XSS,SSTI & SQL injection

A03:2021-Injection

Injection meaning

User input is treated as code, not data

Routes and files

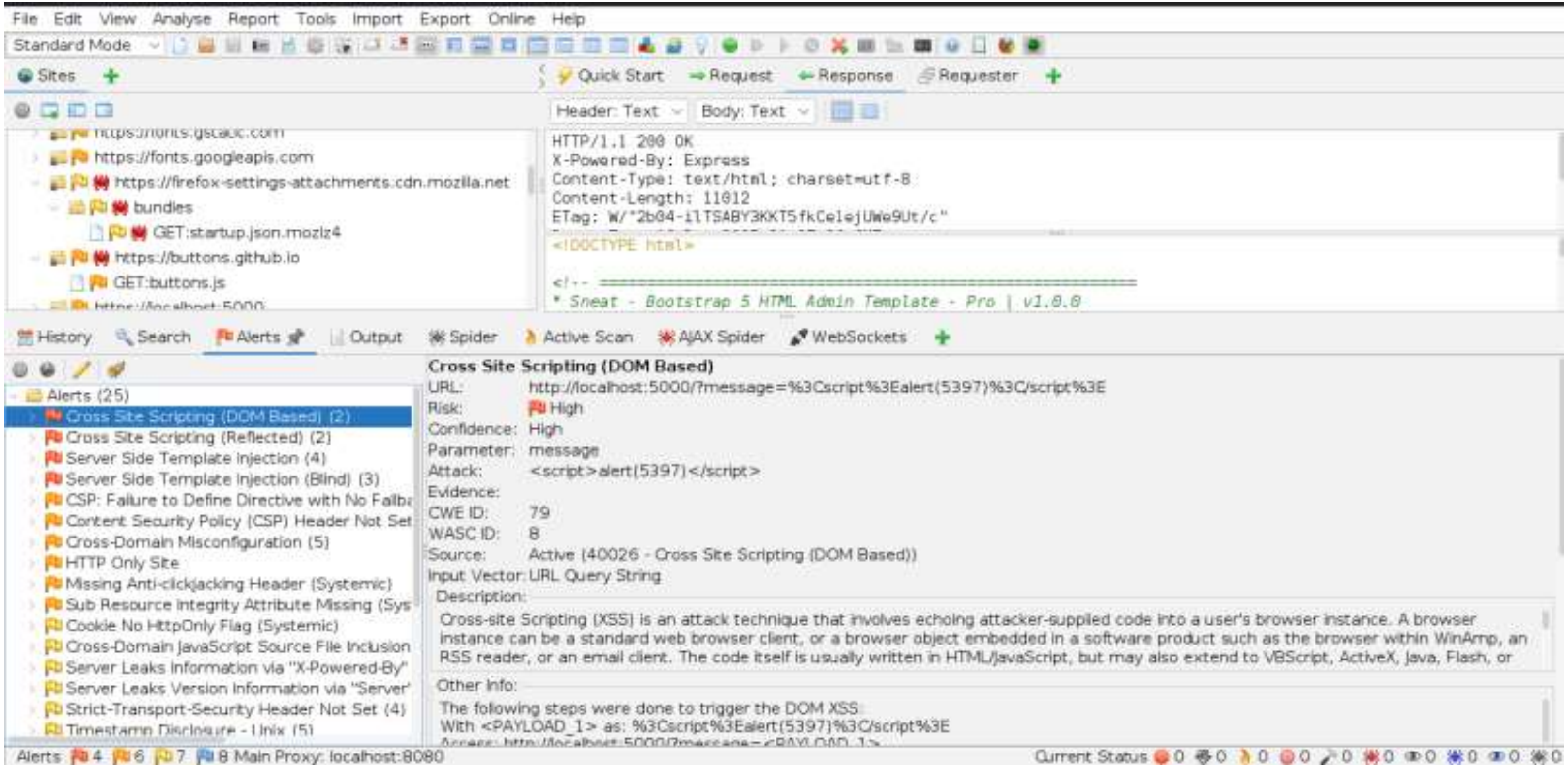
Files

- frontend.js
- Order.js
- auth-login-basic.html

Routes

- http://localhost:5000/?message={js code}
- http://localhost:5000/?message={{nunjucks evaluates}}
- /v1/search/:filter/:query

Zap tool analysis



https://fonts.googleapis.com
https://firefox-settings-attachments.cdn.mozilla.net
bundles
GET:startup.json.mozlz4
https://buttons.github.io
GET:buttons.js

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 11012
ETag: W/"2b04-i1TSABY3KKT5fkCelejUWe9Ut/c"
```

