Craft Compliance

Software Security

When to Sanitize User Data and Why

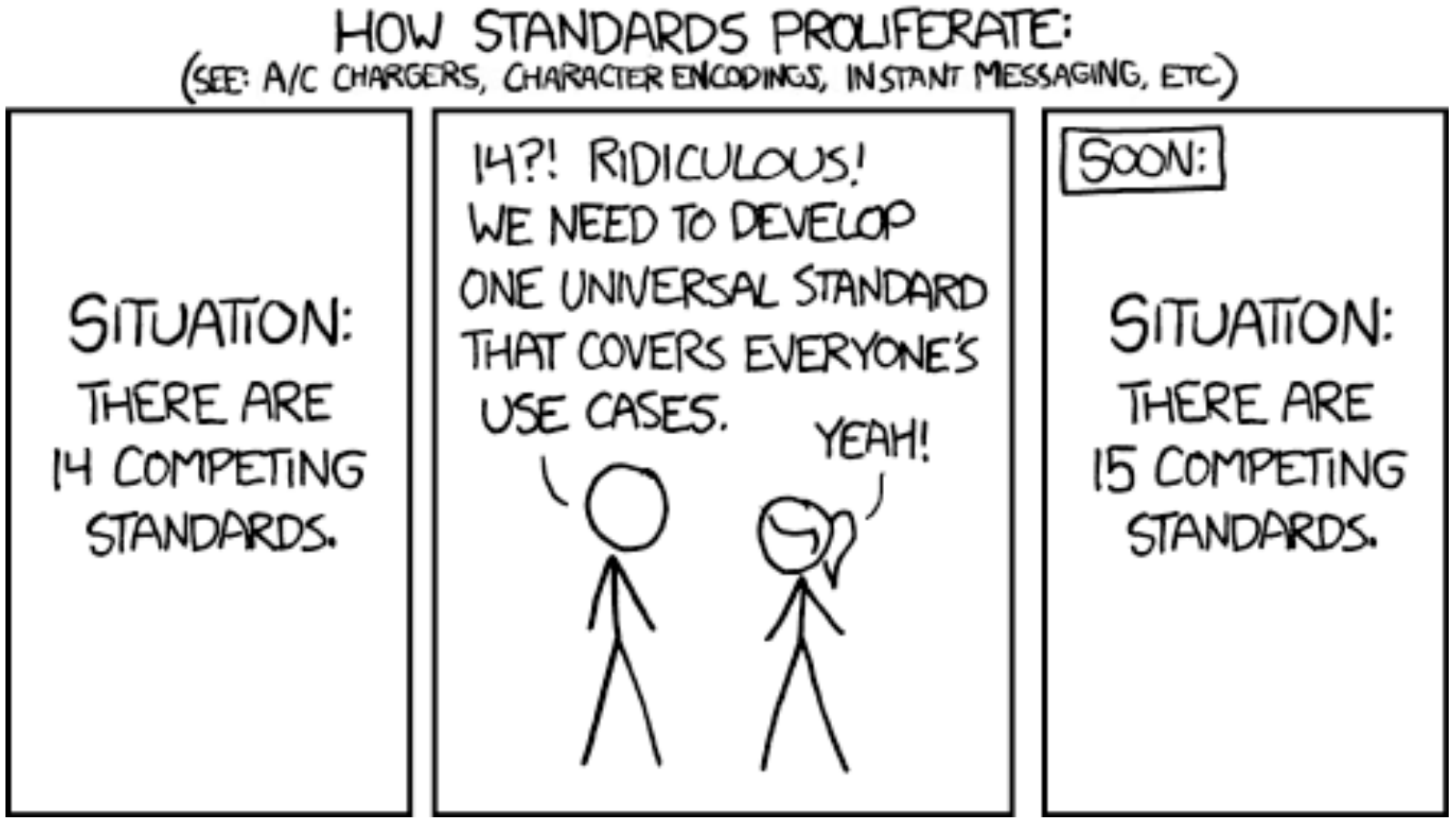
whoami

- Nathaniel (Nat) Shere
- Cybersecurity Consultant
  - Penetration testing (“ethical hacking”)
  - Secure Web Development
- Security Engineer
- Hobbies: security, programming, board games

