**Lab Assignment 3**

1. **Write a script to log the current URL of the page using the window.location property. Extract and log any query parameters. Discuss how query parameters might be exploited in a phishing attack.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Log URL</title>

</head>

<body>

    Java Script

  <script>

    console.log("Current URL:", window.location.href);

    const queryParams = new URLSearchParams(window.location.search);

    queryParams.forEach((value, key) => {

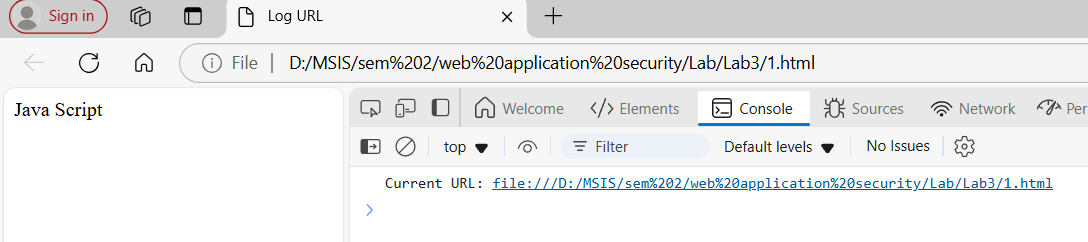
      console.log(`${key}: ${value}`);

    });

  </script>

</body>

</html>



* **Credential Harvesting**: Attackers might craft a URL with deceptive query parameters like ?username=admin&password=12345. A malicious site might display a fake login page and trick users into submitting their credentials, thinking they are logging in to a legitimate site.
* **URL Spoofing**: A phishing URL could look like https://example.com/login?redirect=https://bank.com. The query parameter (redirect) could be used to redirect a user to a malicious page after they log in.
* **Session Hijacking**: If a website uses query parameters to store session tokens or user IDs (e.g., ?session=abcdef12345), attackers might craft URLs with manipulated session tokens to hijack active user sessions.
* **Cross-Site Scripting (XSS)**: If user input is not properly sanitized and included in query parameters, attackers could inject JavaScript into the URL, leading to potential XSS vulnerabilities. For example, a URL like https://example.com/?search=<script>alert('Hacked')</script> could execute malicious scripts if the site doesn't properly sanitize the input.

1. **Write a script to log the hostname, protocol, and pathname of the current page using window.location. Explain how attackers might manipulate these properties to perform malicious redirects.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Log Location Details</title>

</head>

<body>

  <h2>Logged Information:</h2>

  <script>

    // Log the details to the console

    console.log("Hostname:", window.location.hostname);

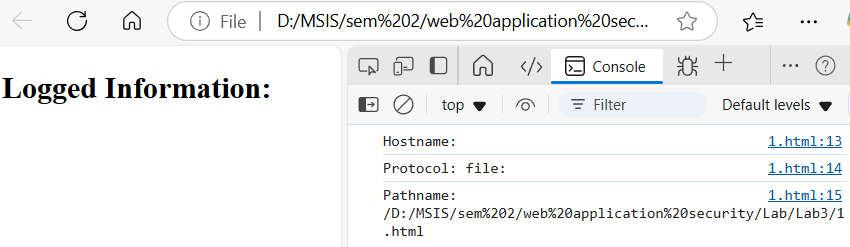
    console.log("Protocol:", window.location.protocol);

    console.log("Pathname:", window.location.pathname);

  </script>

</body>

</html>



The `window.location` object in JavaScript provides essential information about the current URL, including the hostname, protocol, and pathname. While this functionality is beneficial for web developers, it can also be exploited by attackers to perform malicious redirects. For instance, an attacker might manipulate `window.location` to redirect users to a fraudulent site that mimics a legitimate one, such as a banking login page, thereby tricking users into entering their sensitive information. This manipulation can occur through JavaScript injection, particularly in cases of cross-site scripting (XSS), where an attacker injects malicious code into a web page to alter the location. Additionally, attackers may craft deceptive URLs that appear trustworthy but lead to harmful sites. Therefore, while `window.location` serves a vital role in web development, it poses security risks that necessitate robust protective measures, such as input validation and content security policies, to safeguard users from potential phishing attacks and malware infections.

