EXPERIMENT NO. 13

AIM: Cross-Site scripting prevention

THEORY:

Cross-Site Scripting (XSS) is a type of security vulnerability found in web applications where attackers inject malicious scripts into web pages viewed by other users. XSS typically targets the client-side, enabling attackers to execute arbitrary scripts in a user's browser, often stealing cookies, session tokens, or redirecting users to malicious sites.

To prevent XSS, several mitigation strategies can be employed:

1. Content Security Policy (CSP)

CSP is a security feature that acts as a whitelist for what types of content can be loaded on a web page. By setting CSP headers, you define which sources are allowed to provide resources like scripts, images, and styles. If the browser detects content being loaded from an unauthorized source, it blocks the request. This reduces the risk of script injection attacks by preventing unauthorized scripts from executing.

For example, you can restrict scripts to be loaded only from trusted domains, disallow inline scripts, and disable `eval()` functions. While CSP is not a full-proof defense against all XSS attacks, it significantly raises the barrier for attackers.

2. Escape Output

Escaping output means transforming special characters in user inputs (like <, >, `"`, `'`, etc.) into their HTML entity equivalents. This prevents browsers from interpreting them as executable code. For instance, if a user inputs a script, instead of being executed, it will be displayed as plain text, nullifying the attack.

Different types of data (HTML, JavaScript, CSS) require different escaping mechanisms. Ensuring that output is properly escaped based on its context (HTML, URL, etc.) is a fundamental layer of defense against XSS.

3. Sanitize HTML

Sanitizing involves removing or altering unsafe portions of user inputs before they are rendered on a web page. Tools like DOMPurify, widely used in React apps, help strip out any unsafe elements (like `<script>` tags) or attributes that could lead to XSS vulnerabilities. By sanitizing inputs, you allow only safe HTML content to be processed by the browser.

Libraries like DOMPurify are vital because they are constantly updated to address newly discovered threats. While escaping focuses on avoiding injection by altering output, sanitization ensures that the content passed through user inputs is scrubbed clean of potential threats.

By combining these techniques—CSP for external script management, escaping for safe output rendering, and sanitization for clean inputs—you can effectively mitigate XSS vulnerabilities in web applications.

Code :

Frontend Code (ReactJS)

```
import React, { useState, useEffect } from
'react';
import axios from 'axios';
import DOMPurify from 'dompurify';
import './Comments.css';

const CommentForm = ({ onCommentAdded })
=> {
   const [comment, setComment] =
useState('');

   const handleSubmit = async (e) => {
      e.preventDefault();
      const sanitizedComment =
DOMPurify.sanitize(comment);
      try {
         await axios.post('http://localhost:4000/
api/comments', { content:
sanitizedComment });
         setComment('');
         onCommentAdded();
      } catch (error) {
         alert('An error occurred while
submitting the comment: ' + error.message);
      }
   };
```

```
  return (
      <form className="comment-
form" onSubmit={handleSubmit}>
         <textarea
            className="comment-
textarea"
            value={comment}
            onChange={(e) =>
setComment(e.target.value)}
            placeholder="Enter your
comment"
            required
         />
         <button
className="comment-submit-
button" type="submit">Submit</
button>
      </form>
   );
};
```

```
      const CommentList = ({ comments })
=> {
  return (
    <div className="comment-list-
container">
      <h2 className="comment-
title">Comments</h2>
      {comments.map((comment) => (
        <div
          className="comment-item"
          key={comment._id}

dangerouslySetInnerHTML={{ __html:
DOMPurify.sanitize(comment.content) }}
        />
      ))}
    </div>
  );
};
const Comments = () => {
  const [comments, setComments] =
useState([]);

  const fetchComments = async () => {
    try {
      const response = await
axios.get('http://localhost:4000/api/
comments');
      setComments(response.data);
    } catch (error) {
      alert('An error occurred while fetching
comments: ' + error.message);
    }
  };

  useEffect(() => {
    fetchComments();
  }, []);
```

```
return (
        <div className="comment-
container">
                  <CommentForm
onCommentAdded={fetchComments}
/>
                   <CommentList
comments={comments} />
    </div>
  );
};

export default Comments;
```

