ChatGPT

CRLF Injection in Web Applications

CRLF injection is a web vulnerability where an attacker is able to insert **carriage return (CR)** and **line feed (LF)** characters (`\r\n`) into HTTP headers or other output that uses them. HTTP/1.1 uses the CR+LF sequence to terminate header lines, so an injected CRLF can prematurely end one header and start another. For example, an attacker who supplies `%0D%0A` (the URL-encoded form of `\r\n`) in input that is reflected in a header can split the HTTP response into multiple header blocks or inject new headers. OWASP defines a CRLF injection as an attack "when a user manages to submit a CRLF into an application," commonly via an HTTP parameter or URL [1]. Imperva similarly explains that inserting extra CRLF into headers lets attackers "modify the header or inject malicious content," effectively splitting the HTTP response and controlling what follows [2]. In short, CRLF injection breaks the expected header/body boundary, enabling attacks like HTTP response splitting and log manipulation.

How CRLF Injection Works

In practice, CRLF injection is straightforward: an attacker finds an input field or URL parameter that ends up in the HTTP response headers, and injects `\r\n` into it. As one guide notes, the injection mechanism is "relatively straightforward" – the attacker inserts the `%0D%0A` sequence in user input, which causes arbitrary new headers to be injected or the response to split [3]. For example, consider a web app that reflects a user parameter into a `Location:` redirect header. If the user supplies a value containing `%0d%0a`, the server may output something like:

```
HTTP/1.1 302 Found
Location: /path?foo=value%0d%0aSet-Cookie:session=abcd
Content-Type: text/html
```

Here the injected `%0d%0a` ends the `Location:` header and begins a new `Set-Cookie:` header. In general, by inserting CRLF in a header context, attackers can create extra headers, control redirections, or even start a new HTTP response. HackTricks gives many examples of this, e.g. using `/…%0d%0aSet-Cookie:mycookie=myvalue` to test if a new cookie appears [4]. Because HTTP proxies and browsers interpret CRLF as header separators, even a single injected newline can have dramatic effects. As Imperva explains, "By injecting extra CRLF sequences into the input data, the attacker can split the HTTP response header and control the data that follows it" [2].

CRLF injection can also be used in logs: if an application includes user input in log files, an attacker's CRLF can fake log entries or hide actions [5] [6]. In summary, any place user input goes into an HTTP header or similar format without filtering is a potential CRLF injection point.

Common Impacts

A successful CRLF injection can lead to a range of serious attacks. In HTTP response splitting, attackers insert arbitrary headers (such as `Set-Cookie`, `Location`, or `Content-Type`) by terminating one header and starting another [3] [7]. This can be used to:

- **Inject HTTP Headers (e.g. Cookies):** Set malicious cookies on victims or overwrite security headers. Imperva notes that by manipulating headers, an attacker can cause "cookie injection" or "set arbitrary HTTP headers" [8] [9]. For example, supplying `/%0d%0aSet-Cookie:evil=1` in a URL might cause the server to set that cookie on the user's browser.
- **Cross-Site Scripting (XSS):** Split the response to inject a `<script>` payload. For instance, the injected CRLF can allow crafting a fake HTTP response containing a malicious script [10]. Imperva explains that CRLF attacks can enable XSS by injecting scripts into the victim's browser through header manipulation [11] [8].
- **Open Redirect / Phishing:** Overwrite a `Location` header to redirect users to an attacker-controlled site. By injecting a new `Location:` line, an attacker can hijack an application's redirect flow. This can lead to phishing or theft of credentials [8] [12].
- **Session/Authentication Hijack:** Manipulate `Set-Cookie` to hijack sessions or fixate sessions. As Praetorian notes, inserting a `Set-Cookie` header can allow session fixation or account takeover [7].
- **Web Cache Poisoning:** Since caches index responses by URL and headers, adding unexpected headers can poison a cache. Imperva explicitly lists cache poisoning as a consequence of CRLF injection [8].
- **Log Forgery / Tampering:** If user input is logged, injected CRLF can create fake log lines. For example, an attacker could hide their tracks or incriminate another IP by splitting a log entry into multiple entries [9]. The hacker GeeksforGeeks example shows how `%0d%0a` can add log lines, misleading administrators [9] [13].
- **Server-Side Request Forgery (SSRF) / RCE:** In some contexts (especially CGI scripts), CRLF injection can chain into SSRF or even remote code execution. For instance, the Orange Tsai blog demonstrates how a CRLF-injectable CGI redirect could be abused to call arbitrary Apache handlers (like `mod_proxy` or `mod_php`) via a crafted `Content-Type` header, leading to SSRF or RCE [14] [15]. They warn that CRLF bugs are often misreported as XSS but **can** enable powerful server-side attacks [16].
- **Other Side Effects:** Attackers may bypass security filters (e.g. disable XSS filters via a CRLF hack [17]) or break same-origin policies by injecting extra content [18] [2].