1. **Write a script that uses window.location to redirect users to another webpage. Add a condition to restrict redirects only to trusted domains. Discuss the implications of open redirects in phishing attacks.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Page Visit Count</title>

</head>

<body>

    Java Script

  <script>

    console.log("Pages visited in this session:", window.history.length);

  </script><!DOCTYPE html>

  <html lang="en">

  <head>

      <meta charset="UTF-8">

      <meta name="viewport" content="width=device-width, initial-scale=1.0">

      <title>Redirect with Trusted Domains</title>

      <script>

          // List of trusted domains

          const trustedDomains = [

              'example.com',

              'trusted-site.com',

              'another-safe-site.org'

          ];

          // Function to redirect to a trusted URL

          function redirectToTrustedSite(url) {

              try {

                  // Parse the URL to get the domain

                  const urlObj = new URL(url);

                  const domain = urlObj.hostname;

                  // Check if the domain is trusted

                  if (trustedDomains.includes(domain)) {

                      // Redirect to the URL

                      window.location.href = url;

                  } else {

                      // Display a warning message or handle it accordingly

                      alert('Redirect blocked: Untrusted domain');

                  }

              } catch (error) {

                  // Handle invalid URL format

                  alert('Invalid URL');

              }

          }

      </script>

  </head>

  <body>

      <h1>Redirect Demo with Trusted Domains</h1>

      <p>Click the buttons below to test redirects:</p>

      <button onclick="redirectToTrustedSite('https://trusted-site.com')">Go to Trusted Site</button>

      <button onclick="redirectToTrustedSite('https://untrusted-site.com')">Go to Untrusted Site</button>

      <button onclick="redirectToTrustedSite('https://another-safe-site.org')">Go to Another Safe Site</button>

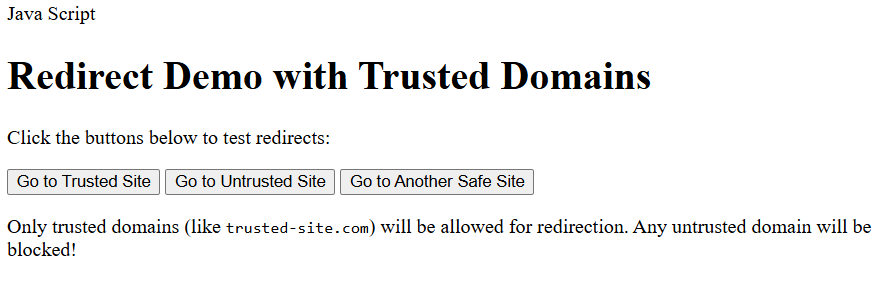
      <p>Only trusted domains (like <code>trusted-site.com</code>) will be allowed for redirection. Any untrusted domain will be blocked!</p>

  </body>

  </html>

</body>

</html>



Open redirects are a significant threat in phishing attacks, as they allow attackers to exploit legitimate URLs to mislead users. When a website has an open redirect vulnerability, malicious actors can create links that appear to lead to trusted sites but actually redirect victims to fraudulent pages designed to steal sensitive information, such as login credentials. This not only compromises individual security but also damages the reputation of the legitimate site being exploited, as users may lose trust in its integrity. To combat these risks, organizations must prioritize securing their web applications against open redirects, ensuring that users can navigate the web safely and avoid falling victim to these deceptive schemes.

1. **Write a script to log the number of pages visited in the current session using window.history. Discuss potential privacy concerns if malicious scripts attempt to track browser history.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Session Visit Counter</title>

</head>

<body>

    <h1>Page Visit Tracker</h1>

    <p><strong>Number of times you've visited this page in this session:</strong> <span id="visitCount"></span></p>

    <script>

        // Check if "pageVisits" exists in sessionStorage

        let visitCount = sessionStorage.getItem("pageVisits");

        // If it doesn't exist, initialize it to 1

        if (!visitCount) {

            visitCount = 1;

        } else {

            // If it exists, increment it

            visitCount = parseInt(visitCount) + 1;

        }

        // Store the updated count back in sessionStorage

        sessionStorage.setItem("pageVisits", visitCount);

        // Display the visit count

        document.getElementById("visitCount").textContent = visitCount;

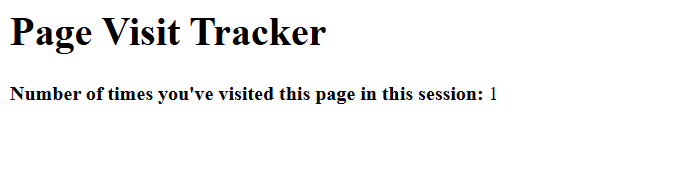
        // Log the count to the console

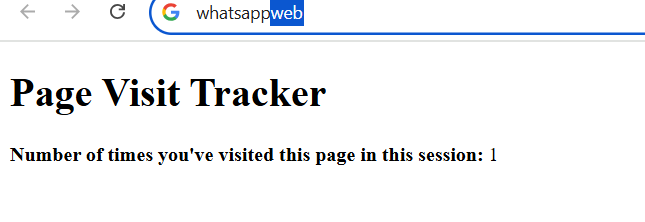
        console.log("Number of times this page has been visited in this session:", visitCount);

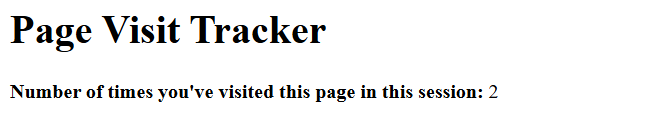
    </script>

</body>

</html>





****

* Session Storage: The script uses sessionStorage to keep track of the number of pages visited during the current session. sessionStorage stores data only for the duration of the page session (i.e., until the browser or tab is closed).
* Tracking Visits: On each page load (window.onload), the page visit count is incremented, and the new count is stored in sessionStorage.
* Console Logging: After incrementing the visit count, it logs the current count to the browser console so you can monitor how many times the user has visited pages during the session.

