

# INF 212

## ANALYSIS OF PROG. LANGS

### *ADVERSITY*

Instructors: Crista Lopes  
Copyright © Instructors.

# random quotes. just because

We have forty million reasons for failure, but not a single excuse.

It's fine to celebrate success but it is more important to heed the lessons of failure.

I can accept failure, everyone fails at something. But I can't accept not trying.

I've come to believe that all my past failure and frustration were actually laying the foundation for the understandings that have created the new level of living I now enjoy.

My great concern is not whether you have failed, but whether you are content with your failure.

Success represents the 1% of your work which results from the 99% that is called failure.

Failure is always an option.

# My quote



Failure happens!

# Approaches to failure

- Let it fail
  - ▣ Good in development: understand failure mode
- Defend against the possible and the impossible
  - ▣ Good in production. Detect and...
    - Correct?
    - Ignore?
    - Report?
    - Pass up?
    - Stop?
- Prevent
  - ▣ Ideal(ist)

# Obliviousness

- ❑ Failure is an option!
  - ▣ Especially when you learn from it to avoid it in the future
- ❑ Obliviousness exposes problems
  - ▣ Better than hiding them
  - ▣ Shows you failure conditions you might not have considered
- ❑ Fix as you go, during development
- ❑ Avoid in production

# Defensive

---

- Detect every possible failure
- Paranoid: detect unlikely failures too

# Reaction to failures

- Detect and correct (constructivist)
  - Detect and ignore (lazy)
  - Detect and report (tantrum)
  - Detect and pass up the stack (passive-aggressive)
- 
- Recover vs. Stop immediately

# Overreaction is bad



```
Integer one = 1;  
Integer two = one + one;
```

```
if (two > one) {  
    ...do stuff...  
}
```



# Overreaction is bad

```
public class Person {
    private String name = null;

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }
}

public Person newPerson(String name) {
    Person person = new Person();
    if (name != null) {
        person.setName(name);
    }
    return person;
}
```

# Lazyness is bad

```
void addFriendToList(List<Friend> friends, Friend newFriend) {  
    if (friends != null && newFriend != null) {  
        friends.add(newFriend);  
    }  
}
```

bad

```
void addFriendToList(List<Friend> friends, Friend newFriend) {  
    friends.add(newFriend);  
}
```

better

```
void addFriendToList(List<Friend> friends, Friend newFriend) {  
    if (friends != null && newFriend != null) {  
        friends.add(newFriend);  
    }  
    else throw new Exception("...");  
}
```

better

# Lazyness is bad

```
public List<Friend> findFavoriteFriends(Person person) {
    List<Friend> favoriteFriends = new ArrayList<Friend>();

    if (person != null) {
        List<Friend> friends = person.getFriends();

        if (friends != null) {
            for (Friend friend : friends) {
                if (friend != null) {
                    if (friend.isFavorite()) {
                        favoriteFriends.add(friend);
                    }
                }
            }
        }
    }

    return favoriteFriends;
}
```

bad

# Spectrum of reactions

- Recover: do you have a good guess for reasonable state?
- Report & proceed
- Pass up
- Fail fast: avoid corruptions by stopping immediately after a failure occurs

# Recover

```
public class Person {
    private String name = "Unknown User";

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }
}

public Person(String name) {
    if (name != null) {
        person.setName(name);
    }
    // otherwise, use default
    return person;
}
```

# Report & Proceed

```
public class Person {
    private String name = "Unknown User";

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }
}

public Person(String name) {
    if (name == null) {
        log.Warn("Person constructor given null name arg");
    }
    person.setName(name);
    // otherwise, use default
    return person;
}
```

# Pass up

```
public class Person {
    private String name = "Unknown User";

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }
}

public Person(String name) {
    if (name == null) {
        raise new Exception("null name");
    }
    person.setName(name);
    // otherwise, use default
    return person;
}
```

# Fail fast

```
public class Person {
    private String name = "Unknown User";

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }
}

public Person(String name) {
    if (name == null) {
        log.Warn("Person constructor given null name arg");
        System.exit(1);
    }
    person.setName(name);
    // otherwise, use default
    return person;
}
```



