SCENARIO

The application possesses multiple independent vulnerabilities, including cache key injection. We will try to take advantage of the cache's normalization process to exploit this vulnerability. We will try to exploit the XSS vulnerability and inject a payload that will execute alert(1) in the victim's browser. We will need to make use of the Pragma: x-get-cache-key header

**PROCEDURE**

1. Open the web application and send the request for **homepage** to BurpSuite’s Repeater.
2. Observe that the redirect at **/login** excludes the parameter **utm\_content** from the cache key using a flawed regex. This allows us to append arbitrary unkeyed content to the lang parameter.
3. Observe that the page at **/login/** has an import from **/js/localize.js** which is vulnerable to client-side parameter pollution via the lang parameter because it doesn't URL-encode the value.
4. Observe that the login page references an endpoint at **/js/localize.js** that is vulnerable to response header injection via the **Origin** request header, provided the **CORS** parameter is set to 1.
5. Use the Pragma: x-get-cache-key header to identify that the server is vulnerable to cache key injection, meaning the header injection can be triggered via a crafted URL.
6. Now we need to combine these four behaviours by poisoning the cache with following two requests Payload one after another.
7. Send the malicious request after removing the cache buster parameter and keep replaying the request until we see our exploit server URL being reflected in the response and **X-Cache: hit** in the headers.

**PROOF OF CONCEPT**

****

**PAYLOAD**

1. GET /js/localize.js?lang=en?utm\_content=z&cors=1&x=1 HTTP/1.1

Origin: x%0d%0aContent-Length:%208%0d%0a%0d%0aalert(1)$$$$

1. GET /login?lang=en?utm\_content=x%26cors=1%26x=1$$Origin=x%250d%250aContent-Length:%208%250d%250a%250d%250aalert(1)$$%23 HTTP/1.1

**REMEDIATION**

1. **Key All Input Parameters:** Ensure that all parameters, including those in the query string and headers, are part of the cache key. This prevents an attacker from injecting malicious content into cache responses.
2. **Limit Use of Dynamic Headers:** Headers like Origin shouldn't be reflected or used in responses without strict validation. If such headers are needed for functionality (like CORS), validate against a whitelist of allowed origins.
3. **Avoid URL Redirections:** Limit or avoid URL redirection based on user input. If unavoidable, ensure user inputs are strictly validated and sanitized.
4. **Escape Output:** Always escape user-controlled input that is reflected in HTML or JavaScript to prevent cross-site scripting (XSS) attacks.
5. **Sanitize Inputs:** Implement strict input validation for all user-controlled data. Ensure data conforms to expected formats, ranges, and values.
6. **Content Security Policy:** Implement a strong Content Security Policy (CSP) to mitigate the risk of cross-site scripting and other code injection attacks. A good CSP restricts sources of scripts and other resources, mitigating the risks posed by injection attacks.