1. **Write a script using window.navigator to display the user agent string. Modify the script to identify and log whether the browser is Google Chrome, Mozilla Firefox, or an automation tool.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Browser Detection</title>

</head>

<body>

    <h1>Browser Information</h1>

    <p><strong>User Agent:</strong> <span id="userAgent"></span></p>

    <p><strong>Detected Browser:</strong> <span id="browser"></span></p>

    <script>

        // Get the user agent string

        const userAgent = window.navigator.userAgent;

        // Function to detect browser

        function detectBrowser(ua) {

            if (/chrome|chromium|crios/i.test(ua) && !/edg/i.test(ua)) {

                return "Google Chrome";

            } else if (/firefox|fxios/i.test(ua)) {

                return "Mozilla Firefox";

            } else if (/headless|bot|crawler|spider|phantomjs|selenium/i.test(ua)) {

                return "Automation Tool Detected";

            } else {

                return "Unknown Browser";

            }

        }

        // Get browser name

        const browserName = detectBrowser(userAgent);

        // Log results to console

        console.log("User Agent:", userAgent);

        console.log("Detected Browser:", browserName);

        // Display on webpage

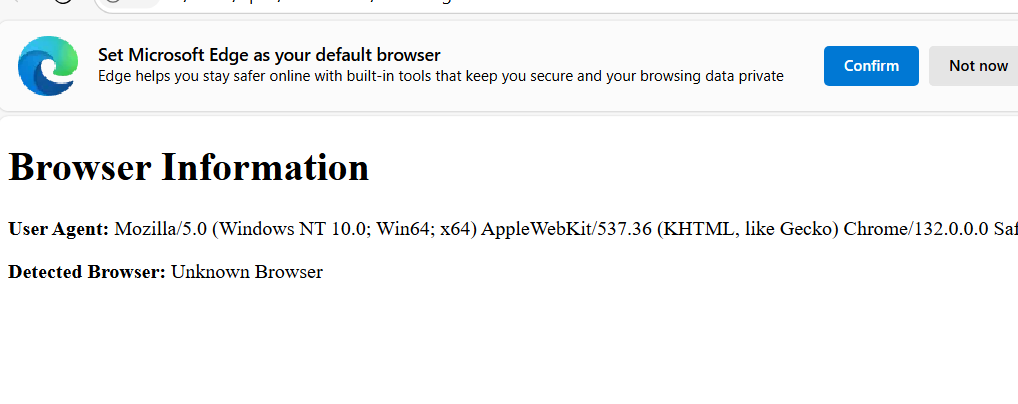
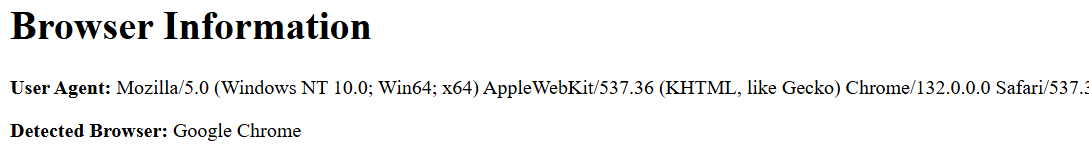
        document.getElementById("userAgent").textContent = userAgent;

        document.getElementById("browser").textContent = browserName;

    </script>

</body>

</html>



1. **Write a script to log the user's screen width and height using window.screen. Explain how attackers might use screen dimensions to craft phishing attacks.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Screen Size Logger</title>

</head>

<body>

    <h1>Screen Size Information</h1>

    <p><strong>Screen Width:</strong> <span id="screenWidth"></span> pixels</p>

    <p><strong>Screen Height:</strong> <span id="screenHeight"></span> pixels</p>

    <script>

        // Get screen width and height

        const screenWidth = window.screen.width;

        const screenHeight = window.screen.height;

        // Log to console

        console.log("Screen Width:", screenWidth);

        console.log("Screen Height:", screenHeight);

