1. **Question:** A web application sanitizes user input by stripping out <script> tags. However, it uses a WYSIWYG editor that allows users to embed <img> tags. How can you exploit this for XSS?

   Answer: Use the onerror event handler within the <img> tag: <img src="invalid" onerror="alert('XSS')">.

   Explanation: Even though <script> is blocked, event handlers like onerror can execute JavaScript when triggered by an event (in this case, an error loading the image).
2. **Question:** A web application uses a blacklist to prevent certain keywords like javascript:. How can you bypass this to execute JavaScript within an <a> tag's href attribute?

   Answer: Utilize URL encoding or case manipulation: Click Me or Click Me.

   Explanation: Blacklists are often incomplete. Browsers might normalize or decode the input before processing, bypassing the simple string matching.
3. **Question:** A web application has a strict Content Security Policy (CSP) that disallows inline scripts (script-src 'self'). However, it allows loading scripts from a specific trusted domain. How can you achieve XSS if you can control a file hosted on that trusted domain?

   Answer: Upload a JavaScript file containing your XSS payload to the trusted domain and then include it in the vulnerable page: <script src="https://trusted-domain.com/your_malicious_script.js"></script>.

   Explanation: CSP's effectiveness relies on strict configuration. If an attacker can inject a <script> tag pointing to a whitelisted domain they control, they can bypass the inline script restriction.

4. **Question:** A web application uses a template engine that escapes HTML entities by default. However, it provides a "raw" output option for certain user-controlled data. How can you exploit this?

   Answer: Inject HTML-encoded JavaScript within an event handler attribute within a tag that is rendered using the "raw" output: Hover Me .

   Explanation: While the template engine escapes HTML for regular content, the "raw" output bypasses this, allowing the browser to interpret the HTML-encoded JavaScript within the event handler.

5. **Question:** A web application uses a client-side JavaScript framework that dynamically renders content based on user input. You find a sink where user-controlled data is passed to a function that manipulates the DOM without proper sanitization. How can you exploit this?

   Answer: Craft input that, when processed by the JavaScript framework, injects malicious HTML or JavaScript into the DOM. This often involves understanding the framework's specific APIs and how it handles data binding or templating. For example, in some frameworks, you might be able to inject HTML that includes a <script> tag or an event handler.

   Explanation: Client-side rendering doesn't inherently prevent XSS. If the JavaScript code manipulating the DOM doesn't properly sanitize user input before injecting it, vulnerabilities can arise.

6. **Question:** A web application uses a web worker to process user-provided data. The worker then sends a message back to the main thread, which updates the DOM with this data. How can you achieve XSS in this scenario?

   Answer: Craft malicious data that, when processed by the web worker and sent back to the main thread, results in the injection of HTML or JavaScript into the DOM when the main thread updates it. This might involve exploiting vulnerabilities in how the worker processes the data or how the main thread handles the message.

   Explanation: XSS can occur even when data is processed in separate threads if the final rendering of the data in the main thread isn't properly secured.

7. **Question:** A web application allows users to upload SVG files. How can you leverage this functionality for XSS?

   Answer: Embed JavaScript directly within the SVG file using <script> tags or within event attributes of SVG elements (e.g., <svg><script>alert('XSS')</script></svg> or <svg><rect onmouseover="alert('XSS')" /></svg>).

   Explanation: Browsers often render SVG files, including any embedded JavaScript. If the application doesn't properly sanitize uploaded SVG content, it can lead to XSS.

8. **Question:** A web application uses a third-party library that has a known XSS vulnerability. You can control the input that is passed to a vulnerable function in this library. How can you exploit this?

   Answer: Provide input specifically crafted to trigger the XSS vulnerability within the third-party library. This requires knowledge of the library's weaknesses.

   Explanation: Relying on third-party libraries introduces potential security risks. If these libraries have vulnerabilities and user-controlled data interacts with them, XSS can be exploited.

9. **Question:** A web application sets an HTTP-only cookie containing sensitive information. However, it also reflects some user input into the HTML without proper sanitization. Can you still steal the cookie?

   Answer: No, HTTP-only cookies are protected from client-side JavaScript access. While you can inject JavaScript, you won't be able to read the HTTP-only cookie using document.cookie.

   Explanation: The HTTP-only flag is a crucial defense mechanism against cookie theft via XSS.