In short, CRLF injection is a gateway to numerous attacks by giving the attacker *control of the HTTP response*. As one security podcast notes, when exploitable, CRLF injection "can allow you to overwrite existing headers, and introduce new headers – it's essentially full HTTP response control" [19].

Locating CRLF Injection – Black-Box Testing

When testing an application externally (black-box), the approach is to fuzz inputs and watch HTTP responses for anomalies. Key steps include:

- **Identify Candidate Inputs:** Find fields or parameters that influence HTTP headers or redirect values. For example, parameters used in `Location:` redirects, `Set-Cookie`, or any header-

producing code. In many CTFs, fields like `name` or `target` (as in the given script) that end up in a `Location:` header are prime suspects.

- **Inject Encoded CRLF:** Submit payloads containing `%0d` (CR) and `%0a` (LF). For example, try adding `%0d%0aSet-Cookie:evil=1` to a URL or form input. HackTricks suggests test strings like `/%0D%0ASet-Cookie:mycookie=myvalue` or including `%0d%0aLocation:` to see if the response sets a cookie or changes redirect location [4]. Because some servers strip literal newlines, using URL-encoded `\r\n` is common.
- **Inspect Raw Responses:** Use an intercepting proxy (Burp, OWASP ZAP) or `curl -v` to view raw HTTP headers. If a response header block suddenly splits into two (empty line in the middle), or you see your injected header, you've found CRLF injection. For instance, adding `%0d%0aX-Test:1` might yield an extra `X-Test: 1` header in the output.
- **Automated Scanners:** Tools like Burp Scanner, OWASP ZAP, or specialized tools can automate testing many payloads. A recent Medium article recommends using dynamic scanners (Burp Suite, ZAP, Nessus) for CRLF among other injection tests [20]. There are even specific tools: e.g. *crlfuzz* (a Go-based fuzzing tool) scans for CRLF injection in URLs or headers [21], and the Nuclei template engine has CRLF test payloads. Praetorian's Chariot team reported using the Nuclei scanner to discover a CRLF injection by automatically inserting `%0ASet-Cookie:test=test` in a request [22].
- **Use Timing or Behavior Indicators:** Sometimes header injection won't break content but can change behavior (e.g. click-through flows, cookies). Check for unusual cookies set or redirects occurring. Turn off automatic redirect following and see raw 302 responses.
- **Check Logs (if accessible):** If you have access to server logs or admin panels, see if your injected input created fake log lines (e.g. your IP twice, or strange entries). This can confirm a log-based CRLF injection.

By systematically sending CRLF payloads and examining the HTTP response (and logs), black-box testers can uncover CRLF flaws. The HackTricks "cheat sheet" even lists example payloads for testing response splitting and XSS via CRLF [4]. For example, one might try a URL like `http://example.com/?q=%0d%0aLocation:%20http://evil.com` and see if the `Location` header changes. Tools like Burp's intruder and Repeater are very helpful here.

Locating CRLF Injection – White-Box (Code Review)

In code review or source analysis, one looks for any place that user input could end up in HTTP headers **without neutralizing CRLF**. Key strategies:

- **Search for Header Output:** In the codebase, find all statements that write HTTP headers or redirects using user data. This includes calls like `print("Location: ...")` in CGI scripts, `Response.Headers.Add` or `res.setHeader` in web frameworks, string templates for headers, etc. In Python CGI or Flask/Django, look for `print(f"Location: {user_input}")` or `HttpResponseRedirect(...)`. In Node/Express, check for `res.setHeader()` or `res.redirect()` using user-controlled values.
- **Review Input Sanitization:** See how those input values are sanitized. Are CR or LF characters removed or encoded? For example, in the provided Python script the `name` parameter is used directly in `print(f'Location: ...{name}...')` with no filtering, which is unsafe. White-box auditors should flag any header-building that directly interpolates unsanitized input. As Praetorian notes, a common root cause is "when an HTTP header within a server's response body reflects attacker-controlled input" [23].