        // Display on webpage

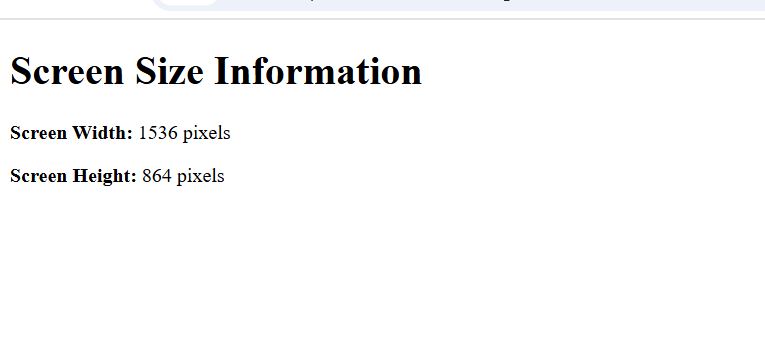
        document.getElementById("screenWidth").textContent = screenWidth;

        document.getElementById("screenHeight").textContent = screenHeight;

    </script>

</body>

</html>



Attackers can exploit screen dimensions in phishing attacks by designing malicious websites that are tailored to specific screen sizes, ensuring they appear legitimate on the victim's device. By manipulating the layout and elements based on the user's screen resolution, attackers can create a more convincing interface that mimics trusted sites, increasing the likelihood of user interaction and data theft. This tactic can involve adjusting the size of pop-up windows or forms to fit seamlessly within the user's display, making it harder for victims to recognize that they are not interacting with a genuine site. Additionally, attackers may use responsive design techniques to ensure that their phishing pages look professional and trustworthy, regardless of the device being used. By leveraging these visual cues, they can effectively lower the user's guard, leading to higher success rates in capturing sensitive information like login credentials or personal data.

1. **Write a script to open a new popup window using window.open. Discuss the risks associated with popup-based attacks and suggest mitigation strategies.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Popup Window Example</title>

</head>

<body>

    <h1>Open a Popup Window</h1>

    <button onclick="openPopup()">Click here</button>

    <script>

        function openPopup() {

            // Open a new window with specific properties

            const popupWindow = window.open(

                "https://example.com",  // URL to open in the popup

                "popupWindow",          // Name of the window

                "width=600,height=400,scrollbars=yes,resizable=yes"  // Window properties

            );

            // Check if the window opened successfully

            if (popupWindow) {

                console.log("Popup window opened successfully");

            } else {

                console.log("Failed to open the popup window.");

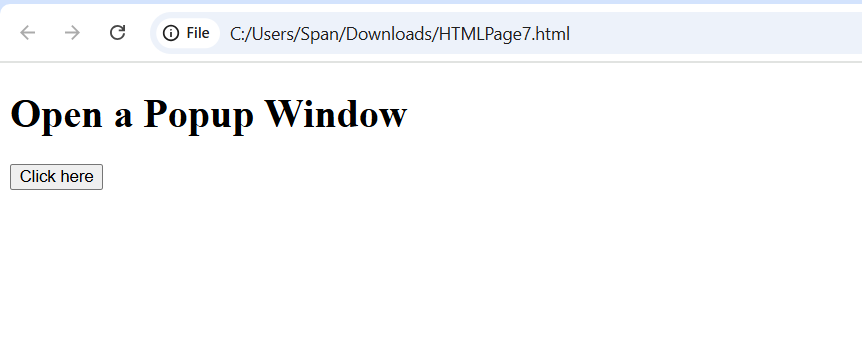
            }

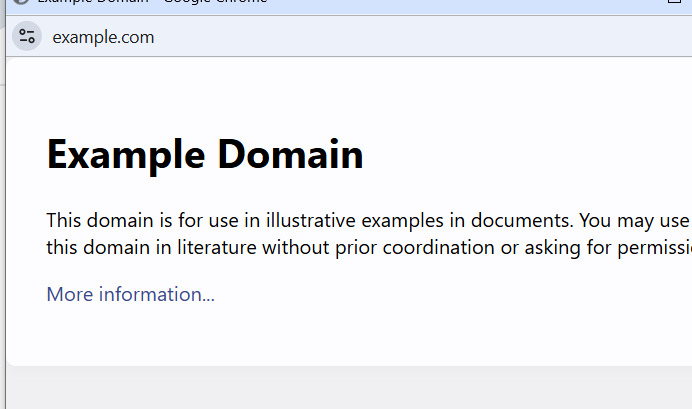
        }

    </script>

</body>

</html>





Popup-based attacks pose significant risks, including the potential for users to inadvertently download malware, disclose sensitive information, or fall victim to phishing schemes. These attacks often exploit the trust users place in familiar interfaces, making it crucial to implement mitigation strategies such as using robust popup blockers, educating users about recognizing suspicious popups, and ensuring that security software is up-to-date to detect and block malicious content. Additionally, organizations can enhance their defenses by regularly updating their web browsers and plugins, as outdated software can be more susceptible to exploitation. Implementing strict access controls and monitoring user behavior can also help identify unusual activities that may indicate a popup-based attack. Encouraging users to verify the legitimacy of any popup before interacting with it, especially those requesting personal information or prompting downloads, can further reduce the risk of falling prey to these deceptive tactics. By fostering a culture of security awareness and vigilance, organizations can significantly mitigate the threats posed by popup-based attacks.