```
<!DOCTYPE html>

<!-- =================================================================
* Sneat - Bootstrap 5 HTML Admin Template - Pro | v1.0.0
```

Alerts (25)
- Cross Site Scripting (DOM Based) (2)
- Cross Site Scripting (Reflected) (2)
- Server Side Template Injection (4)
- Server Side Template Injection (Blind) (3)
- CSP: Failure to Define Directive with No Fallba
- Content Security Policy (CSP) Header Not Set
- Cross-Domain Misconfiguration (5)
- HTTP Only Site
- Missing Anti-clickjacking Header (Systemic)
- Sub Resource Integrity Attribute Missing (Sys
- Cookie No HttpOnly Flag (Systemic)
- Cross-Domain JavaScript Source File Inclusion
- Server Leaks Information via "X-Powered-By"
- Server Leaks Version Information via "Server"
- Strict-Transport-Security Header Not Set (4)
- Timestamp Disclosure - Unix (5)

Cross Site Scripting (DOM Based)

URL:	http://localhost:5000/?message=%3Cscript%3Ealert(5397)%3C/script%3E
Risk:	High
Confidence:	High
Parameter:	message
Attack:	<script>alert(5397)</script>
Evidence:	
CWE ID:	79
WASC ID:	8
Source:	Active (40026 - Cross Site Scripting (DOM Based))
Input Vector:	URL Query String

Description:

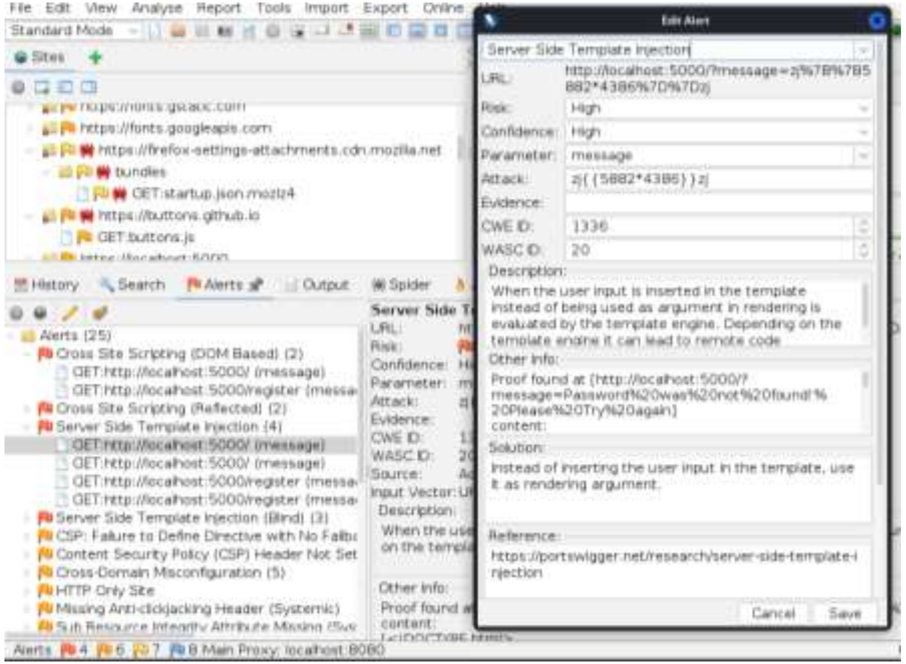
Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or

Other Info:

The following steps were done to trigger the DOM XSS:
With <PAYLOAD_1> as: %3Cscript%3Ealert(5397)%3C/script%3E

Alerts 4 6 7 8 Main Proxy: localhost:8080 Current Status 0 0 0 0 0 0 0 0 0 0

Zap tool analysis ssti



Alerts (25)
- Cross Site Scripting (DOM Based) (2)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/register (messa
- Cross Site Scripting (Reflected) (2)
- Server Side Template Injection (4)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/register (messa
  - GET:http://localhost:5000/register (messa
- Server Side Template Injection (Blind) (3)
- CSP: Failure to Define Directive with No Fallb
- Content Security Policy (CSP) Header Not Set
- Cross-Domain Misconfiguration (5)
- HTTP Only Site
- Missing Anti-clickjacking Header (Systemic)
- Sub Resource Integrity Attribute Missing (Sys

Edit Alert

Server Side Template Injection

URL: http://localhost:5000/?message=zj%7B%7B5882*4386%7D%7Dzj
Risk: High
Confidence: High
Parameter: message
Attack: zj{{5882*4386}}zj
Evidence:
CWE ID: 1336
WASC ID: 20

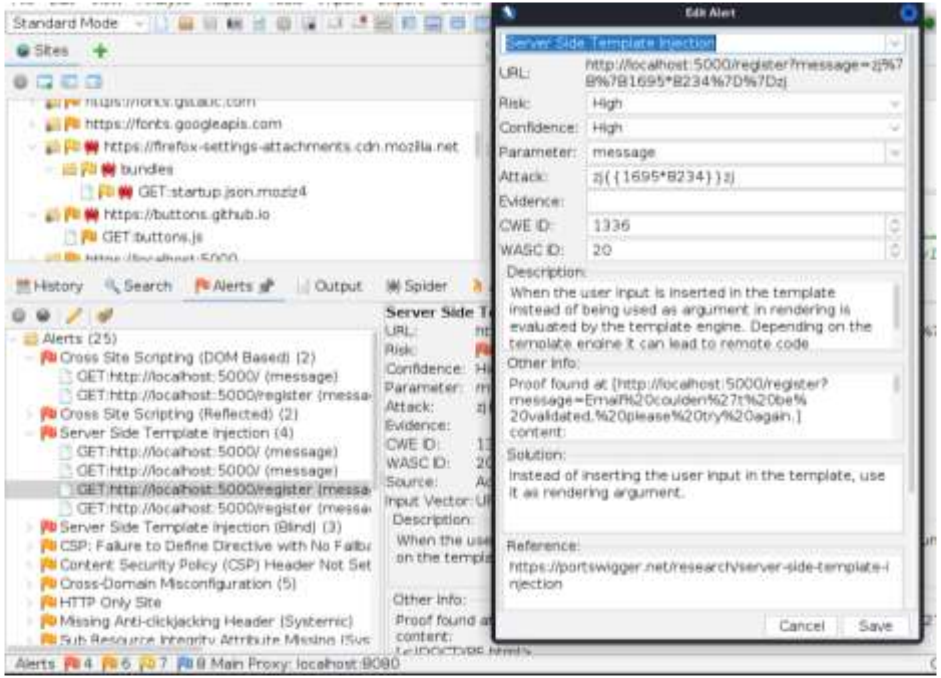
Description:
When the user input is inserted in the template instead of being used as argument in rendering is evaluated by the template engine. Depending on the template engine it can lead to remote code

Other Info:
Proof found at [http://localhost:5000/?message=Password%20was%20not%20found!%20Please%20Try%20again]
content:

Solution:
Instead of inserting the user input in the template, use it as rendering argument.

Reference:
https://portswigger.net/research/server-side-template-injection

Cancel Save



Edit Alert

Server Side Template Injection

URL: http://localhost:5000/register?message=zj%7B%7B1695*8234%7D%7Dzj
Risk: High
Confidence: High
Parameter: message
Attack: zj{{1695*8234}}zj
Evidence:
CWE ID: 1336
WASC ID: 20

Description:
When the user input is inserted in the template instead of being used as argument in rendering is evaluated by the template engine. Depending on the template engine it can lead to remote code

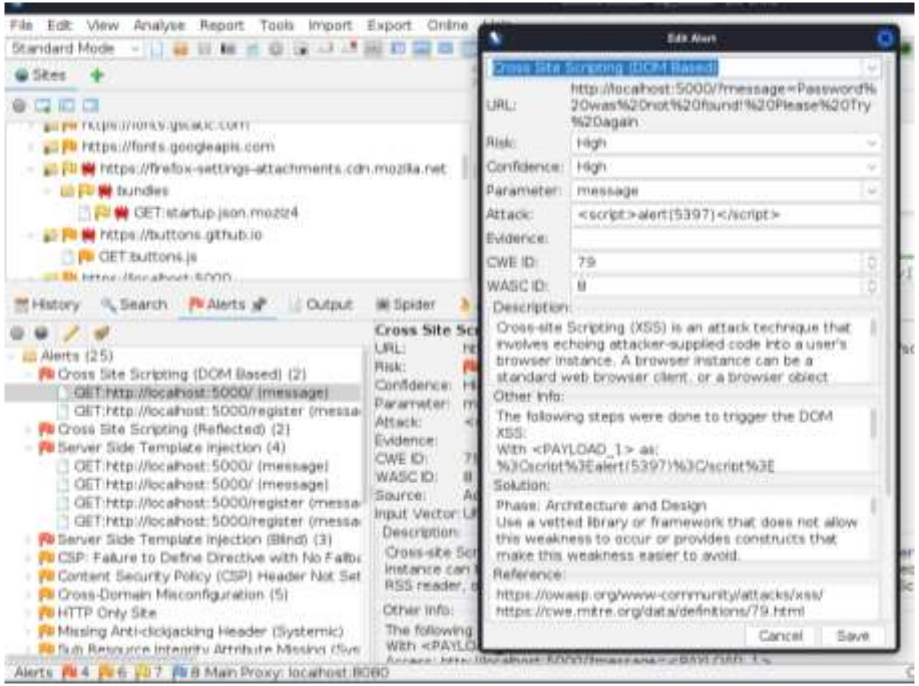
Other Info:
Proof found at [http://localhost:5000/register?message=Email%20coulden%27t%20be%20validated.%20please%20try%20again.]
content:

Solution:
Instead of inserting the user input in the template, use it as rendering argument.

Reference:
https://portswigger.net/research/server-side-template-injection

Cancel Save

Zap tool analysis xss



Alerts (25)
- Cross Site Scripting (DOM Based) (2)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/register (messa
- Cross Site Scripting (Reflected) (2)
- Server Side Template Injection (4)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/register (messa
  - GET:http://localhost:5000/register (messa
- Server Side Template Injection (Blind) (3)
- CSP: Failure to Define Directive with No Fallb
- Content Security Policy (CSP) Header Not Set
- Cross-Domain Misconfiguration (5)
- HTTP Only Site
- Missing Anti-clickjacking Header (Systemic)
- Sub Resource Integrity Attribute Missing (Sys

Edit Alert

Cross Site Scripting (DOM Based)

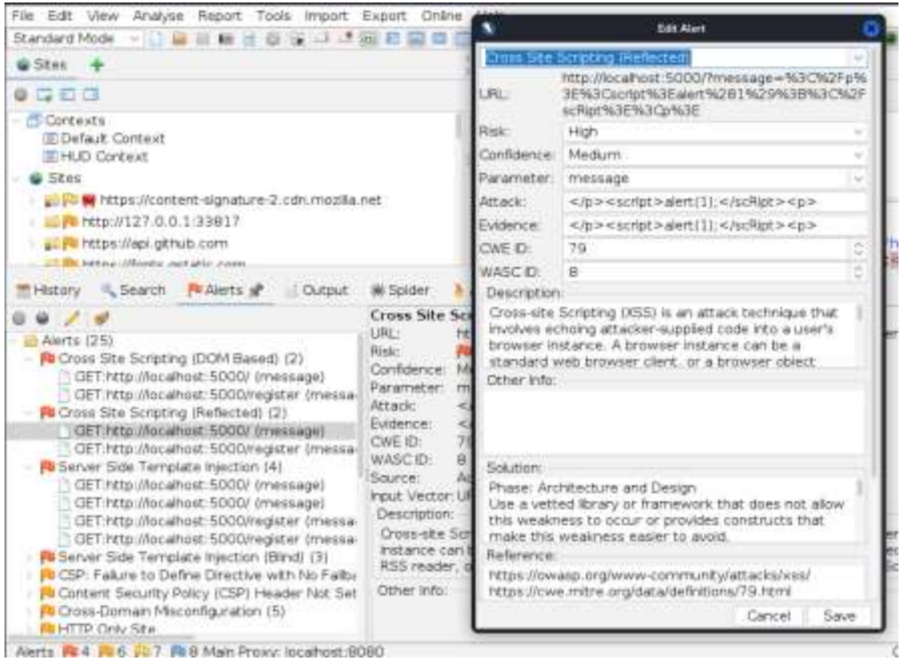
URL:	http://localhost:5000/?message=Password%20was%20not%20found!%20Please%20Try%20again
Risk:	High
Confidence:	High
Parameter:	message
Attack:	<script>alert(5397)</script>
Evidence:	
CWE ID:	79
WASC ID:	8

Description:
Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object

Other Info:
The following steps were done to trigger the DOM XSS:
With <PAYLOAD_1> as:
%3Cscript%3Ealert(5397)%3C/script%3E

Solution:
Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

Reference:
https://owasp.org/www-community/attacks/xss/
https://cwe.mitre.org/data/definitions/79.html



Sites
- https://content-signature-2.cdn.mozilla.net
- http://127.0.0.1:33817
- https://api.github.com

Alerts (25)
- Cross Site Scripting (DOM Based) (2)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/register (messa
- Cross Site Scripting (Reflected) (2)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/register (messa
- Server Side Template Injection (4)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/ (message)
  - GET:http://localhost:5000/register (messa
  - GET:http://localhost:5000/register (messa
- Server Side Template Injection (Blind) (3)
- CSP: Failure to Define Directive with No Fallb
- Content Security Policy (CSP) Header Not Set
- Cross-Domain Misconfiguration (5)
- HTTP Only Site

Edit Alert

Cross Site Scripting (Reflected)

URL:	http://localhost:5000/?message=%3C%2Fp%3E%3Cscript%3Ealert%281%29%3B%3C%2FscRipt%3E%3Cp%3E
Risk:	High
Confidence:	Medium
Parameter:	message
Attack:	</p><script>alert(1);</scRipt><p>
Evidence:	</p><script>alert(1);</scRipt><p>
CWE ID:	79
WASC ID:	8

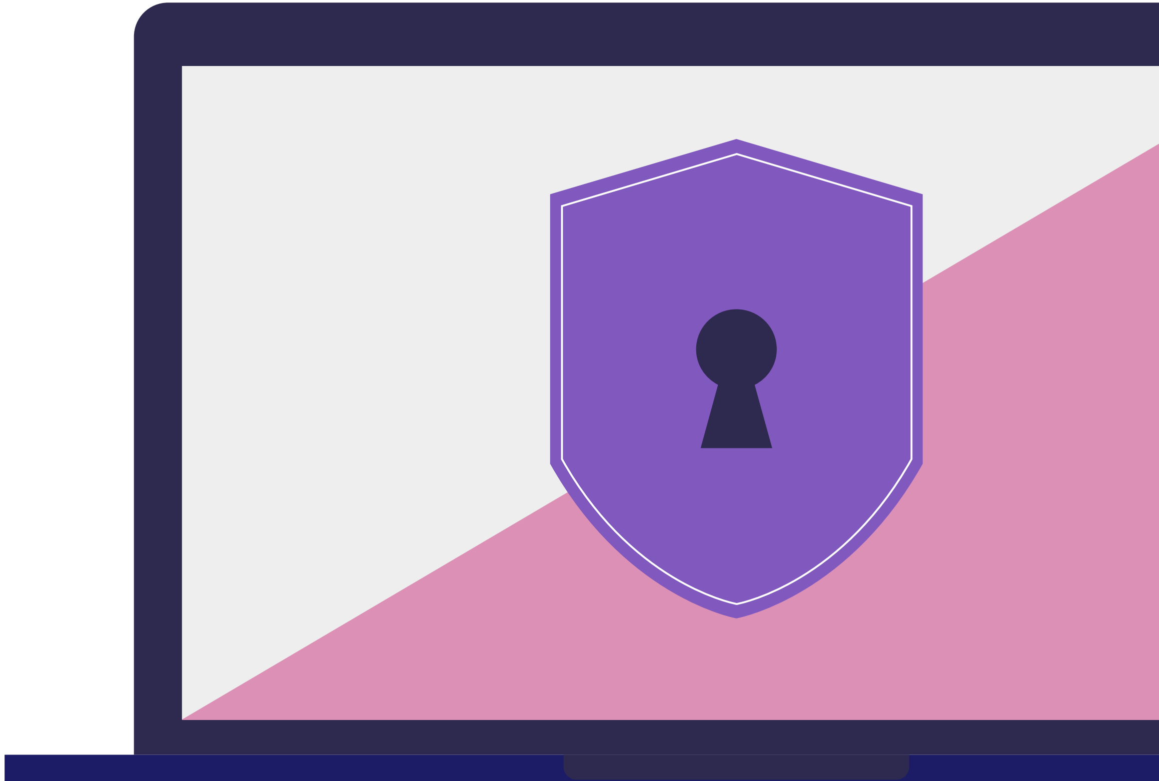
Description:
Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object

Other Info:

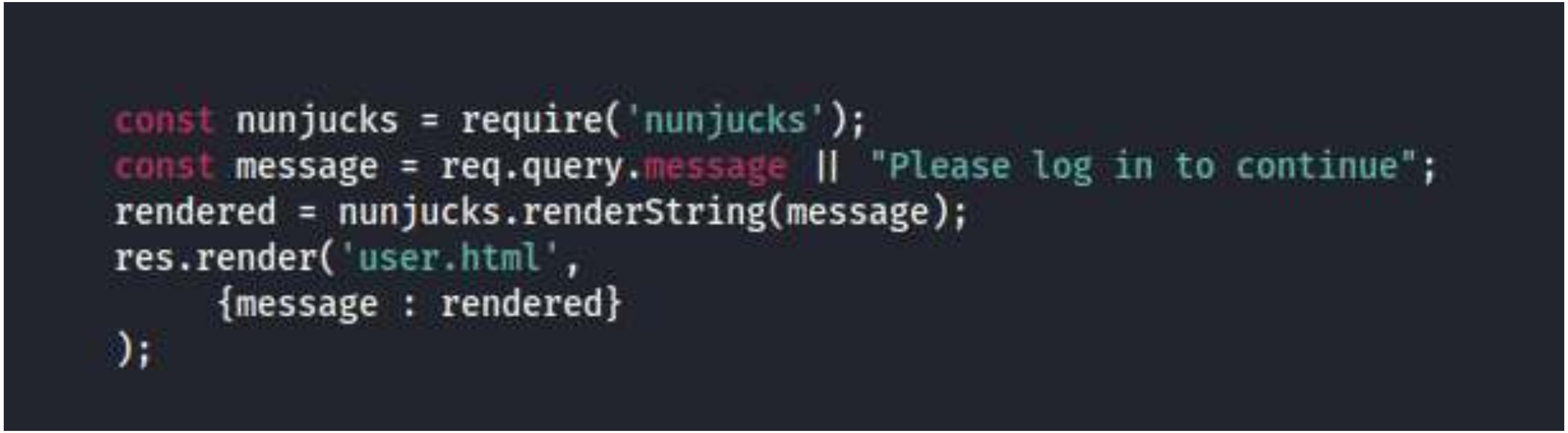
Solution:
Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

Reference:
https://owasp.org/www-community/attacks/xss/
https://cwe.mitre.org/data/definitions/79.html

XSS



vulnerable code