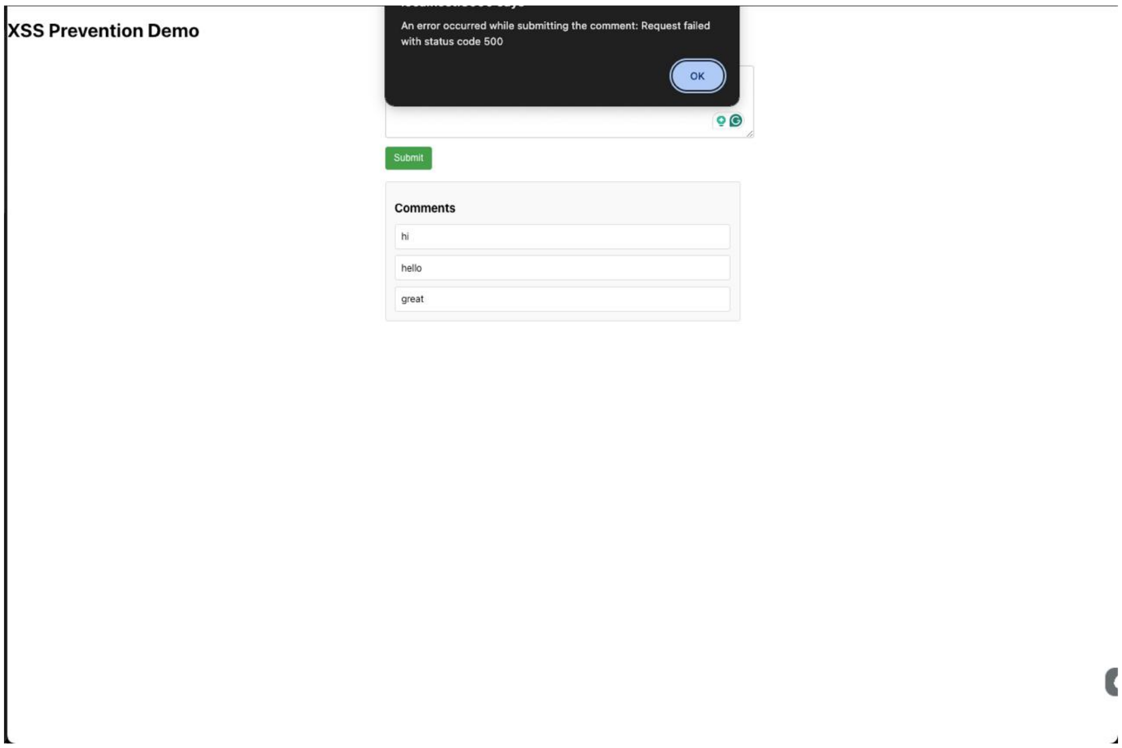
Backend Code (Node.js + Express) :

```
backend: const express =
require('express');
const mongoose =
require('mongoose');
const helmet =
require('helmet');
const cors = require('cors');

const app = express();
app.use(express.json());
app.use(helmet());
app.use(cors());


app.use(

helmet.contentSecurityPolicy({
      directives: {
          defaultSrc: ["'self'"],
          scriptSrc: ["'self'",
"https://trusted-cdn.com"],
          objectSrc: ["'none'"],

upgradeInsecureRequests: [],
      },
   })
);
```

```
mongoose.connect('mongodb://localhost:27017/xss-
demo', {
   useNewUrlParser: true,
   useUnifiedTopology: true,
}).then(() => console.log('MongoDB
connected')).catch(err => console.log(err));


const CommentSchema = new mongoose.Schema({
   content: { type: String, required: true },
});
const Comment = mongoose.model('Comment',
CommentSchema);

app.post('/api/comments', async (req, res) => {
   try {
      const comment = new Comment(req.body);
      await comment.save();
      res.status(201).json(comment);
   } catch (err) {
      res.status(500).json({ error: 'Failed to add
comment' });
   }
});
```

```
app.get('/api/comments', async
(req, res) => {
    try {
        const comments = await
Comment.find();
        res.json(comments);
    } catch (err) {

res.status(500).json({ error:
'Failed to fetch comments' });
    }
});


const PORT = 4000;
app.listen(PORT, () =>
console.log(Server running on
port ${PORT}));
```