What I Will Cover

- The Importance of Sanitizing Data
- Where to Sanitize
  - Different Approaches
  - Pros and Cons of Approaches
- Code Examples

A Word of Caution

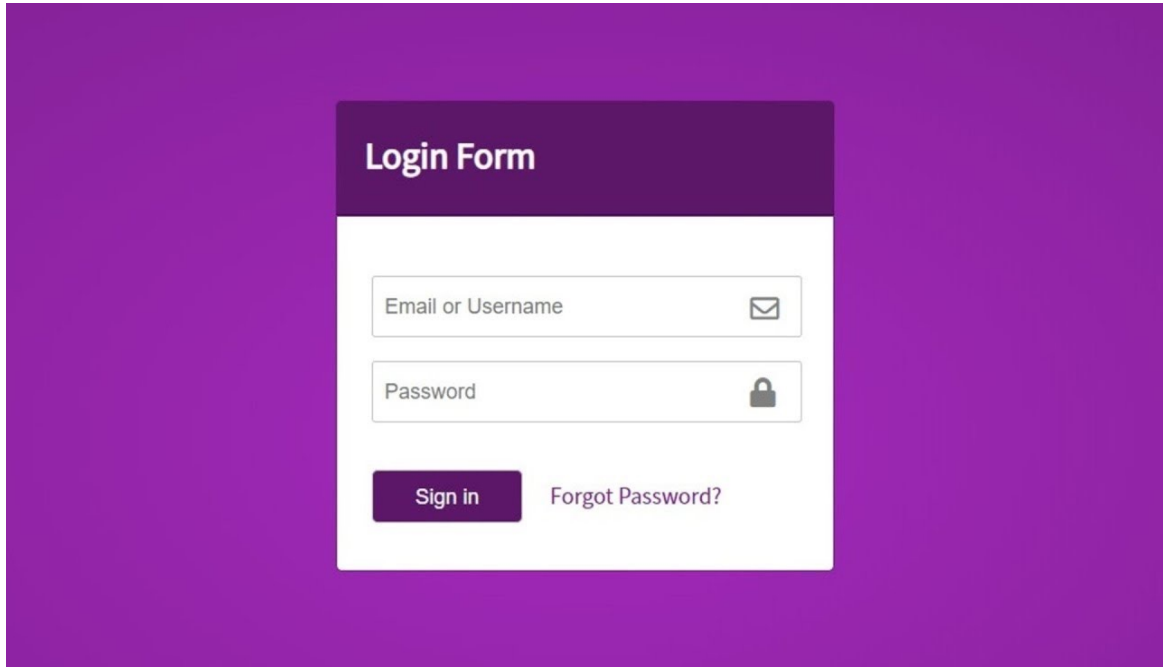


Source: https://xkcd.com/927/

Importance of Sanitizing Data

- Security
- Performance
  - Easy for developers to *maintain*
- Usability
  - Easy for users to *use*

