Code Injection, Dynamic Evaluation, and CSP Mitigation in a Web App

Asrith Krishna Vejandla

Student ID: 005033913

MSCS-535 Secure Software Development

Project

Professor Yousef Nijim

July 27th, 2025

Questions

1. Provide a JavaScript code injection via web applications, where an attacker can inject malicious scripts into web pages.

In this project, I created a minimal web application using Node.js and Express that serves two pages. The route /vulnerable demonstrates a client-side code injection problem. The page includes a simple input text field tag, where user-supplied input is appended to the DOM using innerHTML. When untrusted data is written with innerHTML, the browser parses it as HTML. Although script tags inserted this way do not execute, event handler attributes and other HTML based payloads still run in the page context. This is enough to demonstrate a real cross-site scripting issue because the attacker controls execution in the victim's browser. MDN documents that <script> tags added through innerHTML do not run, and also notes that other constructs like <img onerror=…> still execute, which is the vector used here.

The vulnerable portion of the page is shown below. The key line is the assignment to innerHTML.

<!-- public/vulnerable.html -->

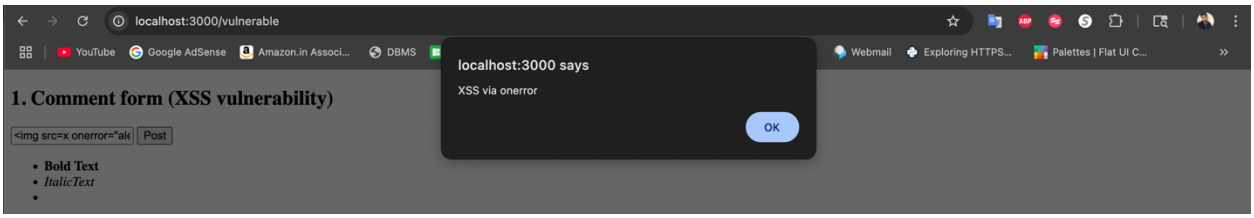
```
<input type="text" id="commentInput" placeholder="Your comment">

<button id="post">Post</button>

<ul id="comments"></ul>



<script>

  document.getElementById('post').onclick = () => {
```

```
    const t = document.getElementById('commentInput').value;
    const li = document.createElement('li');
    li.innerHTML = t;                // vulnerable sink
    document.getElementById('comments').appendChild(li);
  };
</script>
```

I verified the injection in the browser with a payload that creates a broken image and triggers its error handler. This produces visible evidence and an alert dialog.

Test input used in the comment box on /vulnerable:

<img src=x onerror="alert('XSS via onerror')">



I also confirmed that plain HTML renders, which proves that innerHTML parsed the markup rather than escaping it. For example, entering bold shows the word in bold. Finally, I confirmed that a script tag like <script>alert(1)</script> inserts a script node but does not execute. MDN explains this behavior of innerHTML.

2. Provide a dynamic evaluation of code at runtime, such as eval() in JavaScript that can be exploited by attackers.

The second requirement is covered by a small calculator on the /vulnerable page. The implementation passes whatever the user types directly into JavaScript's eval. This is dynamic code evaluation at runtime, and it is unsafe. MDN warns that evaluating JavaScript from strings is a significant security risk because it enables arbitrary code execution if the input is attacker-controlled.

The calculator's vulnerable code is shown below. The sink is the call to eval(expr).