```
const nunjucks = require('nunjucks');
const message = req.query.message || "Please log in to continue";
rendered = nunjucks.renderString(message);
res.render('user.html',
     {message : rendered}
);
```

exploitation

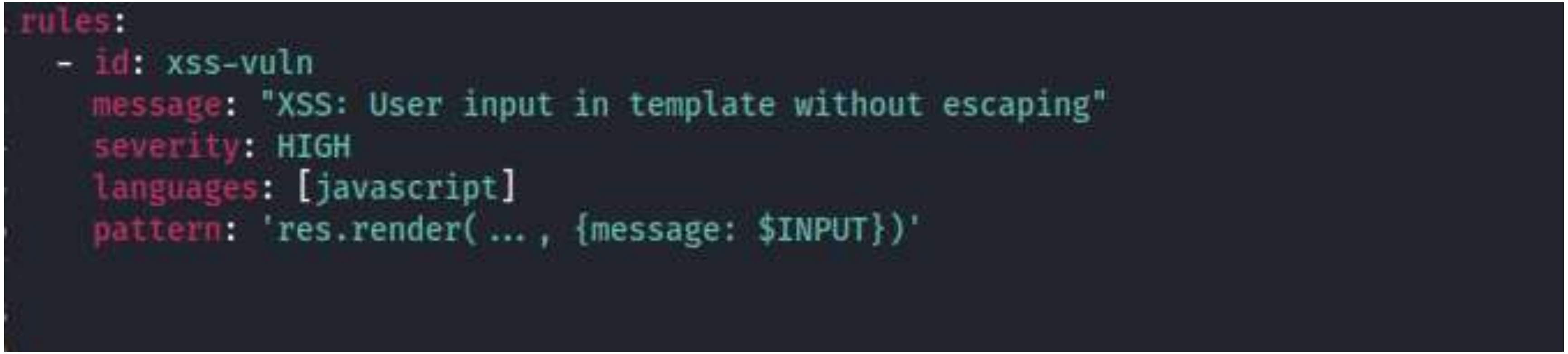
you could render any js code on the website using this url route:

- http://localhost:5000/?message=<script>......</script>

explanation

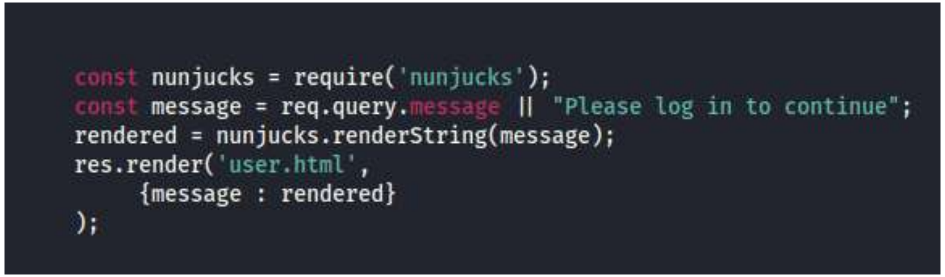
The application is vulnerable to both Reflected Cross-Site Scripting (XSS) and Server-Side Template Injection (SSTI). User-controlled input is passed directly to nunjucks.renderString() and rendered without sanitization. This allows attackers to execute arbitrary JavaScript in the browser and arbitrary commands on the server, leading to Remote Code Execution.

Semgrep rule

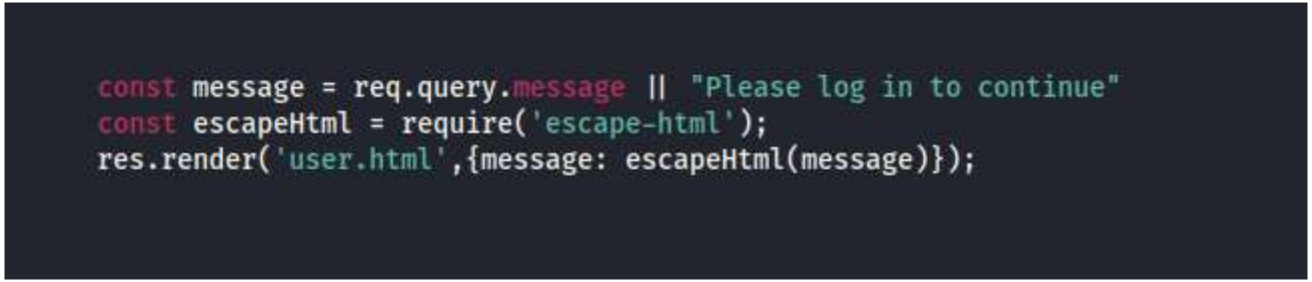