Importance of Sanitizing Data: Security



admin:secretpassword

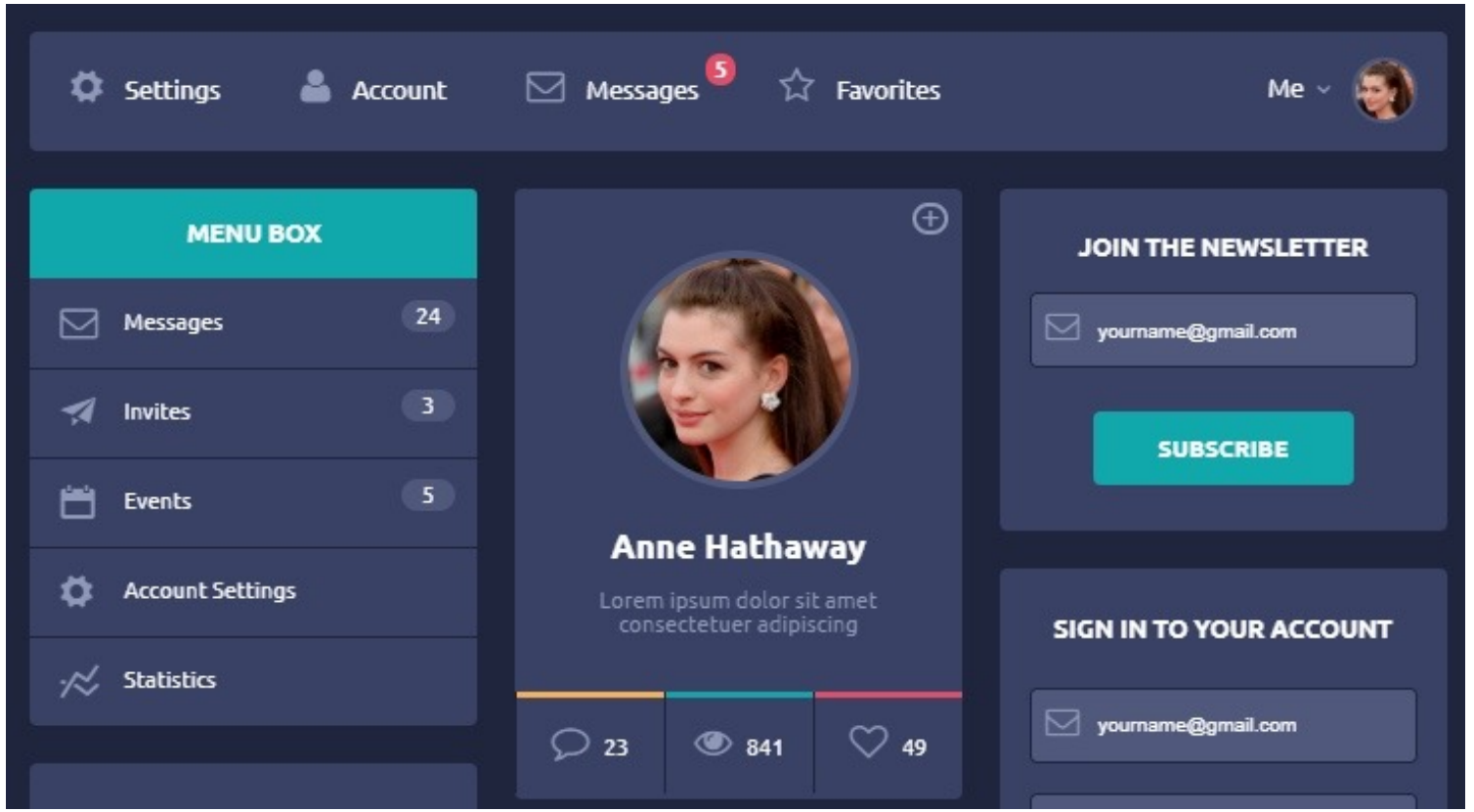
*SELECT * FROM users WHERE username='admin' AND password='secretpassword';*

Importance of Sanitizing Data: Security

admin:' OR 1=1;--

*SELECT * FROM users WHERE username='admin' AND password='' OR 1=1;--';*

Importance of Sanitizing Data: Security



Data Enumeration

GET /users/**4**/profile
GET /users/**5**/profile
GET /users/**6**/profile

Importance of Sanitizing Data: Security

Vulnerability	Remediation
SQL Injection	Sanitize User Input
Cross Site Scripting	Sanitize User Input
Insecure Deserialization	Sanitize User Input
Path Traversal	Sanitize User Input
File Uploads	Sanitize User Input
Code/OS Injection	Sanitize User Input
Buffer Overflow	Sanitize User Input
Open Redirection	Sanitize User Input
Server Side Request Forgery	Sanitize User Input
Insecure Object Reference	Sanitize User Input

Importance of Sanitizing Data: Security

- 17 Out of 25 of the Top 25 Most Dangerous Software Weaknesses (CWE Top 25) are directly a result of malicious user input.
- The other 8 are indirect results.

