CRLF Injection

Exploiting CRLF Injection: HTTP Request Example

To exploit CRLF injection, an attacker crafts a malicious HTTP request that includes CRLF sequences in user-controllable parts of the request, such as query parameters, headers, or cookies. Let's break down how a client can exploit this with a detailed example.

Scenario: Vulnerable HTTP Header Handling

Suppose we have a web application that reads a user-supplied value from a query parameter and inserts it into an HTTP header. This application is vulnerable to CRLF injection because it does not sanitize the input.

Example Code (Vulnerable)

```
from flask import Flask, request, make_response
app = Flask(__name__)
@app.route('/vulnerable')
def vulnerable():
    user_input = request.args.get('user_input')
    response = make_response("Vulnerable Page")
    response.headers['Custom-Header'] = user_input
    return response
```

In this example, the value of `user_input` is directly inserted into the `Custom-Header` HTTP header.

Exploitation via HTTP Request

An attacker can exploit this vulnerability by crafting an HTTP request with malicious input in the `user_input` parameter. Here's how:

Malicious HTTP Request

```
GET /vulnerable?user_input=attacker%0D%0AContent-
Length:%200%0D%0A%0D%0A<script>alert('XSS')</script> HTTP/1.1
Host: victim.com
```

Explanation:
- `attacker%0D%0A` translates to `attacker\r\n`, which ends the `Custom-Header`.
- `Content-Length: 0%0D%0A%0D%0A` sets the `Content-Length` to 0, followed by an extra CRLF sequence, effectively ending the HTTP headers.
- `<script>alert('XSS')</script>` is the injected malicious content.

Resulting Response

When the server processes this request, it mistakenly interprets the injected CRLF sequences as the end of the header section and the start of the body. The response might look something like this:

```
HTTP/1.1 200 OK
Custom-Header: attacker
Content-Length: 0

<script>alert('XSS')</script>
```

Here, the browser interprets the injected script as part of the HTTP response body and executes it, leading to an XSS attack.

Mitigation Strategies

To prevent such attacks:
1. Sanitize User Input: Ensure that user inputs are sanitized to remove CRLF sequences.

```
from flask import Flask, request, make_response
import re

app = Flask(__name__)

@app.route('/secure')
```

```
def secure():
    user_input = request.args.get('user_input')
    # Only allow alphanumeric and basic punctuation, reject inputs with CRLF characters
    if re.search(r'[\r\n]', user_input):
        return "Invalid input", 400
    response = make_response("Secure Page")
    response.headers['Custom-Header'] = user_input
    return response
```

2. Validate Headers: Ensure headers do not contain CRLF sequences before including them in the response.
3. Encode Output: Properly encode user input when inserting it into HTTP headers or other sensitive parts of the application.

Detecting CRLF Injection Vulnerabilities in Your Website

1. Identify Entry Points
Identify all parts of your application where user input is processed and used in HTTP headers, logs, or other sensitive contexts. Common entry points include:
- URL query parameters
- Form fields
- HTTP headers (like cookies, user-agent)
- Any user-supplied data stored and processed by the application
2. Manual Testing
Perform manual testing by injecting CRLF sequences (`%0D%0A` for URL-encoded `\r\n`) into these input fields and observing the behavior of the application.
Example:
> Inject CRLF in URL:
http://yourwebsite.com/vulnerable?input=test%0D%0ASet-Cookie:%20malicious=true
> Analyze Response:
  - Check if the `Set-Cookie` header is set in the response.
  - Look for any anomalies in the response headers or body.

3. Automated Scanning Tools
Use automated web vulnerability scanners that can detect CRLF injection. Some popular tools include:

- Burp Suite: A comprehensive web vulnerability scanner that can identify CRLF injection among other vulnerabilities.
- OWASP ZAP (Zed Attack Proxy): An open-source tool for finding vulnerabilities in web applications.

Example: Using Burp Suite

1. Intercept Traffic: Configure Burp Suite to intercept the traffic between your browser and the web application.
2. Send Requests to Repeater: Identify requests with user input and send them to the Repeater tool in Burp Suite.
3. Modify Requests: Modify the parameters with CRLF sequences and analyze the responses.
4. Active Scan: Use the Active Scan feature to automatically test for CRLF injection.

Mitigation Verification

1. Sanitize Input:
   - Ensure your application properly sanitizes user inputs to strip CRLF characters.
   - Use the sanitized input in headers and other critical areas.
2. Test After Fixes:
   - After implementing input sanitization, repeat the pentesting steps to verify the vulnerability is fixed.