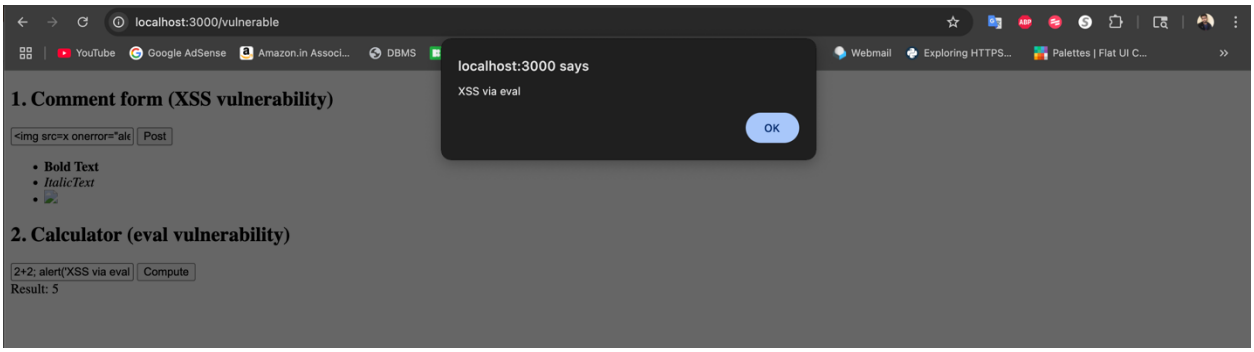
```
<!-- public/vulnerable.html (eval part) -->
<input id="expr" placeholder="Expression">
<button id="calc">Compute</button>
<div>Result: <span id="output"></span></div>

<script>
  document.getElementById('calc').onclick = () => {
    try {
      const expr = document.getElementById('expr').value;
      const r = eval(expr);          // dynamic code execution
      document.getElementById('output').textContent = r;
    } catch {
```

```
    document.getElementById('output').textContent = 'Error';
  }
};
</script>
```

I demonstrated the risk by entering a valid arithmetic expression with an injected side effect. The math result is computed, and the extra code runs too.

Test input used in the calculator on /vulnerable:

2+2; alert('XSS via eval')



3. Then provide the mitigation code to fix the security problems for the vulnerabilities you listed in 1 and 2.

I implemented two layers of mitigation on the /fixed route. First, I removed the injection sinks. The input text field now uses textContent, which treats all user-supplied strings as literal text, so tags and event handlers cannot execute. For the calculator, I replaced eval with a small arithmetic parser that only accepts digits, periods, whitespace, parentheses, and the operators + - * /. The parser tokenizes the input, converts it to Reverse Polish Notation using operator precedence, and evaluates the RPN stack. There is no dynamic code execution. Second, I added a strict Content Security Policy header to the /fixed response that allows scripts only from the same origin and disallows inline scripts. CSP gives an extra safety net in case a future change accidentally reintroduces an unsafe sink. MDN and OWASP both describe CSP as a defense-in-depth control that reduces the exploitability of XSS defects.

The safe comment code is straightforward.

```
<!-- public/fixed.html (excerpt) -->
<input type="text" id="commentInput" placeholder="Your comment">
<button id="post">Post</button>
<ul id="comments"></ul>
<script src="/static/js/fixed.js"></script>
```

```
// public/js/fixed.js (excerpt)
document.getElementById('post').addEventListener('click', () => {
  const t = document.getElementById('commentInput').value;
```

```
  const li = document.createElement('li');
  li.textContent = t;                // escaped safely
  document.getElementById('comments').appendChild(li);
});
```

The calculator mitigation avoids eval entirely.