- **Static Analysis Tools:** Use linters or static analyzers configured to detect header injection patterns. For instance, one can use ESLint or SonarQube with security rules to scan for header-setting code (Land2Cyber suggests ESLint and SonarQube as examples [20]). Specialized SAST tools (e.g. Checkmarx, Fortify, or open-source bandit for Python) often include CWE-113 (CRLF injection) checks. Documentation for tools like Syhunt explicitly shows that CRLF injection is checked in both black-box and source-code (white-box) modes for Python, PHP, etc [24].
- **Code Patterns:** Grep or search for `\r` or `\n` in string literals or for functions that write headers. Also inspect any use of low-level functions (e.g. `os.popen` with user data, `urllib` with user URLs) – these may inherit language-specific CRLF issues. For example, the legacy `urllib2` in Python 2 did not escape CRLF in URLs, leading to CVE-2019-9740 as shown in the NVD [25]; code review should look for such outdated usage.
- **Review Domain/URL Validation:** The example script had an `is_domain` regex for `target`, but it still treated `name` freely. Ensure any domains or URLs are validated to not contain CRLF. If user input is supposed to be a domain or path, use strict regex (e.g. `^(?!-)[A-Za-z0-9-]{1,63}(?<!-)\.[A-Za-z]{2,63}$` as in the example) and also strip `\r\n` [26].
- **Unit Tests:** As a white-box tester, write unit tests that simulate inputs containing `%0d%0a` and verify that the generated HTTP response does not contain new unintended headers. This is a practical way to catch CRLF at development time.

In summary, white-box detection relies on tracing user input through the code paths that produce HTTP responses. Any sink that writes to headers should be reviewed. Static scanners like Sonar or CAST (which explicitly warns about CRLF in Python headers [26]) can automate this. Remember that even languages that normally protect headers (like some frameworks) can be vulnerable if developers bypass them.

Language- and Framework-Specific Notes

- **Python:** Plain CGI scripts (using `print` for headers) are common culprits. The example code is CGI-based and vulnerable. More generally, old Python libraries had known CRLF bugs: for instance, CVE-2019-9740 showed that `urllib.request.urlopen` did not encode `\r\n` in URLs (allowing header injection if the query string was attacker-controlled) [25]. This has been fixed in recent Python releases, but it highlights the need to use up-to-date interpreters. In frameworks like Flask or Django, use the provided redirect and header functions (they typically escape headers) and avoid manually concatenating user data into headers.
- **JavaScript/Node.js:** Similar issues exist. In Feb 2023, Node.js patched a security issue where the new `fetch` API did not sanitize the `Host` header against CRLF injection [27]. This underscores that libraries can introduce vulnerabilities. When writing Node code, avoid setting headers from raw user input. Use `res.setHeader()` or `res.redirect()`, which internally check for illegal characters, and always validate or encode input. Libraries like `undici` or `axios` should be updated as needed.
- **CGI (Perl, etc.):** Many CGI apps (as Orange Tsai describes) simply `print` headers. The excerpt shows a Perl CGI that does `print "Location: $redir\n"; print "Content-Type: text/html\n\n";`. If `$redir` contains `%0d%0a`, it injects a new header [28]. Thus, CGI code in any language must sanitize. Use frameworks or modules that handle headers for you, or explicitly strip CR/LF from user strings.
- **Other Frameworks:** In PHP, .NET, Ruby, etc., ensure you use the language's safe header functions (e.g. `header()` in PHP with proper escaping, `.redirect()` in Rails). Check documentation: many

high-level frameworks automatically forbid newline characters in header values, but custom code or older libraries might not.

- **Security Filters and WAF:** Some web application firewalls (WAFs) try to block CRLF injection patterns. For instance, Praetorian discovered an Akamai WAF that could be bypassed via creative content-encoding and CRLF injection [29]. Do not rely solely on WAF; fix the code. However, as defense-in-depth, a WAF or RASP that sanitizes headers at runtime can block blatant CRLF sequences.