Importance of Sanitizing Data: Performance

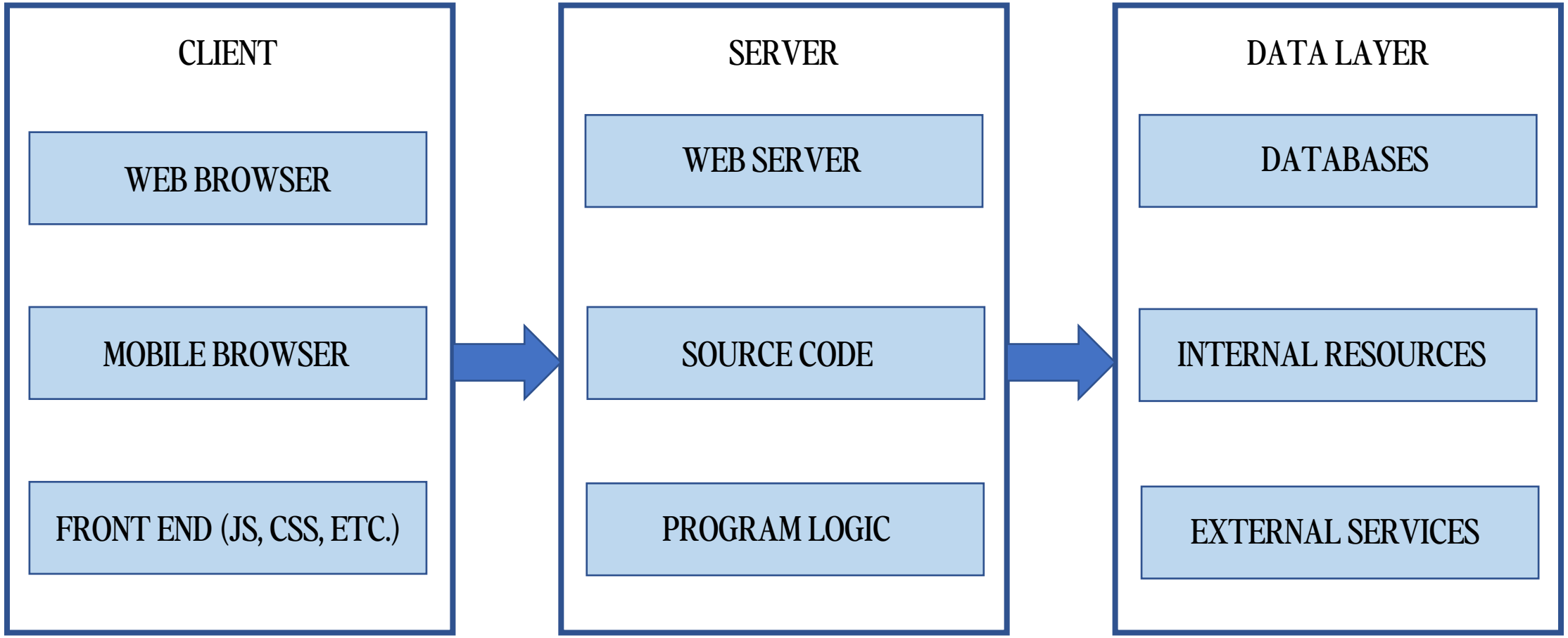
- Reduces server errors
- Data quality
- Efficiency