10. **Question:** A web application uses postMessage to communicate between different frames or windows. You can control the postMessage content sent to a vulnerable frame. How can you exploit this for XSS?

    Answer: Craft a malicious message that, when received and processed by the vulnerable frame, leads to the execution of JavaScript, often by manipulating the receiving frame's DOM based on the message content.

    Explanation: Improper handling of messages received via postMessage can create XSS vulnerabilities if the receiving end doesn't validate and sanitize the data.

11. **Question:** A web application uses server-side rendering and includes user-provided data directly into JavaScript code within <script> tags. How can you achieve XSS?

Answer: Inject JavaScript code that terminates the existing script block and introduces your own malicious JavaScript. For example, if the code looks like var data = "${userInput}";, you could inject "; alert('XSS'); //.

Explanation: Directly embedding user input into JavaScript code without proper encoding can lead to the execution of arbitrary JavaScript.

12. **Question:** A web application allows users to customize CSS. How can you potentially exploit this for XSS?

    Answer: While direct JavaScript execution via CSS is limited, you might be able to use CSS expressions (in older browsers) or leverage vulnerabilities in how the application handles CSS to leak information or potentially trigger other browser behaviors. In modern browsers, this is significantly harder but not entirely impossible in all edge cases (e.g., through url() with crafted content or interaction with other vulnerabilities).

    Explanation: While not a direct route to <script> execution in modern browsers, malicious CSS can sometimes be used for information leakage or, in combination with other vulnerabilities, for more severe attacks.

13. **Question:** A web application uses a URL shortener. If you can control the target URL, how can you potentially use this for XSS?

    Answer: Redirect the shortened URL to a page you control that contains XSS payload. When a user clicks the shortened link, they will be redirected to your malicious page, and the XSS will execute in the context of your page (not the original application). This is more of a social engineering attack leveraging the application's trust.

    Explanation: While not directly an XSS vulnerability in the URL shortener itself, it can be used as a delivery mechanism for XSS hosted elsewhere.

14. **Question:** A web application uses server-sent events (SSE) to push updates to the client. If you can control the data sent via SSE, how can you achieve XSS?

    Answer: If the client-side JavaScript handling the SSE data directly renders it into the DOM without proper sanitization, you can inject malicious HTML or JavaScript within the SSE data.

    Explanation: Just like with regular server responses, data received via SSE needs to be treated carefully on the client-side to prevent XSS.

15. **Question:** A web application uses a WebSocket connection to exchange data with the client. If you can control the messages sent by the server, how can you achieve XSS?

    Answer: Similar to SSE, if the client-side JavaScript that receives and processes WebSocket messages directly renders the data into the DOM without sanitization, you can inject malicious HTML or JavaScript.

    Explanation: Real-time communication channels like WebSockets are also potential vectors for XSS if the client-side handling of the data is insecure.

16. **Question:** A web application has a vulnerability that allows you to control a part of the URL hash (#). The client-side JavaScript reads this hash and uses it to dynamically update the page content. How can you exploit this for XSS?

Answer: Craft a malicious payload within the URL hash that, when read and processed by the client-side JavaScript, results in the injection of HTML or JavaScript into the DOM. This often involves understanding how the JavaScript parses and uses the hash value. For example, if it uses innerHTML with the hash, you could inject <img src=x onerror=alert('XSS')>.

Explanation: Even data within the URL fragment (hash) can be a source of XSS if client-side scripts process it unsafely.

17. **Question:** A web application uses browser local storage to store user-provided data. Another part of the application reads this data from local storage and uses it to update the DOM without sanitization. How can you exploit this?

    Answer: Inject a malicious payload into local storage through one part of the application. When the vulnerable part of the application reads this data and uses it to update the DOM, the XSS payload will be executed.

    Explanation: Client-side storage mechanisms like local storage can be a vector for stored XSS if data written to them is not properly sanitized when read and rendered.

18. **Question:** A web application allows users to upload files and then displays previews of these files using the browser's built-in rendering capabilities. How can you exploit this for XSS?

    Answer: Upload a specially crafted file (e.g., an HTML file with a <script> tag, or a seemingly harmless file format that can be interpreted as HTML in certain contexts) that contains your XSS payload. When the browser renders the preview, the malicious script will execute.