// public/js/fixed.js (excerpt)

```
function tokenize(expr) {
  const out = []; let i = 0;
  while (i < expr.length) {
    const ch = expr[i];
    if (/\s/.test(ch)) { i++; continue; }
    if (/[0-9.]/.test(ch)) {
      let j = i, dot = 0;
      while (j < expr.length && /[0-9.]/.test(expr[j])) {
        if (expr[j] === '.') dot++;
        if (dot > 1) throw new Error('Invalid number');
        j++;
      }
```

```
      out.push({ t: 'n', v: parseFloat(expr.slice(i, j)) });
      i = j; continue;
    }
    if ('+-*/()'.includes(ch)) { out.push({ t: ch }); i++; continue; }
    throw new Error('Invalid character');
  }
  return out;
}

function toRPN(toks) {
  const out = [], st = [], prec = { '+':1, '-':1, '*':2, '/':2 };
  for (const t of toks) {
    if (t.t === 'n') out.push(t);
    else if ('+-*/'.includes(t.t)) {
      while (st.length && '+-*/'.includes(st[st.length-1].t)
             && prec[st[st.length-1].t] >= prec[t.t]) out.push(st.pop());
      st.push(t);
    } else if (t.t === '(') st.push(t);
```

```
    else if (t.t === ')') {
      while (st.length && st[st.length-1].t !== '(') out.push(st.pop());
      if (!st.length) throw new Error('Mismatched parens');
      st.pop();
    }
  }
  while (st.length) {
    const x = st.pop();
    if (x.t === '(') throw new Error('Mismatched parens');
    out.push(x);
  }
  return out;
}

function evalRPN(rpn) {
  const st = [];
  for (const t of rpn) {
    if (t.t === 'n') st.push(t.v);
```

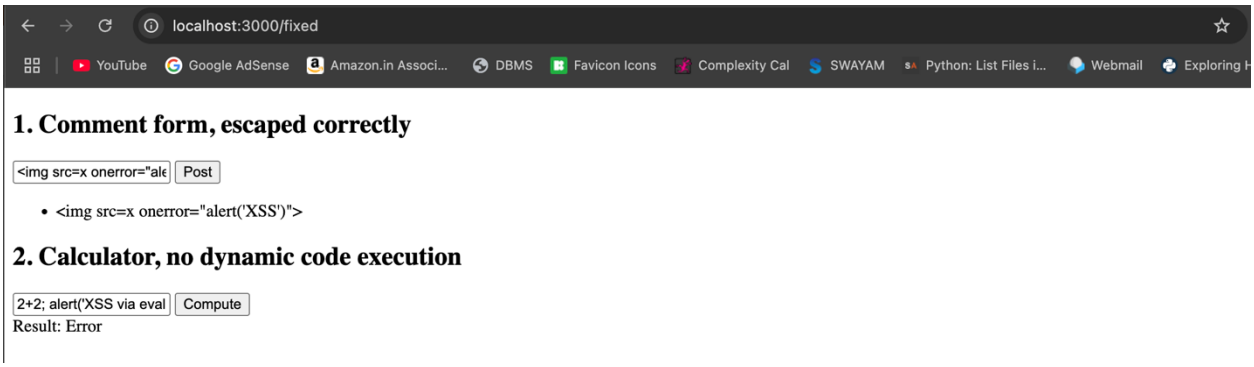
```
    else {
      const b = st.pop(), a = st.pop();
      if (a === undefined || b === undefined) throw new Error('Bad expr');
      st.push(t.t === '+' ? a+b : t.t === '-' ? a-b : t.t === '*' ? a*b : a/b);
    }
  }
  if (st.length !== 1) throw new Error('Bad expr');
  return st[0];
}

document.getElementById('calc').addEventListener('click', () => {
  const expr = document.getElementById('expr').value;
  try {
    const result = evalRPN(toRPN(tokenize(expr)));
    document.getElementById('output').textContent = String(result);
  } catch {
    document.getElementById('output').textContent = 'Error';
  }
```