Importance of Sanitizing Data: Usability

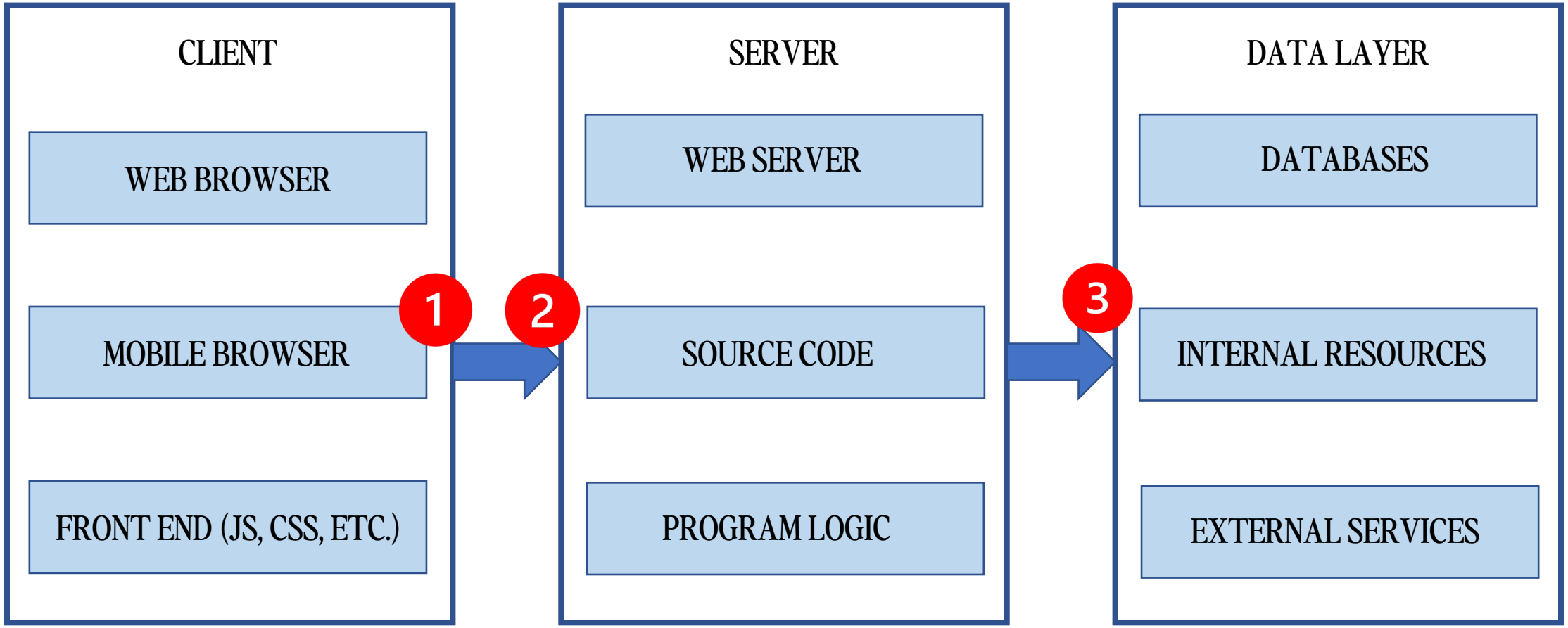
- Smoother experience for the user
- More effective use of support staff

- Stop making ”strange” names call for user support
- Stop preventing stronger security by limiting passwords

