1. HTML Form with JS Validation

This is a single HTML file. The `onsubmit` attribute calls the `validate()` function. If `validate()` returns `false`, the form won't submit.

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Form Validation</title>
</head>
<body>

    <form id="myForm" onsubmit="return validateForm()">
        Email: <input type="text" id="email"><br>
        Phone: <input type="text" id="phone"><br>
        Password: <input type="password" id="password"><br>
        <input type="submit" value="Submit">
    </form>

    <script>
    function validateForm() {
        let email = document.getElementById('email').value;
        let phone = document.getElementById('phone').value;
        let pass = document.getElementById('password').value;

        // Simple regex for email
        let emailRegex = /\S+@\S+\.\S+/;
        if (!emailRegex.test(email)) {
            alert("Invalid email address.");
            return false;
        }

        // Simple check for 10-digit phone
        if (isNaN(phone) || phone.length !== 10) {
            alert("Phone number must be 10 digits.");
            return false;
        }

        // Check password length
        if (pass.length < 8) {
            alert("Password must be at least 8 characters long.");
            return false;
        }

        alert("Form submitted successfully!");
        return true; // Form will submit
```

```
  }
  </script>

</body>
</html>
```

---

2. SQL Injection: Vulnerability and Fix

Let's assume a simple PHP/MySQL setup.

The Vulnerable Code: The code directly inserts user input into the SQL query.

```
PHP
// VULNERABLE
$username = $_POST['username']; // e.g., ' OR '1'='1
$password = $_POST['password'];

$sql = "SELECT * FROM users WHERE username = '$username' AND password =
'$password'";
// The attacker's input makes the query:
// SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '...'
// This logs them in as the first user without a valid password.
```

The Fix (Prepared Statements): This separates the query logic from the data, so user input is *always* treated as data, not as part of the command.

```
PHP
// FIXED
$username = $_POST['username'];
$password = $_POST['password'];

// 1. Prepare the statement
$stmt = $db_connection->prepare("SELECT * FROM users WHERE username = ? AND
password = ?");

// 2. Bind parameters (s = string)
$stmt->bind_param("ss", $username, $password);

// 3. Execute
$stmt->execute();
// No injection is possible. The DB looks for a user with the *literal* username "' OR '1'='1".
```

---

3. Git: Repo, Commit, Push

These are commands you type into your terminal.

Create a repository:
Bash
Create a new folder and go into it
mkdir my-project
cd my-project

Initialize a new Git repository
git init

1.

Make commits:
Bash
Create a file
echo "Hello World" > readme.md

Add the file to be tracked
git add readme.md

Save a snapshot (commit)
git commit -m "Initial commit: Added readme"

2.

Push code to GitHub: (Assumes you've already created an empty repo on GitHub.com)
Bash
Add the GitHub repo as a remote "origin"
git remote add origin https://github.com/your-username/your-repo-name.git

Push your 'main' (or 'master') branch to the remote
git push -u origin main

3.

> **How it maintains code integrity:** Git gives every commit a unique ID (a **hash**) based on its contents and history. If you change *anything* (even a single comma) in a past commit, the hash changes, and all subsequent commit hashes also change. This creates a verifiable chain, making it obvious if history has been tampered with.

---

4. Burp Suite: Intercept Request

1. **Open Burp Suite** and go to the **Proxy > Intercept** tab.
2. **Configure your browser** to use Burp as a proxy (e.g., use FoxyProxy extension, set to `127.0.0.1` on port `8080`).

3. Click the **"Intercept is on"** button in Burp.
4. **In your browser**, visit a simple (non-HTTPS) demo site, like `http://testphp.vulnweb.com`.

Burp will "catch" the request before it goes to the server. 6. **Look at the captured text:**
HTTP
GET / HTTP/1.1
Host: testphp.vulnweb.com
User-Agent: Mozilla/5.0 (...)
Cookie: JSESSIONID=abc123...

5.
6. **Identify insecure parameters:** If you were logging in, you might see `username=test&password=123` in the request body. If you visited `http://testphp.vulnweb.com/listproducts.php?id=1`, the `id=1` is a user-controllable parameter and a prime target for attack.

---

5. DVWA: OS Command Injection & Mitigation

This assumes you have DVWA (Damn Vulnerable Web Application) running.

The Attack:

1. Log in to DVWA and set the **DVWA Security** level to **Low**.
2. Go to the **Command Injection** page.
3. The page asks for an IP to ping. Enter a normal IP: `8.8.8.8`. It works.
4. Now, inject a second command. On Linux, use `&&` or `|`. On Windows, use `&`.
    - Input: `8.8.8.8 && ls` (Linux) or `8.8.8.8 & dir` (Windows)
5. You will see the ping results *and* the output of your injected command (a list of files/directories).

The Mitigation (Input Sanitization):

The prompt suggests `filter_input()` or `regex`. The best and simplest fix using these is to **validate the input** as a legitimate IP address.

Here's the *bad* (vulnerable) code from DVWA (Low):

PHP
// VULNERABLE
$target = $_REQUEST['ip'];
$cmd = shell_exec('ping -c 4 ' . $target);
echo "<pre>{$cmd}</pre>";

Here's a *good* (mitigated) version using `filter_input()`:

PHP
```
// FIXED
$target = $_REQUEST[ 'ip' ];