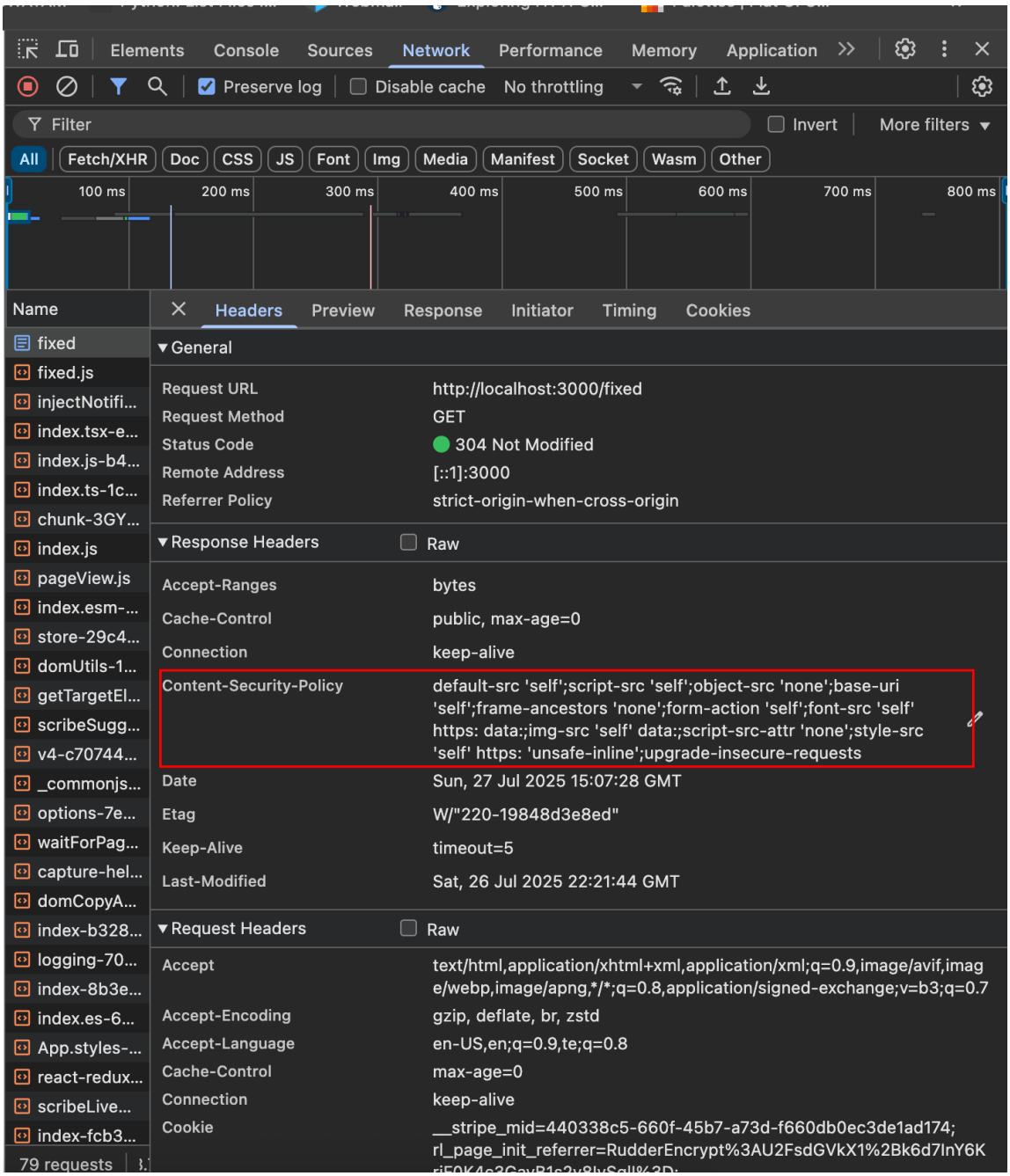
```
});
```

On the server, I attach a CSP header when serving the fixed page. The policy allows scripts from self only, blocks inline scripts, and blocks plugins. MDN's CSP reference explains how script-src 'self' controls JavaScript sources and how CSP reduces XSS impact.

```
// server.js (excerpt)
app.get('/fixed', (req, res) => {
  res.setHeader(
    'Content-Security-Policy',
    "default-src 'self'; script-src 'self'; object-src 'none'; base-uri 'self'; frame-ancestors 'none'"
  );
  res.sendFile(path.join(__dirname, 'public/fixed.html'));
});
```