    Explanation: Relying on the browser's rendering of user-uploaded content without proper sanitization can lead to XSS if the browser interprets the content in an unexpected way.

19. **Question:** A web application uses a service worker to intercept network requests and modify responses. If you can somehow control the content injected by the service worker, how can you achieve XSS?

    Answer: If you can manipulate the service worker to inject malicious HTML or JavaScript into the response of a request that the application then renders, you can achieve XSS. This could involve exploiting vulnerabilities in the service worker's logic or its interaction with cached data.

    Explanation: Service workers, while powerful, can also introduce security risks if not implemented carefully, as they can control and modify the application's network traffic.

20. **Question:** A web application uses a feature that allows embedding content from other websites via iframes. However, it dynamically constructs the src attribute of the iframe based on user input without proper validation. How can you exploit this for XSS?

    Answer: Inject a URL into the src attribute that points to a page you control containing an XSS payload. While the script will execute within the iframe's context, it can still potentially interact with the parent window depending on the application's security settings (e.g., sandbox attribute).

    Explanation: Dynamically generating iframe src attributes based on user input without proper validation can lead to the embedding of malicious pages.

21. **Question:** A web application uses a robust HTML sanitizer on the server-side. However, it allows users to input data that is later used to construct a JSON object. This JSON object is then embedded within a <script> tag and parsed by client-side JavaScript. How can you achieve XSS?

    Answer: Inject characters that, when included in the JSON string and parsed by JavaScript, can break out of the string context and execute arbitrary JavaScript. For example, if the JSON is constructed like {"value": "${userInput}"}, you could inject "}; alert('XSS'); // {"other":". This would result in the JavaScript code: {"value": ""}; alert('XSS'); // {"other":"";}.

    Explanation: Even with HTML sanitization, vulnerabilities can arise if user input is used in other data formats (like JSON) without proper encoding for that format. JavaScript's loose parsing can be exploited to break out of string literals.

22. **Question:** A web application serves static HTML files. However, it uses a custom error page that includes the requested (and non-existent) URL in the error message. This URL is not directly rendered but is used within a client-side JavaScript function to display it. How can you exploit this for XSS?

    Answer: Craft a malicious URL that, when accessed and reflected in the error message, triggers an XSS payload when the client-side JavaScript processes it. This might involve URL encoding or special characters that are not properly handled by the JavaScript function displaying the URL. For instance, if the JavaScript uses decodeURIComponent() on the URL, you might be able to inject %3Cscript%3Ealert('XSS')%3C/script%3E.

    Explanation: Client-side rendering of data derived from the URL, even in error pages, can be a source of XSS if the JavaScript doesn't properly handle potentially malicious characters.

23. **Question:** A web application uses a shadow DOM for certain UI components. User-provided data is used to set the innerHTML of an element within the shadow DOM. Can you achieve XSS that affects the main document context?

    Answer: While the XSS will initially execute within the shadow DOM's scope, depending on the browser's behavior and any custom event handling or JavaScript interactions between the shadow DOM and the main DOM, it might be possible to influence the main document. For example, a script in the shadow DOM could potentially dispatch events that are handled by the main document's JavaScript in an unsafe way, or manipulate properties that have effects outside the shadow DOM. This is a more nuanced and browser-dependent scenario.

    Explanation: Shadow DOM provides encapsulation, but the boundaries aren't always impenetrable, especially when JavaScript interacts between the shadow root and the main document.

24. **Question:** A web application implements a strict CSP with script-src 'none'. However, it allows users to upload and view images. How can you potentially achieve code execution?

    Answer: This is extremely difficult with a strict script-src 'none'. However, historical browser vulnerabilities have sometimes allowed code execution through image rendering or metadata handling. This would rely on a very specific and likely rare browser bug related to image processing. It's not a standard XSS vector under a properly enforced script-src 'none'.

    Explanation: script-src 'none' is a very effective defense against most XSS attacks. Exploiting this would typically require uncovering a novel and severe browser vulnerability.

25. **Question:** A web application uses a web worker to render a preview of user-provided Markdown. The worker sanitizes the Markdown to prevent typical HTML injection. However, the application allows embedding links in the Markdown. How can you exploit this?