```
rules:
  - id: xss-vuln
    message: "XSS: User input in template without escaping"
    severity: HIGH
    languages: [javascript]
    pattern: 'res.render(... , {message: $INPUT})'
```

Fixes

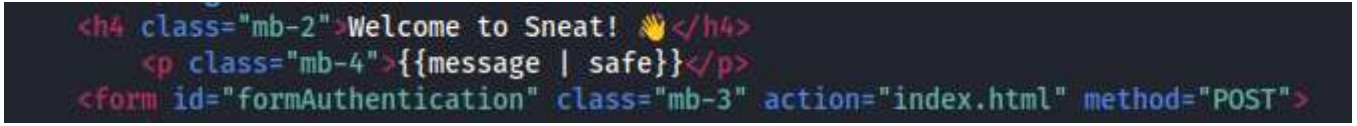


```
const nunjucks = require('nunjucks');
const message = req.query.message || "Please log in to continue";
rendered = nunjucks.renderString(message);
res.render('user.html',
     {message : rendered}
);
```

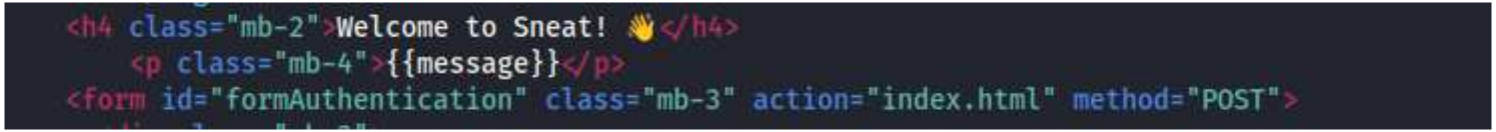


```
const message = req.query.message || "Please log in to continue"
const escapeHtml = require('escape-html');
res.render('user.html',{message: escapeHtml(message)});
```

Fixes

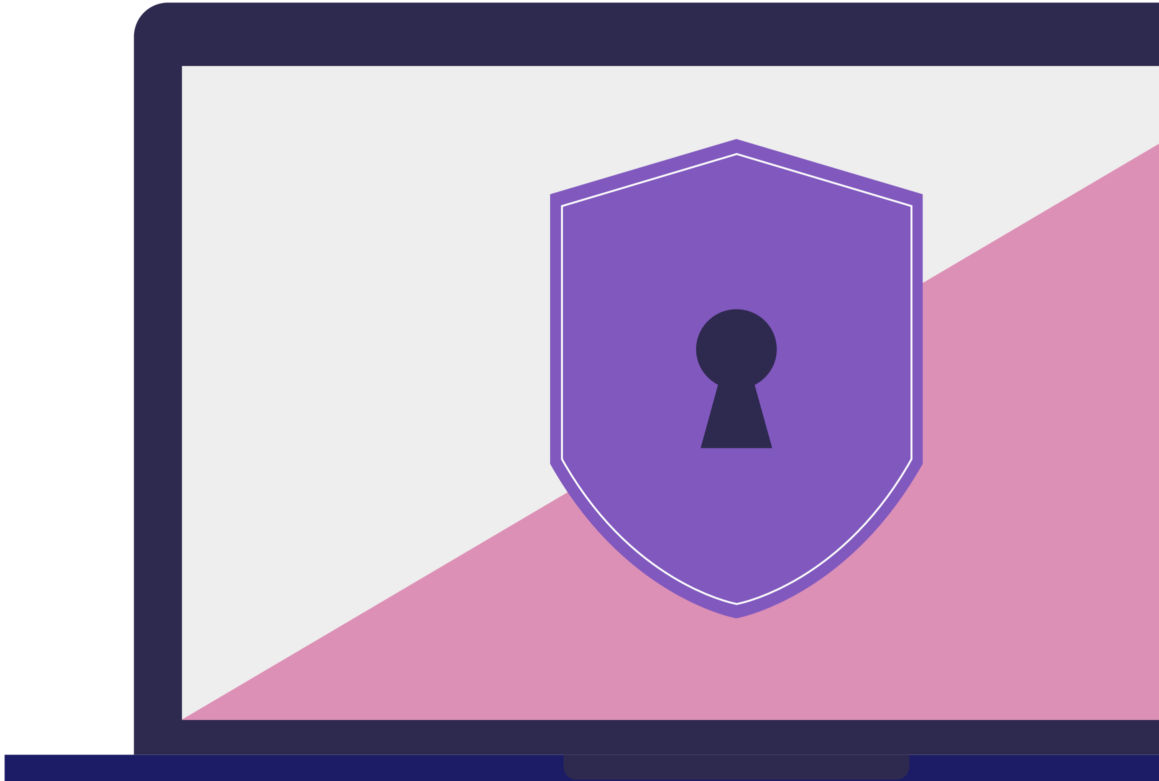