Where to Sanitize – Application Architecture



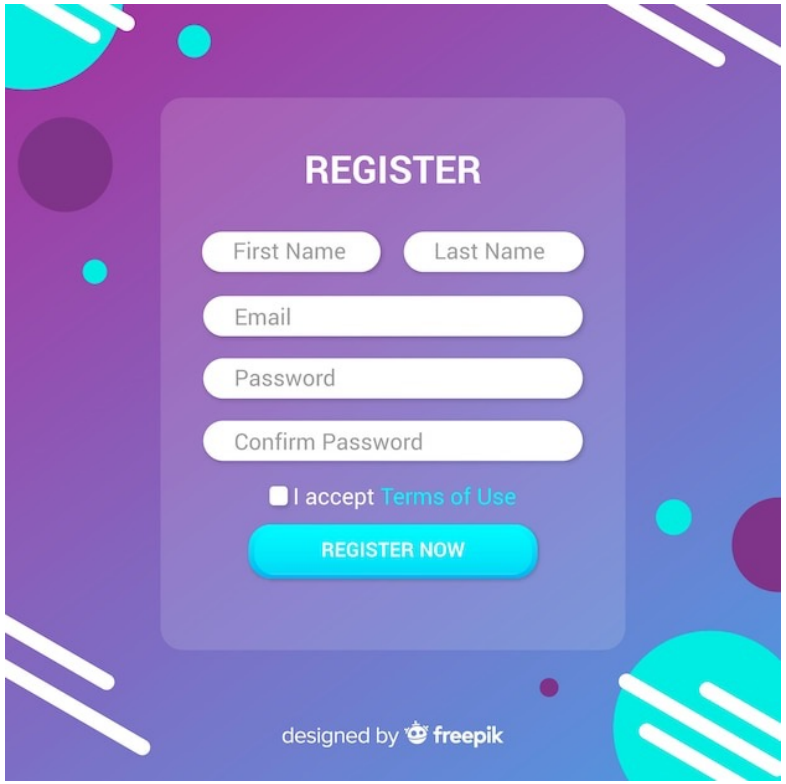
Where to Sanitize - Options



Where to Sanitize

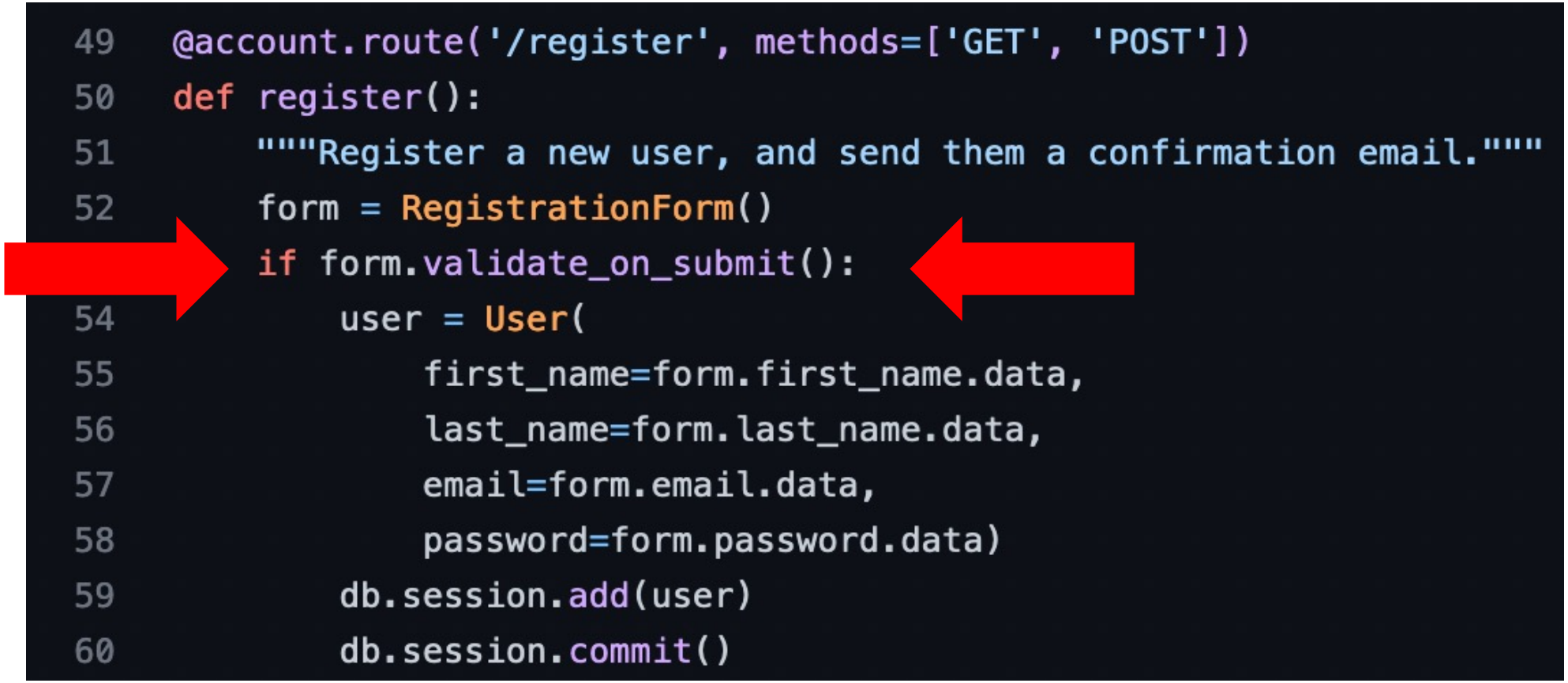
1. When data is created
   - JavaScript controls in the browser
2. When data is received
   - Server/program receives the data from the user
3. When data is used
   - Database processes the data
   - Code logic makes decisions based on the data

When Data is Created



- Performance: ✅
- Usability: ✅
- Security: ❌

When Data is Received



```
49  @account.route('/register', methods=['GET', 'POST'])
50  def register():
51      """Register a new user, and send them a confirmation email."""
52      form = RegistrationForm()
        if form.validate_on_submit():
54          user = User(
55              first_name=form.first_name.data,
56              last_name=form.last_name.data,
57              email=form.email.data,
58              password=form.password.data)
59          db.session.add(user)
60          db.session.commit()
```