    Answer: Craft a Markdown link with a javascript: URL: [Click Me](javascript:alert('XSS')). If the Markdown rendering library in the web worker doesn't explicitly sanitize the protocols of URLs in links, this could lead to JavaScript execution when the link is clicked in the rendered preview.

    Explanation: Even if HTML tags are sanitized, other potential injection points like URL protocols within attributes need to be considered.

26. **Question:** A web application uses a client-side routing library. User-provided data is used to construct part of the route. A specific route handler dynamically includes content based on this route parameter using innerHTML. How can you achieve XSS?

    Answer: Craft a malicious route that, when processed by the routing library and used to construct the path for content inclusion, leads to the injection of HTML containing a <script> tag or an event handler. This depends on how the routing library handles special characters and how the included content is fetched and injected.

    Explanation: Client-side routing, if not carefully implemented, can introduce XSS vulnerabilities if user input influences the content that is subsequently rendered into the DOM using unsafe methods like innerHTML.

27. **Question:** A web application uses a templating engine on the client-side that claims to auto-escape HTML. However, it provides a custom directive or helper function that allows rendering raw HTML. If you can control data passed to this raw rendering directive, how can you exploit it?

    Answer: Inject your XSS payload directly through the raw rendering directive. Since it bypasses the default escaping mechanism, any HTML or JavaScript you provide will be rendered directly into the DOM.

    Explanation: Templating engines with "raw" output features are common XSS attack vectors if user-controlled data can reach them without proper sanitization.

28. **Question:** A web application interacts with a browser extension via custom events. If you can trigger the sending of a custom event with user-controlled data, and the browser extension processes this data and modifies the web page's DOM in an unsafe way, can you achieve XSS?

    Answer: Yes, if the browser extension doesn't properly sanitize the data received via the custom event before using it to manipulate the DOM of the web application, you can achieve XSS. This is a form of extension-assisted XSS.

    Explanation: Security boundaries can extend beyond the web application itself to include browser extensions it interacts with. Vulnerabilities in the extension's handling of data from the web application can be exploited.

29. **Question:** A web application uses a third-party analytics script that dynamically loads other JavaScript resources based on URL parameters. If you can control a URL parameter that influences the loading of these secondary scripts, how can you achieve XSS?

Answer: Craft a URL with a parameter value that points to a JavaScript file you control. If the analytics script doesn't properly validate or sanitize this URL, it will load and execute your malicious script in the context of the web application.

Explanation: Dynamically loading scripts based on user-controlled input is a significant XSS risk. The source of these scripts must be carefully controlled and validated.

30. **Question:** A web application uses a service worker to cache responses. If you can manipulate the cache (e.g., through a cache poisoning vulnerability) to store a malicious HTML page for a specific URL that the application later fetches and renders, can you achieve XSS?

    Answer: Yes, if you can poison the service worker's cache with a malicious HTML page containing an XSS payload, and the application subsequently fetches and renders this cached response, the XSS will be executed. This is a more indirect and challenging form of XSS.

    Explanation: Service worker caching introduces a new layer where vulnerabilities can be exploited. Cache poisoning can lead to the delivery of malicious content even if the server-side application itself is not directly vulnerable to XSS.

31. **Question:** A web application has a seemingly harmless search functionality. The search term is reflected on the results page within a <label> tag. However, the application also stores the last 5 search terms per user in their profile. Another part of the profile page retrieves and displays these stored search terms within an unordered list using innerHTML. How can you achieve persistent XSS?

    Answer: Inject a search term containing an XSS payload (e.g., <img src=x onerror=alert('XSS')>). While the initial reflection within the <label> might not be directly exploitable, it will be stored in the user's profile. When the profile page is viewed, the stored search term will be rendered using innerHTML, executing the XSS payload persistently.

    Explanation: This combines reflected and stored XSS. The initial injection point is reflected, but the persistence occurs through storage and later unsafe rendering.

32. **Question:** A web application allows users to create public notes. The note content is sanitized on the server-side using a blacklist approach. When a note is viewed, the content is displayed. Additionally, users can embed links to other notes using a specific markdown-like syntax: [[note_id]]. When parsed, this is replaced with an <a> tag linking to the other note. If you can create a note and reference it in another, how can you achieve stored XSS that bypasses the blacklist?