Output :



XSS Prevention Demo

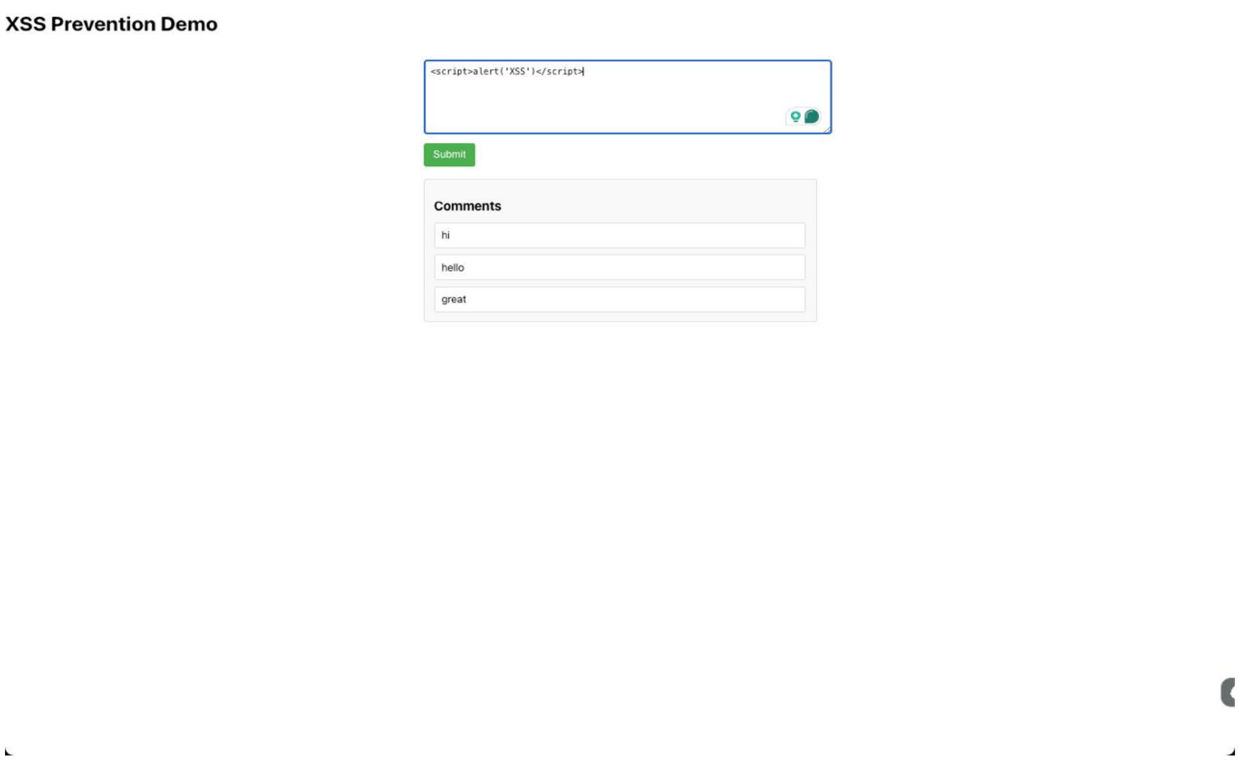
An error occurred while submitting the comment: Request failed with status code 500