**8.Write a script to log the domain and origin of the current page using window.location. Explain how these properties are relevant in CORS security.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Domain and Origin Logger</title>

</head>

<body>

    <h1>Domain and Origin Information of Example.com</h1>

    <p><strong>Domain:</strong> <span id="domain"></span></p>

    <p><strong>Origin:</strong> <span id="origin"></span></p>

    <script>

        // Set the URL to https://example.com

        const url = new URL("https://example.com");

        // Extract domain and origin

        const domain = url.hostname;   // Get domain (e.g., example.com)

        const origin = url.origin;     // Get origin (e.g., https://example.com)

        // Log to console

        console.log("Domain:", domain);

        console.log("Origin:", origin);

        // Display domain and origin on the webpage

        document.getElementById("domain").textContent = domain;

        document.getElementById("origin").textContent = origin;

    </script>

</body>

</html>



Cross-Origin Resource Sharing (CORS) is a security feature that allows web applications to request resources from different origins while maintaining a level of protection against malicious activities. The properties of the `window.location` object—such as hostname, protocol, and pathname—are crucial in this context because they help determine the origin of a request. When a web application makes a cross-origin request, the browser checks the origin of the request against the server's CORS policy. If the origin matches the allowed list specified by the server, the request is permitted; otherwise, it is blocked. This mechanism helps prevent unauthorized access to sensitive data and protects users from cross-site attacks. By carefully managing these properties and implementing strict CORS policies, developers can enhance the security of their applications and safeguard user data from potential threats.

**9.Write a script using window.setTimeout that simulates a login timeout after 10 seconds of inactivity. Enhance the script to reset the timer whenever the user interacts with the page.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Login Timeout Simulation</title>

</head>

<body>

    <h1>Simulate Login Timeout</h1>

    <p>You've been logged in. If you stay inactive for 10 seconds, you will be logged out.</p>

    <p id="timeoutMessage"></p>

    <script>

        let timeoutMessage = document.getElementById("timeoutMessage");

        let timeoutTimer; // Variable to store the timeout timer

        let timeoutDuration = 10000; // 10 seconds timeout duration

        // Function to simulate a timeout (i.e., user is logged out)

        function simulateTimeout() {

            timeoutMessage.textContent = "You have been logged out due to inactivity.";

            console.log("Session Timed Out!");

            alert("Session Timed Out! You have been logged out.");

        }

        // Function to reset the timeout timer

        function resetTimer() {

            console.log("Resetting timeout...");

            clearTimeout(timeoutTimer); // Clear the previous timer

            timeoutMessage.textContent = "Session Active. You are logged in.";

            // Set a new timeout

            timeoutTimer = setTimeout(simulateTimeout, timeoutDuration);

        }

        // Set initial timeout when the page loads

        resetTimer();

        // Reset the timer on various user interactions

        window.addEventListener('mousemove', resetTimer);

        window.addEventListener('keypress', resetTimer);

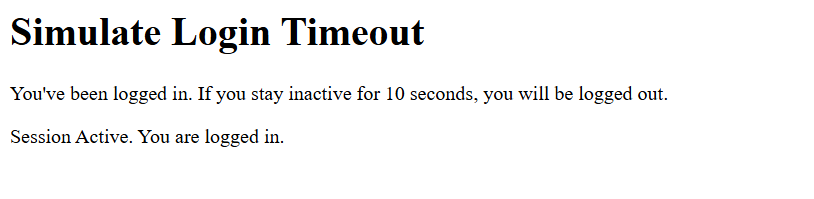
        window.addEventListener('click', resetTimer);

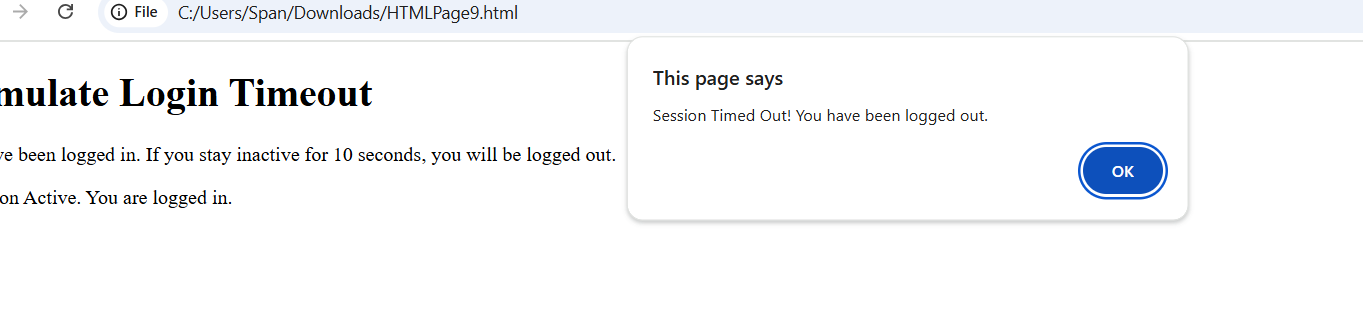
        window.addEventListener('scroll', resetTimer);