```
<h4 class="mb-2">Welcome to Sneat! 👋</h4>
    <p class="mb-4">{{message | safe}}</p>
<form id="formAuthentication" class="mb-3" action="index.html" method="POST">
```

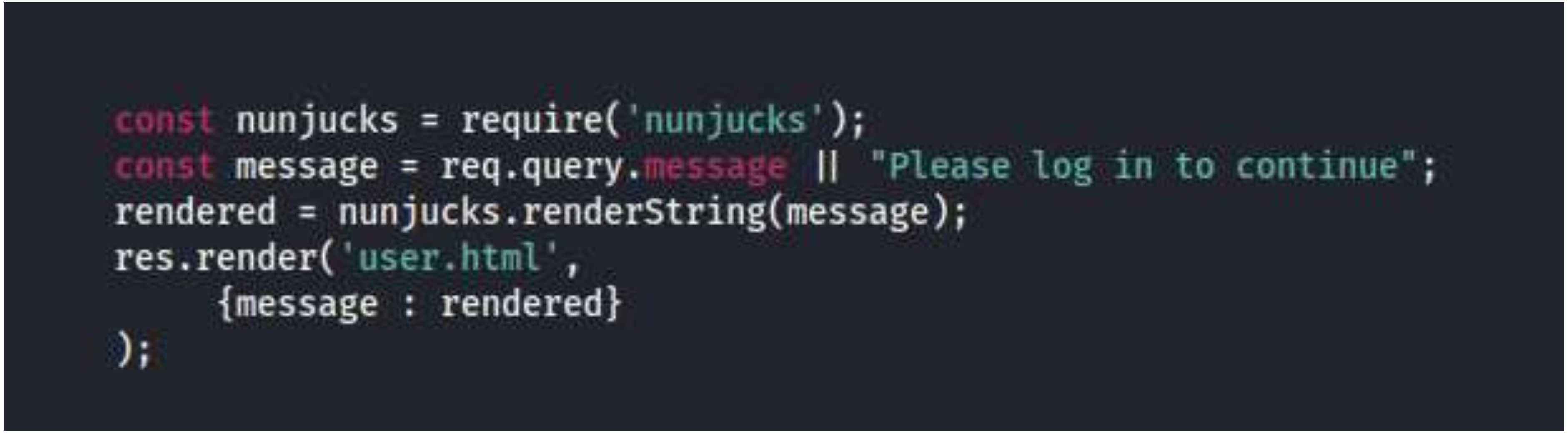


```
<h4 class="mb-2">Welcome to Sneat! 👋</h4>
    <p class="mb-4">{{message}}</p>
<form id="formAuthentication" class="mb-3" action="index.html" method="POST">
```

SSTI



vulnerable code



```
const nunjucks = require('nunjucks');
const message = req.query.message || "Please log in to continue";
rendered = nunjucks.renderString(message);
res.render('user.html',
     {message : rendered}
);
```

exploitation

you could render any nunjucks evaluates to execute it on server side using this url route:

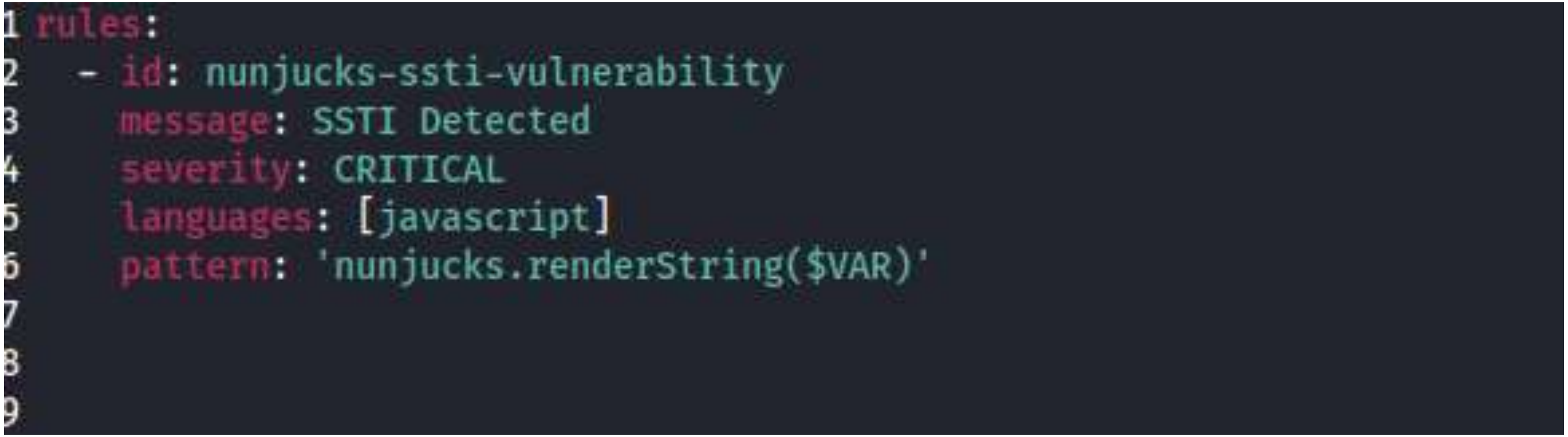
- http://localhost:5000/?message={{nunjucks evaluates}}

---

explanation

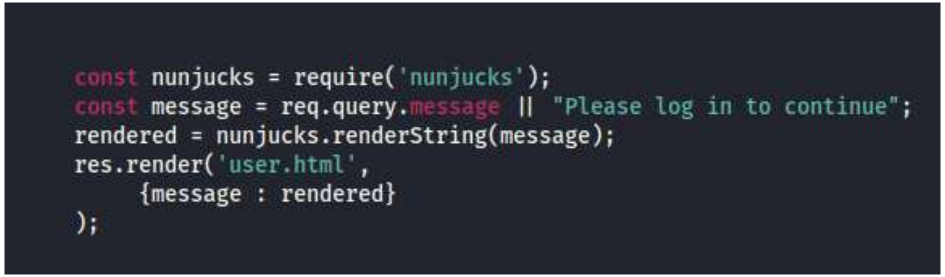
The application is vulnerable to both Reflected Cross-Site Scripting (XSS) and Server-Side Template Injection (SSTI). User-controlled input is passed directly to nunjucks.renderString() and rendered without sanitization. This allows attackers to execute arbitrary JavaScript in the browser and arbitrary commands on the server, leading to Remote Code Execution.

Semgrep rule

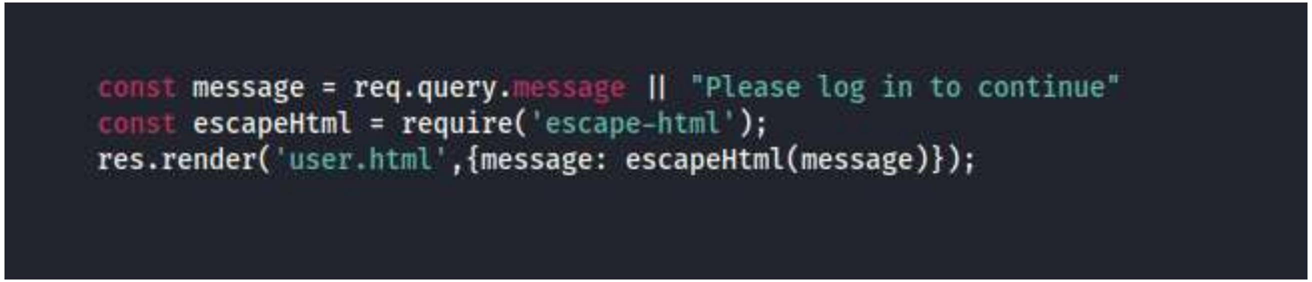