Mitigation Strategies

The primary rule is **never put unsanitized user input into HTTP headers**. Specific mitigations include:

- **Encode or Remove CR/LF:** Strip carriage returns and line feeds from any user-provided data before using it in a header. For example, one secure coding guideline explicitly says to remove `\r` and `\n` from header values [26]. In the Python fixed sample from [9], they do `param_safe = re.sub('[\n\r]', '', tainted)`, effectively sanitizing out newlines [30]. Similarly, HackTricks advises encoding CRLF if input must be in a header [31] (e.g. percent-encode them or use language-safe encoding functions).
- **Use Safe API Functions:** Instead of concatenating strings, use framework methods that validate headers. For example, in Python Flask use `redirect()` or set headers through `Response` objects. In Express, use `res.redirect()` or `res.setHeader()`, which throw an error if the header value is invalid. These APIs often handle newline encoding automatically. If your language/platform has a known fix (as Node's `fetch` update or Java's `URLEncoder` fixes), apply those updates [27] [25].
- **Strict Input Validation:** If user input must be used (e.g. a hostname), validate it against a strict whitelist or regex. The example script's `is_domain()` regex is a good start, but it should also strip CRLF. Validating the format (only letters, digits, hyphen, dot) prevents control characters. Any deviation (including `\r` or `\n`) should be rejected.
- **Security Reviews and Testing:** Include CRLF cases in your test plans. Land2Cyber recommends input validation via regex to filter CRLF [32] and regular automated testing (static and dynamic) as part of the SDLC [33] [34]. For instance, a CI pipeline could run a static scan (like SonarQube) and a dynamic scan (like OWASP ZAP) on new code.
- **Use HTTP Security Headers:** While not a direct fix, headers like CSP and HSTS (cited by [19]) can limit the impact of some CRLF exploits by restricting the contexts where injected content can run [35]. However, these are supplementary – the core fix is sanitization.
- **Firewall/RASP:** As a last defense, a WAF or RASP can catch raw CRLF patterns. Imperva and others offer such protections [36]. But because clever encodings can bypass WAF rules, do not assume it's foolproof. Fix the root cause in code.

In summary, treat any user input in headers as potentially dangerous. Always strip or encode `\r`/`\n`, use framework header methods, keep libraries patched, and validate input format. As HackTricks succinctly advises: **"Do not use user input directly inside response headers...[and] use a function to encode CRLF special characters"** [31]. CAST's secure coding rule likewise recommends **"Remove CR, LF characters for values coming from user inputs and used in HTTP headers"** [26]. Following these best practices will prevent CRLF injection vulnerabilities.

Sources: Definitions and examples from OWASP and security blogs [1] [2]; risks and impacts from Imperva, GeeksforGeeks, and Praetorian research [8] [3] [7]; testing techniques from Pentest write-ups

and tools [4] [20] [22]; language-specific advisories [25] [27]; and mitigation guidelines from HackTricks and CAST [31] [26].

---

[1] CRLF Injection | OWASP Foundation
https://owasp.org/www-community/vulnerabilities/CRLF_Injection

[2] [3] [5] [8] [9] [11] [36] What is CRLF Injection | Types & Prevention Techniques | Imperva
https://www.imperva.com/learn/application-security/crlf-injection/

[4] [17] [21] [31] CRLF (%0D%0A) Injection | HackTricks - Boitatech
https://hacktricks.boitatech.com.br/pentesting-web/crlf-0d-0a

[6] [14] [15] [16] [28] Confusion Attacks: Exploiting Hidden Semantic Ambiguity in Apache HTTP Server! | Orange Tsai
https://blog.orange.tw/posts/2024-08-confusion-attacks-en/

[7] [22] [23] [29] Bypassing Akamai's Web Application Firewall Using an Injected Content-Encoding Header | Praetorian
https://www.praetorian.com/blog/using-crlf-injection-to-bypass-akamai-web-app-firewall/

[10] [12] [13] CRLF Injection Attack - GeeksforGeeks
https://www.geeksforgeeks.org/linux-unix/crlf-injection-attack/

[18] [20] [32] [33] [34] [35] “Automating CRLF Injection Testing” Tools and Techniques | by Land2Cyber | Medium
https://medium.com/@Land2Cyber/automating-crlf-injection-testing-tools-and-techniques-19813e8c7f7a

[19] [HackerNotes Ep. 59] Bug Bounty Gadget Hunting & Hacker's Intuition
https://blog.criticalthinkingpodcast.io/p/useful-gadgets-when-bug-bounty-hunting

[24] Vulnerability Checks | Syhunt Web Application Security Docs
https://www.syhunt.com/docwiki/index.php?n=SyhuntHybrid4.Vulnerabilities

[25] NVD - cve-2019-9740
https://nvd.nist.gov/vuln/detail/cve-2019-9740

[26] [30] Avoid HTTP header injection (Python) | CAST Appmarq
https://www.appmarq.com/public/security,1021098,Avoid-HTTP-header-injection-Python

[27] Node.js — Thursday February 16 2023 Security Releases
https://nodejs.org/fr/blog/vulnerability/february-2023-security-releases