    </script>

</body>

</html>





**10.Write a script that checks if the current URL uses "https" using window.location. Alert the user if the page is not secure and suggest measures to handle such scenarios.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>HTTPS Check</title>

</head>

<body>

    <h1>Check for HTTPS</h1>

    <p>This page checks if the current URL uses HTTPS.</p>

    <script>

        // Check if the current URL uses "https"

        if (window.location.protocol !== "https:") {

            alert("This page is not secure. It's using HTTP instead of HTTPS. For security reasons, please ensure that your connection is encrypted.");

            // Suggest measures to handle the non-secure page

            console.log("You should consider switching to HTTPS by obtaining an SSL certificate and configuring your server.");

        } else {

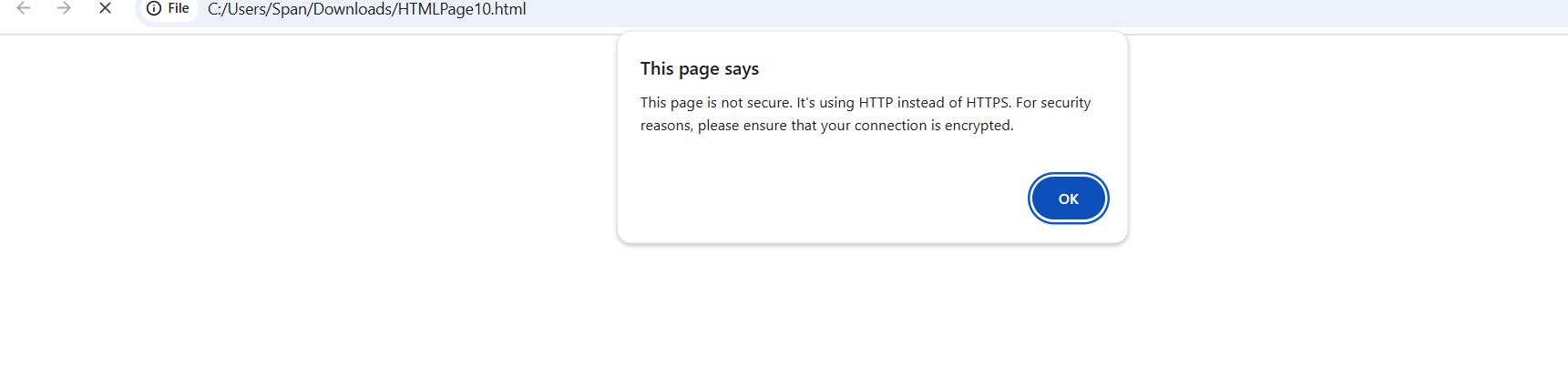
            console.log("The page is secure. It uses HTTPS.");

        }

    </script>

</body>

</html>



**11.Write a script to open a new window and then close it using window.close.**

**Discuss scenarios where malicious scripts might misuse this functionality to disrupt user experience.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Open and Close Window</title>

</head>

<body>

    <h1>Open and Close a New Window</h1>

    <button id="openWindowBtn">Open and Close Window</button>

    <script>

        let newWindow;

        // Function to open a new window and then close it after 3 seconds

        function openAndCloseWindow() {

            // Open a new window

            newWindow = window.open('', '\_blank', 'width=400,height=300');

            newWindow.document.write('<h1>This is a new window!</h1><p>The window will close automatically in 3 seconds.</p>');

            // Close the window after 3 seconds

            setTimeout(() => {

                newWindow.close();

            }, 3000);

        }

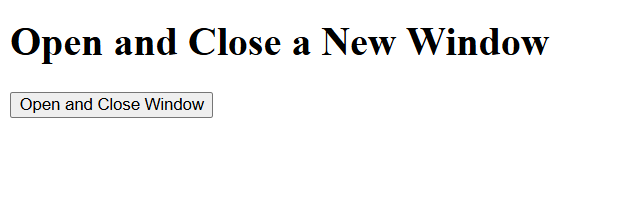
        // Attach the event to the button