I verified the mitigations by repeating the same inputs on /fixed. The comment widget displays the literal characters of <img src=x onerror="alert('XSS')"> and never executes any handler. The calculator accepts valid arithmetic like (10 + 20) * 3.5, and rejects strings that contain letters or semicolons, which results in the output "Error". The browser developer tools confirm the CSP header is present on the fixed page. These observations align with MDN and OWASP guidance on output encoding, removal of dynamic evaluation, and CSP as a second layer of protection.


localhost:3000/fixed
YouTube
Google AdSense
Amazon.in Associ...
DBMS
Favicon Icons
Complexity Cal
SWAYAM
Python: List Files i...
Webmail
1. Comment form, escaped correctly
<img src=x onerror="ale
Post
<img src=x onerror="alert('XSS')">
2. Calculator, no dynamic code execution
2+2; alert('XSS via eval
Compute
Result: Error


Elements
Console
Sources
Network
Performance
Memory
Application
Preserve log
Disable cache
No throttling
Filter
Invert
More filters
All
Fetch/XHR
Doc
CSS
JS
Font
Img
Media
Manifest
Socket
Wasm
Other
100 ms
200 ms
300 ms
400 ms
500 ms
600 ms
700 ms
800 ms
Name
Headers
Preview
Response
Initiator
Timing
Cookies
fixed
fixed.js
injectNotifi...
index.tsx-e...
index.js-b4...
index.ts-1c...
chunk-3GY...
index.js
pageView.js
index.esm-...
store-29c4...
domUtils-1...
getTargetEl...
scribeSugg...
v4-c70744...
_commonjs...
options-7e...
waitForPag...
capture-hel...
domCopyA...
index-b328...
logging-70...
index-8b3e...
index.es-6...
App.styles-...
react-redux...
scribeLive...
index-fcb3...
79 requests
General
Request URL
http://localhost:3000/fixed
Request Method
GET
Status Code
304 Not Modified
Remote Address
[::1]:3000
Referrer Policy
strict-origin-when-cross-origin
Response Headers
Raw
Accept-Ranges
bytes
Cache-Control
public, max-age=0
Connection
keep-alive
Content-Security-Policy
default-src 'self';script-src 'self';object-src 'none';base-uri 'self';frame-ancestors 'none';form-action 'self';font-src 'self' https: data:;img-src 'self' data:;script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
Date
Sun, 27 Jul 2025 15:07:28 GMT
Etag
W/"220-19848d3e8ed"
Keep-Alive
timeout=5
Last-Modified
Sat, 26 Jul 2025 22:21:44 GMT
Request Headers
Raw
Accept
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding
gzip, deflate, br, zstd
Accept-Language
en-US,en;q=0.9,te;q=0.8
Cache-Control
max-age=0
Connection
keep-alive
Cookie
__stripe_mid=440338c5-660f-45b7-a73d-f660db0ec3de1ad174;
rl_page_init_referrer=RudderEncrypt%3AU2FsdGVkX1%2Bk6d7InY6K

The entire web application is uploaded to the GitHub Repository below:

https://github.com/ImAsrith/MSCS_535_Project_2

References

1. Mozilla Contributors. (2025). Element: innerHTML property. MDN Web Docs.
2. Mozilla Contributors. (2025). HTMLScriptElement. MDN Web Docs.
3. Mozilla Contributors. (2025). eval(). MDN Web Docs.
4. Mozilla Contributors. (2025). Content Security Policy (CSP). MDN Web Docs.
5. Mozilla Contributors. (2025). Content-Security-Policy header reference. MDN Web Docs.
6. Mozilla Contributors. (2025). script-src directive. MDN Web Docs.
7. OWASP Foundation. (n.d.). Cross Site Scripting Prevention Cheat Sheet. OWASP Cheat Sheet Series.
8. OWASP Foundation. (n.d.). Content Security Policy Cheat Sheet. OWASP Cheat Sheet Series.