    Answer: Create a first note with content designed to bypass the blacklist (e.g., using event handlers in <img> tags). Then, in a second note, create a link to the first note using [[note_id_of_first_note]]. When the second note is viewed, the link will be rendered, and when clicked or interacted with (depending on the payload in the first note), the XSS will execute.

    Explanation: This leverages a combination of a potentially weak blacklist and an internal linking mechanism to deliver a stored XSS payload.

33. **Question:** A web application has a user settings page where users can set their preferred language. This preference is stored in a cookie. On every page load, client-side JavaScript reads this cookie and dynamically loads a language-specific JSON file containing translations. If you can

somehow influence the value of this language cookie (e.g., through another vulnerability or a social engineering attack), how can you achieve XSS?

Answer: Set the language cookie to a value that points to a JSON file hosted on your server. This JSON file would contain malicious JavaScript code disguised as translation data. When the client-side JavaScript loads and processes this "translation" file, your JavaScript will be executed.

Explanation: This is a form of client-side stored XSS where the "stored" element is a cookie value that influences the loading of external data, which is then treated as executable code.

34. **Question:** A web application allows users to leave comments on articles. The comment input is sanitized on the server-side. However, the application uses a client-side library to automatically linkify URLs in the comments. If you can craft a comment containing a specific type of URL, can you achieve XSS when the comment is displayed?

    Answer: Inject a comment containing a javascript: URL. If the client-side linkify library doesn't properly sanitize the URL protocols before creating the <a> tag, clicking on the generated link will execute the JavaScript. This is a stored XSS vulnerability that relies on client-side processing.

    Explanation: Even if server-side sanitization is in place, client-side processing of stored data can introduce new XSS vulnerabilities.

35. **Question:** A web application has an API endpoint that accepts user-provided data and returns it as a JSON response. This response is then used by client-side JavaScript to dynamically update the DOM using innerHTML. If you can control the input to this API endpoint (perhaps through a reflected vulnerability elsewhere), how can you achieve stored XSS?

    Answer: Inject a payload into the reflected vulnerability that gets sent to the API endpoint and stored. This payload should be crafted to include malicious HTML or JavaScript. When the client-side JavaScript fetches this stored data from the API and uses innerHTML to render it, the XSS will be executed for any user viewing the affected part of the application.

    Explanation: This shows how a reflected vulnerability can be a stepping stone to a stored XSS if the reflected input is subsequently stored and rendered unsafely.

36. **Question:** A web application allows users to customize their profile page with a short bio. This bio is sanitized on the server-side. However, the application also allows users to embed widgets from a predefined list using a specific syntax (e.g., {{widget:weather}}). If you can manipulate the parameters passed to these widgets (perhaps through a separate vulnerability), can you achieve stored XSS?

    Answer: If there's a way to control the parameters of a widget, and one of the widgets fetches and displays external content based on these parameters without proper sanitization, you could potentially inject a URL pointing to a page containing your XSS payload. When another user views the profile with the embedded (and manipulated) widget, the external content (with XSS) will be loaded and executed.

    Explanation: This highlights how even seemingly safe widget embedding features can become XSS vectors if the underlying data fetching or rendering based on user-influenced parameters is insecure.

37. **Question:** A web application uses server-sent events (SSE) to push real-time notifications to users. The content of these notifications is partially based on user-provided data that was stored previously. If you can inject a malicious payload into this stored data, how can you achieve stored XSS via SSE?

    Answer: Inject your XSS payload into the user-provided data that is later used to construct the SSE notifications. When the server pushes these notifications to other users, and the client-side JavaScript handling the SSE events directly renders the notification content into the DOM, the XSS will be executed.

    Explanation: Stored data can lead to XSS in various output contexts, including real-time communication channels like SSE.

38. **Question:** A web application has a contact form. The submitted messages are stored and displayed to administrators in an admin panel. The application uses a server-side library to prevent HTML in the messages. However, it doesn't prevent Unicode control characters. Can you achieve stored XSS in the admin panel?

    Answer: Inject a message containing Unicode control characters that, when rendered in the admin panel's context (potentially with different rendering libraries or browser interpretations), could be misinterpreted or bypass the intended sanitization, potentially allowing the injection of HTML or JavaScript. This is a more subtle and browser/rendering-dependent attack.

    Explanation: Sanitization needs to consider more than just standard HTML tags and attributes. Unicode and other special characters can sometimes be used to bypass filters or exploit rendering engine quirks.