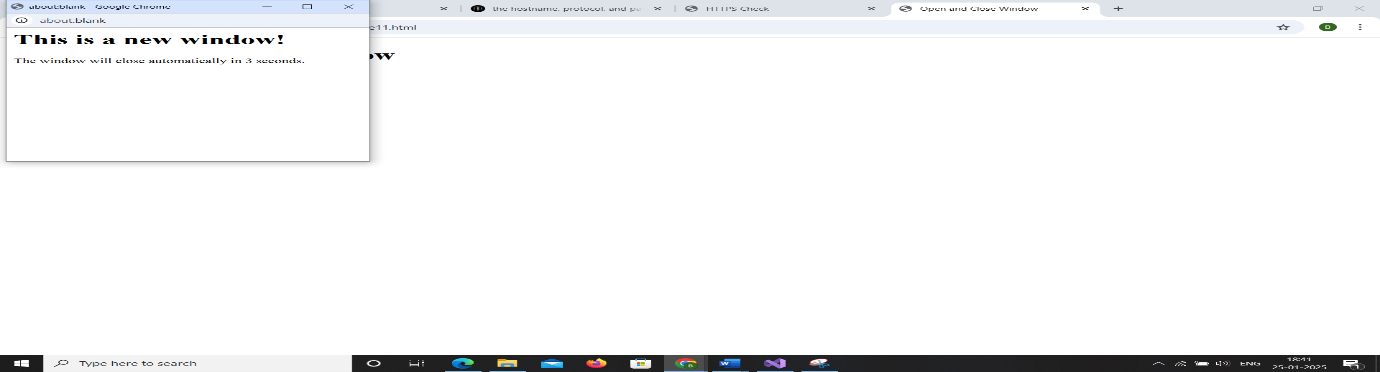
        document.getElementById('openWindowBtn').addEventListener('click', openAndCloseWindow);

    </script>

</body>

</html>





Malicious scripts can significantly disrupt user experience by exploiting functionalities like popups, redirects, and even browser APIs. For instance, an attacker might deploy a script that generates endless popups, overwhelming the user and making it nearly impossible to navigate away from the page. This tactic can lead to frustration and confusion, potentially causing users to inadvertently disclose personal information in a panic. Additionally, scripts can hijack browser controls, redirecting users to unwanted sites or displaying misleading content that mimics legitimate services. Such disruptions not only compromise the user’s online safety but also erode trust in the website, making it essential for developers to implement robust security measures to protect users from these malicious tactics.

**12.Write a script to scroll the webpage to the top using window.scrollTo.**

**Modify the script to scroll to a specific element on the page.**

**Discuss how attackers might use scrolling to hide malicious content or overlays.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Scroll to Top</title>

</head>

<body>

    <h1>Scroll to the Top</h1>

    <p>Scroll down the page and click the button to scroll back to the top.</p>

    <button id="scrollTopBtn">Scroll to Top</button>

    <div style="height: 1500px;">Content here to make the page scrollable...</div>

    <script>

        // Function to scroll to the top of the page

        function scrollToTop() {

            window.scrollTo(0, 0); // This scrolls to the top (without smooth scrolling)

            // Alternative with smooth behavior (for modern browsers):

            // window.scrollTo({ top: 0, behavior: 'smooth' });

        }

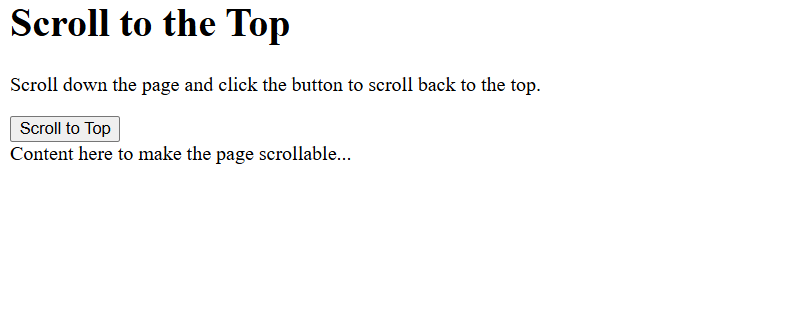
        // Attach the event to the button

        document.getElementById('scrollTopBtn').addEventListener('click', scrollToTop);

    </script>

</body>

</html>



<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Scroll to Specific Element</title>

</head>

<body>

    <h1>Scroll to Specific Element</h1>

    <p>Click the button to scroll to the "Target Element" section.</p>

    <button id="scrollToElementBtn">Scroll to Target Element</button>

    <div style="height: 1500px;">Content here to make the page scrollable...</div>

    <div id="targetElement" style="height: 200px; background-color: lightblue;">

        <h2>hii this is new content of target machine</h2>

        <p>when u scroll to specific target u will land up here.</p>

    </div>

    <div style="height: 1500px;">More content here...</div>

    <script>

        // Function to scroll to a specific element by ID

        function scrollToElement() {

            const element = document.getElementById('targetElement');

            element.scrollIntoView({ behavior: 'smooth' }); // Scrolls smoothly to the element