# Preventing failures

---

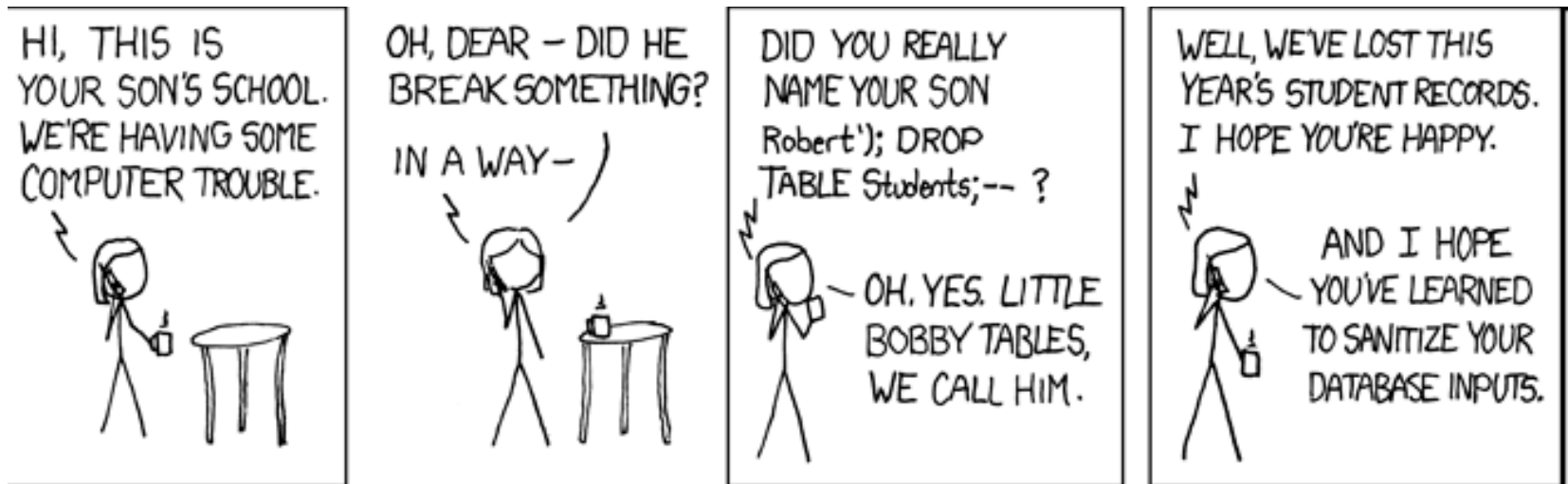
- Before the program runs:
  - ▣ Quarantine vulnerable code
  - ▣ Type checking (next lecture)
  - ▣ Test (won't be covered)

# Vulnerable code

---

- Anything that deals with IO
  - ▣ From users
  - ▣ From network
  - ▣ From database

