)

Attack	What the attacker gets
`' OR 1=1--`	every row: all users' notes
`UNION SELECT ...`	any table: the passwords
`bob'--` in the login	a session as bob
error messages	the database, the query, sometimes data
yes or no questions (blind)	the data, one letter at a time
Defence	
parameterized queries	values never become code
allow-list	for column and table names
ORM, no raw SQL text	the ORM parameterizes for you
validation, plain errors	second layers
least privilege	less damage if a bug slips through

OWASP Top 10:2025, A05 Injection; OWASP SQL Injection Prevention Cheat Sheet