```yaml
rules:
  - id: nunjucks-ssti-vulnerability
    message: SSTI Detected
    severity: CRITICAL
    languages: [javascript]
    pattern: 'nunjucks.renderString($VAR)'
```

Fixes

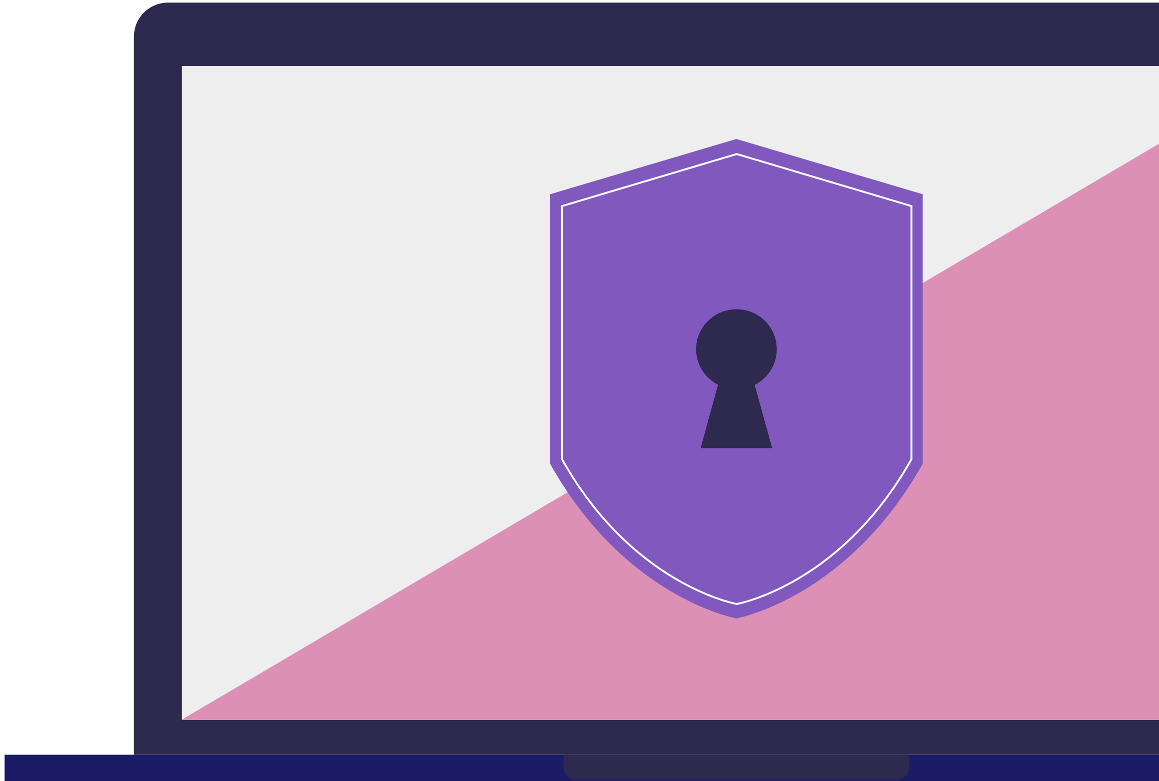


```
const nunjucks = require('nunjucks');
const message = req.query.message || "Please log in to continue";
rendered = nunjucks.renderString(message);
res.render('user.html',
     {message : rendered}
);
```

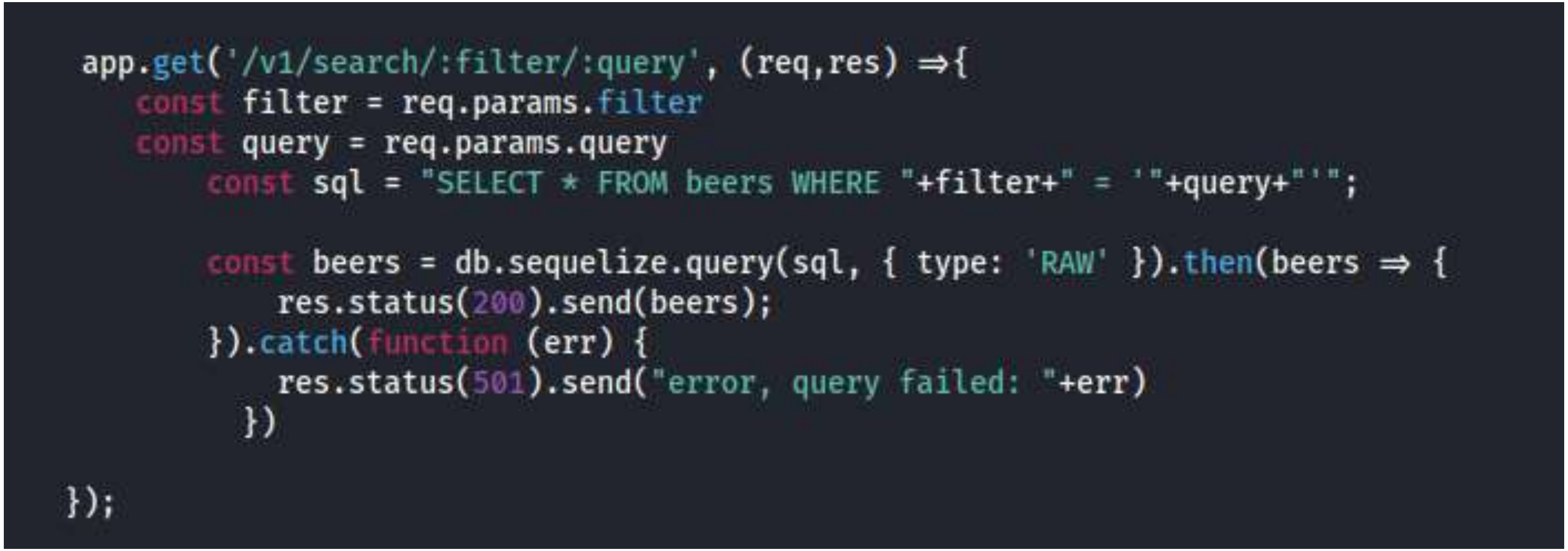


```
const message = req.query.message || "Please log in to continue"
const escapeHtml = require('escape-html');
res.render('user.html',{message: escapeHtml(message)});
```

SQL injection



vulnerable code



