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.

**const url = window.location.href;**

**console.log("Current URL:", url);**

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

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

**console.log(`Query Param: ${key} = ${value}`);**

**});**

**Security Risk:  
Phishing attackers can use query parameters to pass fake login pages or session hijacking URLs.**

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.

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

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

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

window.location.hostname

Retrieves the domain name of the website (e.g., example.com).

window.location.protocol

Retrieves the protocol being used (http: or https:).

window.location.pathname

Retrieves the specific path after the domain (/page).

3.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.

**function safeRedirect(url) {**

**const allowedDomains = ["example.com", "trusted.com"]; // Define trusted domains**

**try {**

**const targetURL = new URL(url); // Parse the URL**

**if (allowedDomains.includes(targetURL.hostname)) {**

**window.location.href = url; // Redirect if domain is trusted**

**} else {**

**console.warn("Blocked redirect to untrusted domain:", url);**

**}**

**} catch (error) {**

**console.error("Invalid URL:", url);**

**}**

**}**

**// Example usage**

**safeRedirect("https://trusted.com/dashboard");**

**safeRedirect("https://phishing-site.com/fake-login"); // This will be blocked**

* An open redirect occurs when a website allows any URL to be used in redirection without validation. Attackers exploit this by crafting URLs that appear to be from a trusted site but redirect users to phishing pages.

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.

**let pageVisitCount = 0;**

**// Function to track page visits**

**function trackPageVisit() {**

**pageVisitCount++;**

**console.log(`Pages visited in this session: ${pageVisitCount}`);**

**}**

**// Monitor history changes (such as page navigations, popstate events)**

**window.onpopstate = function() {**

**trackPageVisit();**

**};**

**// Initial page visit**

**trackPageVisit();**

**While this approach might seem harmless, there are privacy risks associated with tracking browser history:**

1. **Malicious Scripts:**
   * **Malicious scripts could exploit window.history to track user activity and potentially gather sensitive information, like the sequence of sites visited.**
   * **By monitoring history.length or using events like popstate, a script can try to infer patterns in browsing behavior.**
2. **Fingerprinting:**
   * **A script could try to correlate the user's navigation with other identifiers (like IP address, browser user-agent, or device properties) to create a "fingerprint" of their online behavior.**
   * **With enough data, this could lead to tracking users across sessions or even across websites without their knowledge.**
3. **Cross-Site Tracking:**
   * **If a user visits multiple websites with embedded malicious scripts, these scripts could use the history data from one site to track visits to others, potentially violating user privacy.**

5.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.

**// Get the user agent string**

**const userAgent = window.navigator.userAgent;**

**// Log the user agent string**

**console.log("User Agent: " + userAgent);**

**// Function to identify the browser**

**function identifyBrowser(ua) {**

**if (/Chrome/.test(ua) && /Google Inc/.test(navigator.vendor)) {**

**console.log("Browser: Google Chrome");**

**} else if (/Firefox/.test(ua)) {**

**console.log("Browser: Mozilla Firefox");**

**} else if (/HeadlessChrome/.test(ua)) {**

**console.log("Automation Tool: Headless Chrome (likely Puppeteer or similar)");**

**} else {**

**console.log("Browser: Unknown");**

**}**

**}**

**// Call the function to identify the browser**

**identifyBrowser(userAgent);**

**Potential Issues:**

* **False Positives: Automation tools might modify their user agent strings to avoid detection. For example, Puppeteer might use a Chrome user agent and could be missed if it doesn't specifically indicate it's headless.**
* **Customization: Some users may change their user agent string (via browser extensions or settings), so detection might not always be accurate.**

6. 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.