// Validate the input as an IP address
// If it's not a valid IP, it will be set to false.
$safe_ip = filter_var($target, FILTER_VALIDATE_IP);

if ($safe_ip === false) {
    echo "Invalid IP address.";
} else {
    // Only the validated, "safe" IP is used
    $cmd = shell_exec( 'ping -c 4 ' . $safe_ip );
    echo "<pre>{$cmd}</pre>";
}
```

---

6. Python: Caesar Cipher

This Python code shifts only uppercase letters. It's the simplest example.

Python
```
def encrypt(text, key):
    result = ""
    # Go through each character
    for char in text:
        # Encrypt uppercase letters
        if 'A' <= char <= 'Z':
            # ord(A) is 65
            new_code = (ord(char) - 65 + key) % 26 + 65
            result += chr(new_code)
        # Leave other characters (lowercase, spaces, etc.) as is
        else:
            result += char
    return result

def decrypt(text, key):
    # Decrypting is just encrypting with a negative key
    return encrypt(text, -key)

# --- Example ---
message = "HELLO"
key = 3

encrypted_message = encrypt(message, key)
print(f"Encrypted: {encrypted_message}") # Output: KHOOR
```

```
decrypted_message = decrypt(encrypted_message, key)
print(f"Decrypted: {decrypted_message}") # Output: HELLO
```

---

7. Basic Login Form with Session Cookie (PHP)

This requires 3 files in the same directory.

`login.php` (The form)

```
PHP
<?php
session_start(); // Start the session
// Check if user is submitting the form
if (isset($_POST['username'])) {
    $user = $_POST['username'];
    $pass = $_POST['password'];

    // VERY basic check. Don't do this in real life.
    if ($user == 'admin' && $pass == '123') {
        // Set session variables (this creates the cookie)
        $_SESSION['loggedin'] = true;
        $_SESSION['username'] = $user;
        header('Location: welcome.php'); // Redirect to protected page
        exit;
    } else {
        echo "Wrong username or password!";
    }
}
?>

<form method="post">
    Username: <input type="text" name="username"><br>
    Password: <input type="password" name="password"><br>
    <input type="submit" value="Log In">
</form>
```

`welcome.php` (The protected page)

```
PHP
<?php
session_start(); // Re-join the session

// If the 'loggedin' session variable doesn't exist, kick them back to login.
if (!isset($_SESSION['loggedin']) || $_SESSION['loggedin'] !== true) {
    header('Location: login.php');
```

```
    exit;
}

// Welcome the user
echo "Hello, " . $_SESSION['username'];
echo '<br><a href="logout.php">Log Out</a>';
?>
```

`logout.php` (To clear the session)

```
PHP
<?php
session_start();
session_unset(); // Unset all session variables
session_destroy(); // Destroy the session (and the cookie)
header('Location: login.php'); // Send back to login page
exit;
?>
```

---

8. Feedback Form (Frontend & Backend Validation)

Frontend (HTML/JS): Uses the `required` attribute for simple validation and `htmlspecialchars` to prevent script tags *on display*.