```
 app.get('/v1/search/:filter/:query', (req,res) =>{
    const filter = req.params.filter
    const query = req.params.query
        const sql = "SELECT * FROM beers WHERE "+filter+" = '"+query+"'";

        const beers = db.sequelize.query(sql, { type: 'RAW' }).then(beers => {
            res.status(200).send(beers);
        }).catch(function (err) {
            res.status(501).send("error, query failed: "+err)
          })

});
```

exploitation

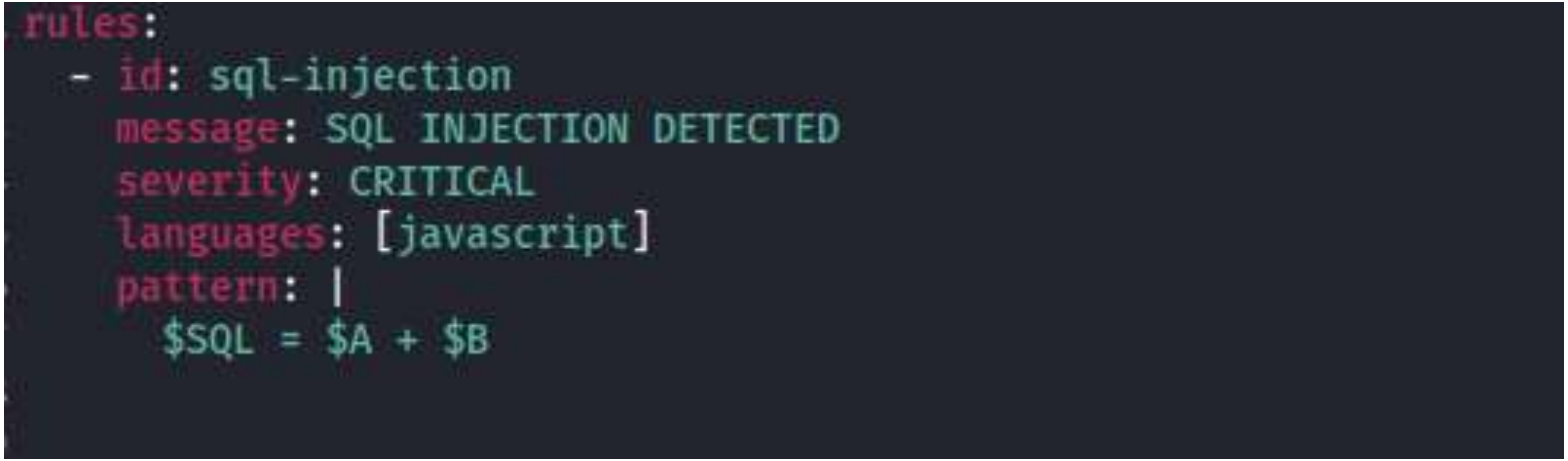
```
sqlmap -u "http://localhost:5000/v1/search/id/1*" --batch --dump-all
```

---

explanation

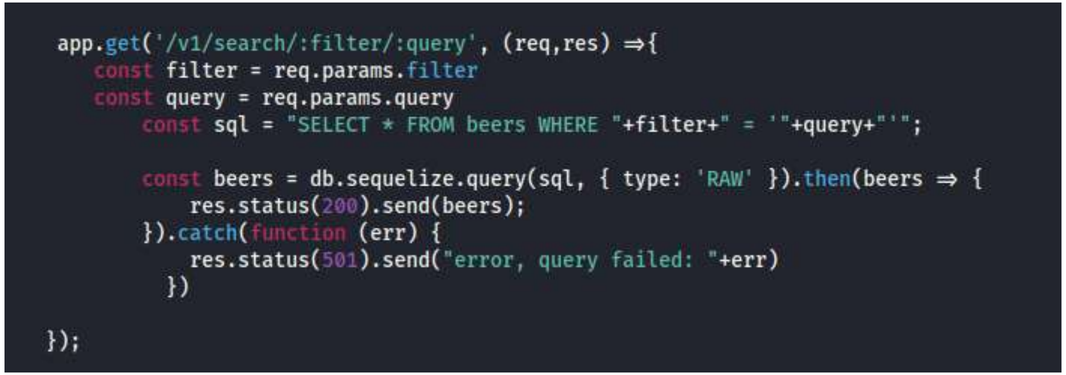
User controls SQL structure because Query is built via string concatenation, No sanitization and No parameter binding

Semgrep rule



```yaml
rules:
  - id: sql-injection
    message: SQL INJECTION DETECTED
    severity: CRITICAL
    languages: [javascript]
    pattern: |
      $SQL = $A + $B
```

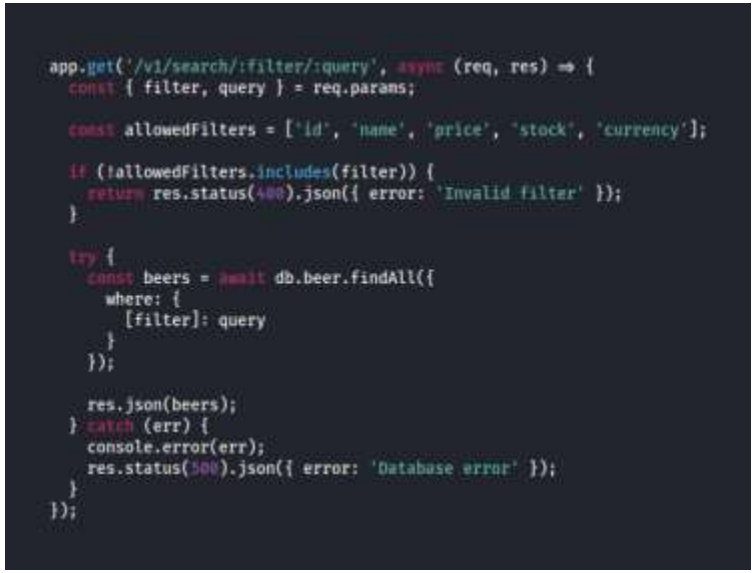
fixes



```
app.get('/v1/search/:filter/:query', (req,res) ⇒{
   const filter = req.params.filter
   const query = req.params.query
       const sql = "SELECT * FROM beers WHERE "+filter+" = '"+query+"'";

       const beers = db.sequelize.query(sql, { type: 'RAW' }).then(beers ⇒ {
           res.status(200).send(beers);
       }).catch(function (err) {
           res.status(501).send("error, query failed: "+err)
         })

});
```



```
app.get('/v1/search/:filter/:query', async (req, res) ⇒ {
  const { filter, query } = req.params;

  const allowedFilters = ['id', 'name', 'price', 'stock', 'currency'];

  if (!allowedFilters.includes(filter)) {
    return res.status(400).json({ error: 'Invalid filter' });
  }

  try {
    const beers = await db.beer.findAll({
      where: {
        [filter]: query
      }
    });

    res.json(beers);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Database error' });
  }
});
```