39. **Question:** A web application allows users to collaborate on documents. Changes are tracked and displayed in a revision history. The application uses a diffing algorithm to show the changes between versions. If you can inject a specific payload into a document that gets stored and then viewed in the revision history, can you achieve XSS?

    Answer: Inject a payload that, when processed by the diffing algorithm and rendered in the revision history view, results in the injection of malicious HTML or JavaScript. This might involve crafting input that manipulates how the diff is calculated and displayed.

    Explanation: Even displaying historical data with diffing can be a source of XSS if the process of generating and rendering the differences isn't secure.

40. **Question:** A web application uses a feature where users can embed content from whitelisted external domains via iframes. The URL of the iframe is constructed based on user-provided input, but it's supposed to be restricted to the whitelisted domains. If you can find a way to manipulate the URL construction to bypass the domain whitelist, how can you achieve a form of stored (indirect) XSS?

    Answer: Exploit a vulnerability in the URL construction logic to point the iframe's src attribute to a domain you control that hosts an XSS payload. While the script executes within the iframe's context, it could potentially interact with the parent window depending on the application's security settings and browser behavior. This is an indirect stored XSS as the malicious URL is stored and served to other users.

Explanation: Even when embedding external content, improper validation of the source URL can lead to the inclusion of malicious content.

41. **Question:** A web application has a feature that allows exporting user data as a CSV file. This CSV file can later be imported back into the application. If you can inject a specific payload into your user data, can you achieve stored XSS when the CSV is imported and processed by client-side JavaScript?

    Answer: Craft your user data to include a CSV value that, when imported and processed by client-side JavaScript (perhaps for display or further manipulation), is interpreted as HTML or JavaScript. This might involve exploiting how the CSV data is parsed and rendered on the client-side.

    Explanation: Data exported and later imported can be a vector for stored XSS if the client-side import process doesn't properly handle potentially malicious content.

42. **Question:** A web application uses a client-side search index (e.g., using a library like Lunr.js) built from stored user-generated content. If you can inject a specific payload into your content that gets indexed, can you achieve XSS when other users perform searches that include your malicious content in the results?

    Answer: Inject a payload into your content that, when included in the search index and later displayed in search results, is rendered as HTML or executes JavaScript. This depends on how the search results are displayed and whether the indexing process inadvertently preserves or introduces exploitable content.

    Explanation: Client-side search functionalities based on stored data can be vulnerable if the display of search results doesn't properly sanitize the indexed content.

43. **Question:** A web application allows users to create polls with custom options. These options are stored and displayed to other users. If you can inject a malicious payload into a poll option, can you achieve stored XSS when other users view the poll?

    Answer: Inject an XSS payload directly into a poll option. When other users view the poll, if the application renders the poll options without proper sanitization (e.g., using innerHTML), the malicious script will execute in their browsers.

    Explanation: Simple user-generated content like poll options can be a direct vector for stored XSS if not handled securely during rendering.

44. **Question:** A web application uses browser notifications. The content of these notifications is generated based on stored user preferences. If you can manipulate your preferences to include a malicious payload, can you achieve XSS when these notifications are displayed?

    Answer: Inject your XSS payload into your user preferences. If the client-side JavaScript that generates the notification content from these preferences doesn't properly sanitize the data, the malicious script will execute when a notification is triggered based on your (now malicious) preferences.

    Explanation: Even browser-level features like notifications can be exploited if the data used to generate their content comes from user-controlled and unsanitized sources.

45. **Question:** A web application uses a feature to embed social media posts using their URLs. If you can find a way to provide a URL that, when embedded, renders content within the application's

context that allows JavaScript execution (either directly from the social media platform or through a manipulation of the embedding process), can you achieve a form of stored XSS?

Answer: This is highly dependent on the specific social media platform and the web application's embedding mechanism. If the social media platform itself has vulnerabilities that allow injecting malicious content that gets rendered when embedded, or if the embedding process is flawed and allows injecting arbitrary HTML attributes (like onload), it might be possible. This is a more indirect and less likely scenario but highlights the risks of embedding external content without strict security measures.

Explanation: Embedding content from third-party sources introduces trust dependencies. Vulnerabilities on the embedded platform or in the embedding process can lead to security issues in the host application.