**// Get screen dimensions**

**const screenWidth = window.screen.width;**

**const screenHeight = window.screen.height;**

**// Log the screen dimensions**

**console.log(`Screen Width: ${screenWidth}px`);**

**console.log(`Screen Height: ${screenHeight}px`);**

**How Attackers Might Use Screen Dimensions in Phishing Attacks:**

1. **Tailored Phishing Pages:**
   * **Attackers could use the screen dimensions to dynamically adjust the layout of a phishing page to fit the victim's screen perfectly. This can create a more convincing fake page that appears legitimate, as it will match the user's display size.**
   * **For example, if they know the user is using a mobile device, they can present a mobile-optimized phishing page. For a desktop user, they could craft a page designed for larger screens.**
2. **Deceptive Pop-ups:**
   * **Knowing the screen size allows attackers to position pop-ups or fake alerts in such a way that they avoid being blocked or noticed by the user. The pop-up could be placed at the perfect spot on the screen, making it appear as if it's part of the operating system or legitimate website.**
3. **Social Engineering with Personalized Experience:**
   * **Attackers could use the screen size to make the phishing page feel personalized, possibly by adding custom visuals, such as resizing logos or buttons, to make the page look more authentic based on the victim’s device.**
4. **Targeting Specific Devices:**
   * **Phishing attackers could also use the screen dimensions to target users of specific devices (e.g., large screens like desktops vs. small screens like smartphones) to deliver specialized attacks, including tailored login forms or fake system notifications.**

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

**// Open a new popup window**

**const popup = window.open('https://www.example.com', 'PopupWindow', 'width=600,height=400');**

**// Check if the popup was blocked**

**if (!popup) {**

**console.log("Popup blocked by the browser.");**

**} else {**

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

**}**

**Risks Associated with Popup-based Attacks:**

1. **Malicious Popups (Phishing and Fraud):**
   * **Attackers can create popups that look like legitimate login forms or alerts, tricking users into entering sensitive information like passwords or credit card details. For example, a fake "security warning" popup may ask users to input their credentials to avoid account suspension.**
2. **Ad Fraud (Popunder Ads):**
   * **Popups can be used to display unwanted advertisements, sometimes even hidden behind the main window (popunder). These ads can lead to misleading or malicious websites.**
3. **Disruptive or Annoying Behavior:**
   * **Constant popup windows can frustrate users and force them to interact with unwanted content. This could harm user experience or even cause a denial-of-service if too many popups are opened.**
4. **Cross-Site Scripting (XSS) and Social Engineering:**
   * **Attackers can exploit popups to inject malicious code (like XSS) into web pages, allowing them to steal user data, cookies, or sessions.**
   * **A popup can also serve as an entry point for further social engineering tactics, such as convincing the user to download malware.**
5. **Bypass User Consent:**
   * **Some popups can prompt users to click on buttons or grant permissions in a deceptive way, such as asking for access to the clipboard, location, or microphone under false pretenses.**

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.

**// Get the domain and origin**

**const domain = window.location.hostname;**

**const origin = window.location.origin;**

**// Log the domain and origin**

**console.log(`Domain: ${domain}`);**

**console.log(`Origin: ${origin}`);**

**Relevance in CORS Security:**

**Cross-Origin Resource Sharing (CORS) is a security mechanism that allows or restricts web applications running at one origin (domain, protocol, and port) from making requests to a different origin. CORS helps prevent malicious websites from reading sensitive data from another domain.**

1. **Origin:**
   * **The origin includes the protocol (e.g., http or https), the domain (e.g., www.example.com), and the port (if specified). It is the most critical part of CORS security because it's used to determine whether a resource request is allowed.**
   * **For example, if a website at https://www.example.com tries to make a request to an API hosted at https://api.example.com, both origins match (they share the same protocol and domain), so CORS may allow the request.**
2. **Domain:**
   * **The domain is a subset of the origin, and CORS policies depend on whether the requesting domain matches the origin of the resource. If the domains differ, the server will check the request's Origin header and decide whether to allow or block the request.**
   * **For example, https://example.com and https://sub.example.com are different domains, and CORS policies might block or allow requests depending on the server's configuration.**

**CORS and Origin-Based Security:**

* **Preflight Requests: Before making a cross-origin request (such as an AJAX request), the browser sends a preflight request (an OPTIONS request) to the server to check if the actual request is safe to send. The server must respond with appropriate CORS headers to confirm that the request is allowed.**
* **Access-Control-Allow-Origin Header: This header is returned by the server in response to a preflight request or a cross-origin request, indicating which domains are allowed to access the resource.**
  + **If the requesting origin is allowed, the server includes that origin in the Access-Control-Allow-Origin header.**
  + **If the origin does not match the allowed domains, the server will deny the request.**

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.