```
HTML
<form action="submit.php" method="post">
    Name: <input type="text" name="name" required><br>
    Email: <input type="email" name="email" required><br>
    Feedback: <textarea name="feedback" required></textarea><br>
    <input type="submit" value="Submit">
</form>
```

Backend (PHP): This is the *most important* part. It re-validates and sanitizes to prevent XSS (script tags).

```
PHP
// submit.php
<?php
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $name = $_POST['name'];
    $email = $_POST['email'];
    $feedback = $_POST['feedback'];

    // 1. Backend Validation (check for empty)
```

```
    if (empty($name) || empty($email) || empty($feedback)) {
        die("Error: All fields are required.");
    }

    // 2. Backend Validation (check email format)
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        die("Error: Invalid email format.");
    }

    // 3. Prevent XSS (Script Tags)
    // htmlspecialchars() converts < into &lt; and > into &gt;
    // This makes script tags harmless.
    $safe_name = htmlspecialchars($name);
    $safe_feedback = htmlspecialchars($feedback);

    // Now it's safe to display or store
    echo "Thank you, " . $safe_name . "!<br>";
    echo "We received your feedback: " . $safe_feedback;
}
?>
```

---

9. Password Strength Checker (JS)

This gives live feedback as the user types.

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Password Strength</title>
</head>
<body>

    Password: <input type="password" id="password" onkeyup="checkStrength()">
    <div id="strength-message"></div>

    <script>
    function checkStrength() {
        let pass = document.getElementById('password').value;
        let message = document.getElementById('strength-message');
        let issues = [];

        // Check for minimum length
        if (pass.length < 8) {
            issues.push("at least 8 characters");
        }
```

```
        // Check for a number
        if (!pass.match(/[0-9]/)) {
            issues.push("a number (0-9)");
        }

        // Check for a special character
        if (!pass.match(/[\W_]/)) { // \W is non-word char, _ is for underscore
            issues.push("a special character (!, @, #, etc.)");
        }

        // Display results
        if (issues.length > 0) {
            message.innerHTML = "Password must include: " + issues.join(', ');
            message.style.color = 'red';
        } else {
            message.innerHTML = "Password strength is good!";
            message.style.color = 'green';
        }
    }
    </script>

</body>
</html>
```

---

10. GitHub Personal Access Token (PAT)

This is a replacement for using your password with Git operations.

How to Generate:

1. Log in to **GitHub**.
2. Click your profile picture (top right) -> **Settings**.
3. On the left menu, scroll down and click **Developer settings**.
4. Click **Personal access tokens** -> **Tokens (classic)**.
5. Click **Generate new token** (and confirm your password).
6. **Name:** Give it a name (e.g., "My Laptop CLI").
7. **Expiration:** Choose an expiration (e.g., 30 days).
8. **Scopes:** Select the permissions. For pushing and pulling code, the `repo` scope is all you need.
9. Click **Generate token** at the bottom.
10. **COPY THE TOKEN!** It will look like `ghp_abc123...`. GitHub will *never* show it to you again. Save it in your password manager.

How to Use (on the command line):

When you `git push` or `git pull` from a new repository:
Bash
$ git push -u origin main
Username for 'https://github.com': your-username
Password for 'https://your-username@github.com':

1.
2. Enter your **username**.
3. When it asks for **Password**, **paste your Personal Access Token (PAT)**.

It's more secure because:

- You can revoke a token anytime without changing your main password.
- You can give tokens limited permissions (e.g., read-only).