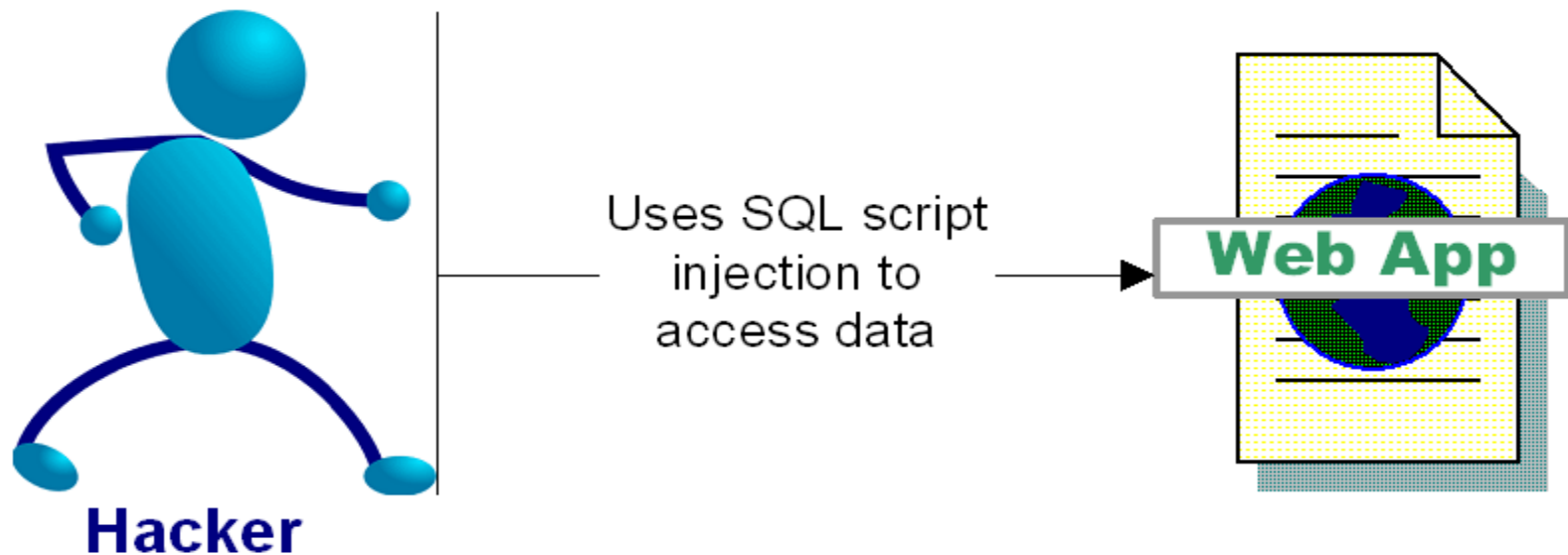
# Exploits of a Mom



<http://xkcd.com/327/>

# SQL Injection Attacks

“**SQL injection** is a security vulnerability that occurs in the database layer of an application. Its source is the incorrect escaping of dynamically-generated string literals embedded in SQL statements.”  
(Wikipedia)



# Impact of SQL Injection - Dangerous

- At best: you can leak information
- Depending on your configuration, a hacker can
  - ▣ Delete, alter or create data
  - ▣ Grant direct access to the hacker
  - ▣ Escalate privileges and even take over the OS

# SQL Injection Attacks

## □ Login Example Attack

- Text in blue is your SQL code, Text in orange is the hacker input, black text is your application code

□ Login:  Password:

## □ Dynamically Build SQL String performing authentication:

- `"SELECT * FROM users WHERE login = " + userName + " and password=" + password + ""`;

## □ Hacker logs in as: ' or ' = '; --

- `SELECT * FROM users WHERE login = " or ' = "; --" and password="`

# More Dangerous SQL Injection Attacks

- Hacker creates a Windows Account:
  - ▣ `SELECT * FROM users WHERE login = ''; exec master..xp_cmdshell 'net users username password /add';--' and password= ''`
- And then adds himself as an administrator:
  - ▣ `SELECT * FROM users WHERE login = ''; exec master..xp_cmdshell 'net localgroup Administrators username /add';--' and password= ''`
- SQL Injection examples are outlined in:
  - ▣ <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>
  - ▣ <http://www.unixwiz.net/techtips/sql-injection.html>

# Preventing SQL injection

- Use Prepared Statements (aka Parameterized Queries)
  - ▣ `$id=1234`
  - ▣ `“select * from accounts where id = “ + $id`  
vs
  - ▣ `“select * from accounts where id =1234”`
- Validate input
  - ▣ Strong typing
    - If the id parameter is a number, try parsing it into an integer
  - ▣ Business logic validation
- Escape questionable characters (ticks, --, semi-colon, brackets, etc.)



# More than SQL

- “Injection Flaw” is a blanket term
- SQL Injection is most prevalent
- Other forms:
  - ▣ XPath Injection
  - ▣ Command Injection
  - ▣ LDAP (Lightweight Directory Access Protocol) Injection
  - ▣ DOM (Document Object Model) Injection
  - ▣ JSON (Javascript Object Notation) Injection
  - ▣ Log Spoofing
  - ▣ On and on and on...

# IO Monad

- A explicit reminder that you can't trust IO
- “Promote” IO-bound functions to higher-order
  - ▣ They don't run until you make an effort