**// Time in milliseconds before the session times out (10 seconds)**

**const timeoutDuration = 10000;**

**// Variable to store the timeout ID**

**let timeoutId;**

**// Function to simulate a logout after inactivity**

**function logoutUser() {**

**alert('Your session has expired due to inactivity. Logging out...');**

**// Simulate user logout (e.g., redirect to login page or log out the user)**

**window.location.href = '/login'; // Change to the actual login URL**

**}**

**// Function to reset the inactivity timer**

**function resetTimer() {**

**// Clear the previous timeout**

**clearTimeout(timeoutId);**

**// Set a new timeout**

**timeoutId = setTimeout(logoutUser, timeoutDuration);**

**}**

**// Set the initial timeout**

**timeoutId = setTimeout(logoutUser, timeoutDuration);**

**// Event listeners to detect user activity and reset the timer**

**window.addEventListener('mousemove', resetTimer); // Mouse movement**

**window.addEventListener('keydown', resetTimer); // Key press**

**window.addEventListener('click', resetTimer); // Mouse click**

**// Optionally, you can add touch events for mobile support**

**window.addEventListener('touchstart', resetTimer); // Touch events (for mobile)**

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.

**// Check if the current URL uses 'https'**

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

**// Alert the user if the page is not secure**

**alert('This page is not secure! It is using HTTP instead of HTTPS.\n\nFor your safety, we recommend the following:\n- Do not enter sensitive information on this page.\n- Contact the website owner to request they enable HTTPS for security.\n- Consider using a browser extension that forces HTTPS (like HTTPS Everywhere).');**

**} else {**

**console.log('The page is secure using HTTPS.');**

**}**

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.

**// Open a new window**

**const newWindow = window.open('https://www.example.com', 'NewWindow', 'width=600,height=400');**

**// Check if the window was successfully opened**

**if (newWindow) {**

**// Close the new window after 3 seconds**

**setTimeout(function() {**

**newWindow.close();**

**console.log("The new window has been closed.");**

**}, 3000);**

**} else {**

**console.log("Failed to open a new window. The browser might have blocked it.");**

**}**

**Potential Malicious Use of window.open and window.close**

**While window.open and window.close can be useful in legitimate scenarios (such as popups for help, forms, or ads), they can also be misused by malicious scripts to disrupt the user experience in various ways:**

1. **Annoying Popups:**
   * **Malicious scripts can continuously open new windows (popups) and close them immediately or at regular intervals. This can result in a flood of popups, overwhelming the user and disrupting their browsing experience.**
   * **Example: A script that opens and closes hundreds of windows in a loop could cause the browser to freeze, making it difficult for the user to regain control.**
2. **Redirecting or Hijacking User's Browser:**
   * **Malicious scripts could open multiple popups with different content, such as fake login forms or phishing sites, in an attempt to steal sensitive information.**
   * **Automatic Redirection: A script could open a popup with a malicious website and close it, leaving behind another hidden window that continues redirecting the user to harmful sites.**
   * **Fake Dialog Boxes: Attackers can also simulate alerts or confirmation dialogs in popups and close them as soon as the user attempts to interact with them, making it difficult for the user to navigate.**
3. **Denial of Service (DoS) via Infinite Popups:**
   * **Scripts that repeatedly open new windows and close them in a loop can overwhelm the browser or cause the system to become unresponsive. This is a form of a Denial of Service (DoS) attack where the user is unable to continue browsing because their resources (e.g., memory, processing power) are consumed by the excessive popups.**
4. **Involuntary Closure of Legitimate Windows:**
   * **A malicious site could open a window that mimics an important task or a form. It could then close the window at an inappropriate time, interrupting the user's experience or causing them to lose unsaved data.**
   * **Example: A user might be filling out a form or reading a document, and a malicious script could automatically close the window, causing frustration and data loss.**
5. **Clickjacking with Popups:**
   * **Popups might be used in conjunction with clickjacking attacks, where an invisible popup or iframe might be used to trick the user into clicking on hidden buttons or submitting forms without their consent. This can lead to unintended actions such as liking a post, transferring money, or installing malware.**