Source: https://github.com/hack4impact/flask-base

When Data is Received

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entity
{
    public class Customer
    {
        [Required]
        public byte CustomerImage { get; set; }

        [Required, MinLength(3), MaxLength(40)]
        public string Surname { get; set; }

        [Required, MinLength(3), MaxLength(40)]
        public string FirstName { get; set; }

        [Required]
        [RegularExpression(@"^[A-Z]+[a-zA-Z'\s]*$")]
        public string Password {get; set;}

        [Required, EmailAddress]
        public string Email { get; set; }
    }
}
```

Source: https://www.securecoding.com/blog/input-validation/

When Data is Used

```
1  s = ""
2  s += "INSERT INTO tbl_products"
3  s += "("
4  s += "ID_product"
5  s += ", t_name_product"
6  s += ") VALUES ("
7  s += "(%ID_product)"
8  s += ", '(%t_name_product)'"
9  s += ")"
10 db_cursor.execute(s, [ID_product, t_name_product])
```

When Data is Received - Advantages

- No bad data in environment
- Faster feedback
- Less time for malicious data to find a flaw
- Downstream logic can trust data

When Data is Received - Disadvantages

- More development time
- Whitelist vs. blacklist
- More documentation
- Distance between validation and execution

When Data is Used - Advantages

- Less development time
- Less reliance on lists
- More resilient to change

When Data is Used - Disadvantages

- Validation logic may get duplicated
- Problems may be ignored
- Bad data may linger

If We Have to Prioritize...

- Context context context

- Plan ahead
  - Pick the best approach for each issue
  - Pick a consistent approach

Best Approach for Each Issue

SQL Injection -- Parameterize

```
public static string secureLogin(string userName, string passWord)
{
    MySqlConnection conn = getConnection("details");
    MySqlCommand command = new MySqlCommand("SELECT userName FROM Users WHERE userName=@userName AND
passWord=@passWord", conn);
    command.Parameters.AddWithValue("@userName", userName);
    command.Parameters.AddWithValue("@passWord", passWord);
    object o = command.ExecuteScalar();
    conn.Close();
    return ((o == null) ? "Error logging in" : "You successfully logged in as " + o.ToString());
}
```

Best Approach for Each Issue

Authorization Controls - Validate

```
164  @admin.route('/user/<int:user_id>/_delete')
165  @login_required
166  @admin_required
167  def delete_user(user_id):
168      """Delete a user's account."""
169      if current_user.id == user_id:
170          flash('You cannot delete your own account. Please ask another '
171                'administrator to do this.', 'error')
172      else:
173          user = User.query.filter_by(id=user_id).first()
174          db.session.delete(user)
175          db.session.commit()
176          flash('Successfully deleted user %s.' % user.full_name(), 'success')
177      return redirect(url_for('admin.registered_users'))
178
```

Source: https://github.com/hack4impact/flask-base

Best Approach for Each Issue

Vulnerability	Best Approach
SQL Injection	Sanitize when used – Parameterization
Cross Site Scripting	Sanitize when used – Encoding
Insecure Deserialization	Sanitize when used – Specify the target object type
Path Traversal	Sanitize when received - whitelist
File Uploads	Sanitize when received - whitelist
Code/OS Injection	Sanitize when received - whitelist
Buffer Overflow	Sanitize when received – length validations
Open Redirection	Sanitize when received - whitelist
Server-Side Request Forgery	Sanitize when received - whitelist
Insecure Object Reference	Sanitize when received – authorization checks

Consistent Approach

- Other controls can be implemented to minimize approach weaknesses
- Document document document

Existing Standards

- https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html
- https://github.com/OWASP/DevGuide/blob/master/03-Build/0x06-InputValidation.md
- https://security.berkeley.edu/secure-coding-practice-guidelines
- https://www.sans.org/cloud-security/securing-web-application-technologies/
- NIST SP 800-53 Rev 5 3.19 SI-10

Questions?

nathaniel.shere@craftcompliance.com

https://www.linkedin.com/in/nathaniel-shere/