        }

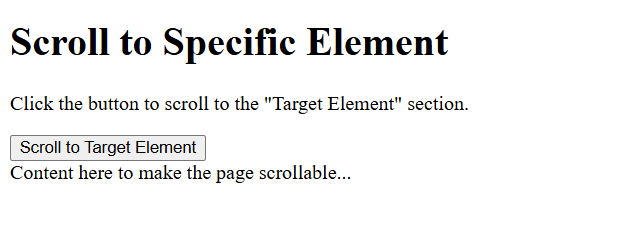
        // Attach the event to the button

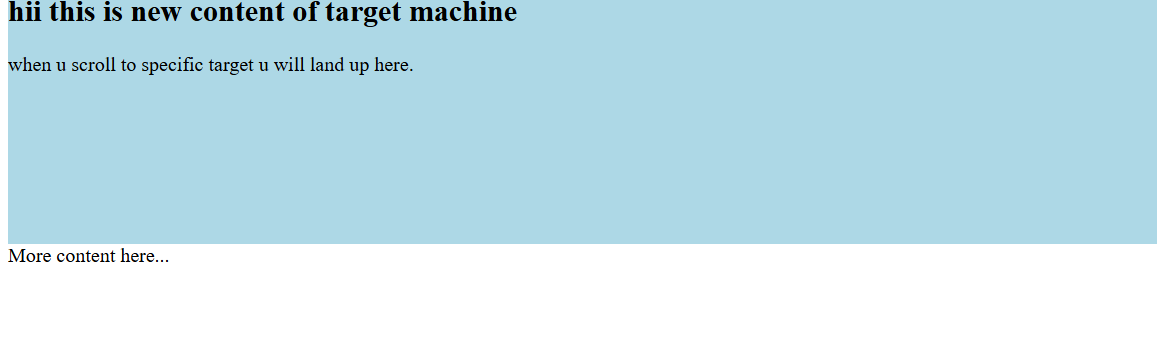
        document.getElementById('scrollToElementBtn').addEventListener('click', scrollToElement);

    </script>

</body>

</html>





Attackers can cleverly exploit scrolling techniques to hide malicious content or overlays, making it difficult for users to notice the deception. For example, they might create a webpage where harmful elements, such as phishing forms or misleading advertisements, are positioned off-screen or behind legitimate content. As users scroll down the page, they may inadvertently reveal these hidden overlays, which can prompt them to enter sensitive information or click on dangerous links without realizing the risk. This tactic takes advantage of user behavior and can lead to significant security breaches, as victims may feel they are interacting with a trustworthy site while being manipulated into compromising their personal data. To combat this, web developers must prioritize transparency and user awareness, ensuring that all content is clearly visible and that users are educated about potential threats lurking in seemingly innocuous scrolling experiences.

1. **Write a script to retrieve the current webpage's title using document.title and log it to the console. Modify the title to include a security warning if it doesn't already contain "Secure." Explain how attackers might manipulate the document title for phishing or social engineering attacks.**

Ans :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Sample Page</title>

</head>

<body>

    <h1>Check and Modify Title</h1>

    <p>This page checks the title and modifies it if necessary.</p>

    <script>

        // Retrieve and log the current title

        let currentTitle = document.title;

        console.log("Current Page Title: " + currentTitle);

        // Check if the title contains the word "Secure"

        if (!currentTitle.includes("Secure")) {

            // Modify the title to include a security warning

            document.title = currentTitle + " - Secure Warning: Ensure Your Connection is Secure!";

        } else {

            console.log("Title already contains 'Secure'. No changes made.");

        }

    </script>

</body>

</html>

Attackers often manipulate document titles to create a false sense of legitimacy, making phishing attempts more convincing. By using familiar or official-sounding titles, they can trick users into believing the document is safe or important, increasing the likelihood that victims will open it and provide sensitive information or click on malicious links. This tactic can involve using titles that mimic legitimate documents, such as "Invoice," "Payment Confirmation," or "Urgent Security Update," which can evoke a sense of urgency or importance. When users see these titles, they may not scrutinize the source or content closely, leading them to act impulsively. This manipulation plays on human psychology, leveraging trust and familiarity to bypass critical thinking, ultimately resulting in successful social engineering attacks that compromise sensitive data.

**----------------------------------------------------------------------------------------**

**'If you spend too much time thinking about a thing, you'll never get it done.' - Bruce Lee**

**'If you really want to do something, you'll find a way. If you don't, you'll find an excuse.' - Jim Rohn**