Comments

hi

hello

great



XSS Prevention Demo

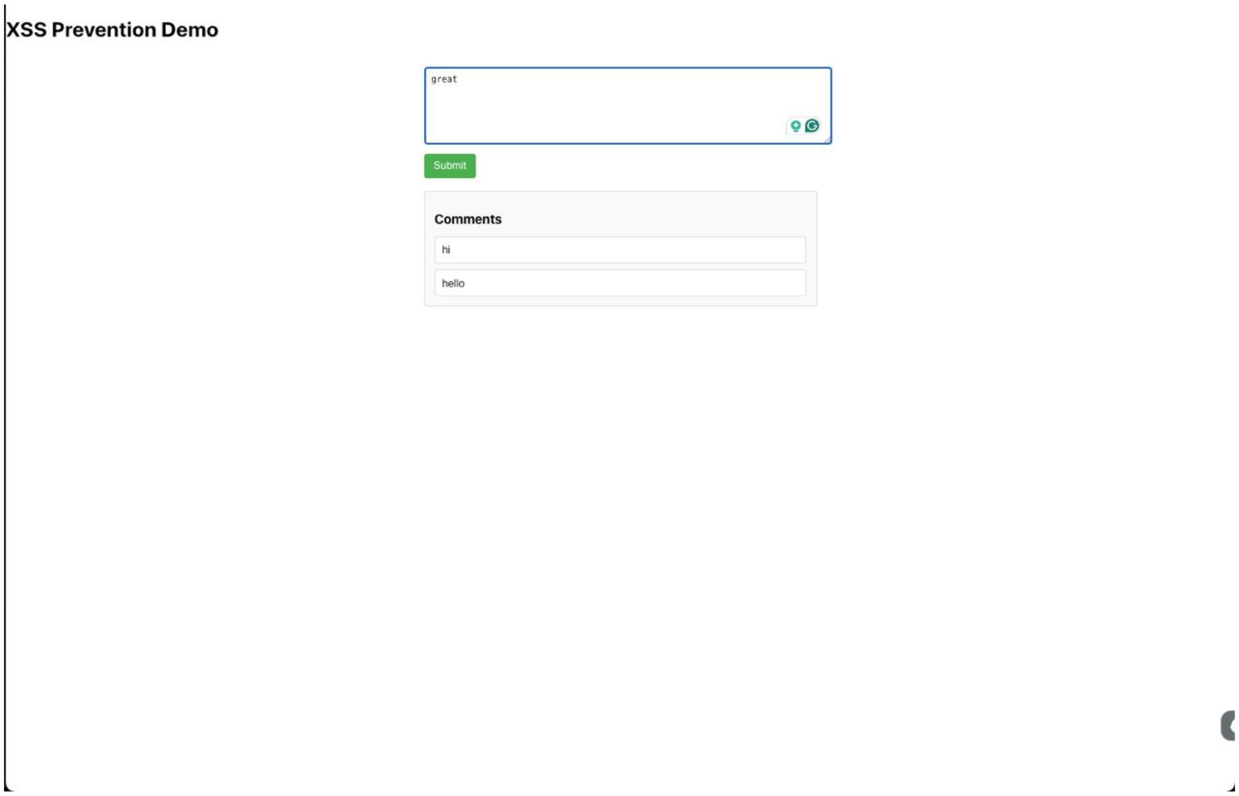
```
<script>alert('XSS')</script>
```

Comments

hi

hello

great



XSS Prevention Demo

```
great
```

Comments

hi

hello

Conclusion :

In conclusion, **Cross-Site Scripting (XSS)** is a significant security threat that can compromise the integrity and security of web applications by allowing malicious scripts to execute in users' browsers. To mitigate this threat:

- **Content Security Policy (CSP)** limits the sources of scripts and content, reducing the likelihood of unauthorized code execution.
- **Escaping Output** ensures that any special characters in user inputs are rendered as plain text, preventing them from being executed as code.
- **Sanitizing HTML** removes or neutralizes unsafe code in user-generated content, ensuring only safe content is rendered.

By integrating these strategies together, developers can establish a robust defense against XSS attacks, safeguarding both their applications and users.