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

**// Scroll to the top of the page**

**window.scrollTo(0, 0);**

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

**// Get the specific element you want to scroll to**

**const targetElement = document.querySelector('#targetElement');**

**// Scroll to the specific element**

**if (targetElement) {**

**targetElement.scrollIntoView({ behavior: 'smooth', block: 'start' });**

**} else {**

**console.log('Element not found.');**

**}**

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

**How Attackers Might Use Scrolling to Hide Malicious Content or Overlays**

**Attackers can misuse scrolling functions like window.scrollTo or element.scrollIntoView in several ways to deceive or mislead users. Some common methods include:**

1. **Hiding Malicious Popups or Overlays:**
   * **Invisible Popups: An attacker could create a malicious popup or overlay (e.g., a fake login form or phishing page) that appears hidden off-screen and then scrolls to that position when the user interacts with the page.**
   * **Example: A script could use window.scrollTo to scroll to a location on the page where a hidden iframe or malicious content is placed. The user might not see the content at first, but after scrolling, the malicious content becomes visible.**
2. **Creating Fake UI Elements:**
   * **Attackers could manipulate the scroll position to display fake UI elements or buttons that look legitimate but are designed to steal user information.**
   * **Example: A page might initially appear safe, but once the user scrolls to a particular section (invisible until triggered by scrolling), a fake "Update Your Account" button might appear, leading to a phishing form.**
3. **Clickjacking:**
   * **By controlling the scroll behavior, attackers can hide an iframe with an invisible or seemingly benign appearance and make it scroll into view only when the user clicks on certain parts of the page. This technique can be used to trick users into clicking hidden buttons that perform unwanted actions.**
   * **Example: An attacker might place a transparent iframe over a legitimate button, then scroll the page so that the user clicks on the hidden iframe instead of the actual button, causing unintended actions like transferring money or liking a post.**
4. **Obfuscating Malicious Content:**
   * **Attackers might use scrolling to control when certain content becomes visible to the user, making it difficult for security tools or the user to spot the malicious code.**
   * **For example, an attacker might initially load a page with an innocuous design, but when the user scrolls down, a hidden form, ad, or download prompt (with malware) becomes visible. This can trick the user into interacting with the malicious content unintentionally.**
5. **Disrupting User Interaction:**
   * **Attackers might manipulate the scroll position to "push" a user down a page, hiding content like legitimate forms or buttons that require interaction. This could be used to prevent users from noticing warnings, buttons, or sections they need to interact** with.

13. 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.

**// Retrieve the current webpage's title**

**const currentTitle = document.title;**

**console.log('Current 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';**

**console.log('Updated Title:', document.title);**

**} else {**

**console.log('The title already contains "Secure".');**

**}**

**How Attackers Might Manipulate the Document Title for Phishing or Social Engineering Attacks**

**Manipulating the document title can be used by attackers to deceive users in various ways, especially in phishing or social engineering attacks. Some scenarios where attackers could exploit this functionality include:**

1. **Mimicking Trusted Websites:**
   * **Attackers may create fake login pages or phishing sites that mimic legitimate websites (e.g., banking, social media). They might manipulate the document title to include the trusted site's name or add terms like "Secure" or "Verified" to convince the user that the page is authentic.**
   * **Example: A phishing site could change the title to "Banking Portal - Secure" to make it appear more trustworthy and encourage users to input sensitive information like login credentials.**
2. **Disguising Malicious Content:**
   * **If a user is on a compromised website, an attacker might change the title of the page to something seemingly harmless or secure-looking to disguise the true nature of the content.**
   * **Example: A user might visit a malicious site, and the attacker could change the title to "Welcome to Your Account - Secure", even though the page is actually a scam designed to steal personal information.**
3. **Manipulating User Trust:**
   * **A phishing attacker could dynamically alter the title to include terms like "Secure", "Verified", or "Encrypted", giving the false impression that the page is safe and secure, thus lowering the user's suspicion and making them more likely to perform risky actions.**
   * **Example: A fake form could say "Payment Details - Secure" in the title, prompting users to enter credit card details on a fake website.**
4. **Phishing with Brand Reputation:**
   * **Attackers might modify the title to impersonate a known brand or service. This can be part of a wider campaign to spoof well-known companies and trick users into thinking they are interacting with legitimate services.**
   * **Example: The attacker could change the title of a phishing page to "PayPal - Secure" or "Amazon - Secure" to impersonate these companies and steal login credentials or credit card numbers.**

----------------------------------------------------------------------------------------

'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