YLChen-007 / **ISSUE-Github-REPORT-SSRF-LFI-Urlopen-Audio-Playback.md**

Secret

Created last month

Code Revisions 1

SSRF + Local File Inclusion (LFI) via urllib.request.urlopen() in AgentScope Audio Playback

`ISSUE-Github-REPORT-SSRF-LFI-Urlopen-Audio-Playback.md`

Advisory Details

Title: Blind SSRF + Local File Inclusion + Denial of Service via `urllib.request.urlopen()` in Audio Playback

Description:

Summary

`AgentBase._process_audio_block()` uses `urllib.request.urlopen()` to fetch audio URLs without any validation. Unlike `requests.get()` (used elsewhere in AgentScope), `urllib.request.urlopen()` natively supports the `file://` protocol, enabling Local File Inclusion (LFI) in addition to standard HTTP-based SSRF.

This is a **blind SSRF** — the fetched content is consumed locally for audio playback on the server's sound device and is **not returned to the attacker**. However, the `file://` protocol support extends the attack surface beyond what the other SSRF sinks in AgentScope offer.

Critically, because `response.read()` is called **without a size limit**, pointing it at an infinite stream device like `file:///dev/urandom` causes **unbounded memory growth** that permanently blocks the thread and eventually triggers an OOM kill — a reliable, single-request Denial of Service.

Details

The vulnerable code is in `src/agentscope/agent/_agent_base.py`, method `_process_audio_block()` (lines 272–317). When an audio content block has `source.type == "url"`, the URL is fetched directly with `urllib.request.urlopen()`:

```
def _process_audio_block(self, msg_id: str, audio_block: AudioBlock) -> None:
    ...
    if audio_block["source"]["type"] == "url":
        import urllib.request
        import wave
        import sounddevice as sd

        url = audio_block["source"]["url"]              # ← no validation
        try:
            with urllib.request.urlopen(url) as response: # ← SSRF + LFI sink
                audio_data = response.read()

            with wave.open(io.BytesIO(audio_data), "rb") as wf:
                ...
                sd.play(audio_np, samplerate)            # ← data consumed l
                sd.wait()

        except Exception as e:
            logger.error("Failed to play audio from url %s: %s", url, str(e))
```

The method is invoked from `AgentBase.print()` (lines 254–258) via the `speech` parameter:

```
async def print(self, msg, last=True, speech=None):
    ...
    if isinstance(speech, list):
        for audio_block in speech:
            self._process_audio_block(msg.id, audio_block)     # ← sink invoc
    elif isinstance(speech, dict):
        self._process_audio_block(msg.id, speech)
```

`ReActAgent._reasoning()` extracts audio blocks from LLM responses and passes them as `speech`:

```
# _react_agent.py line 596
speech = msg.get_content_blocks("audio") or None
await self.print(msg, False, speech=speech)
```

The key difference from the other SSRF sinks (which use `requests.get()`):

Property	`requests.get()`	`urllib.request.urlopen()`
HTTP/HTTPS	✅	✅
`file://` protocol	❌ raises `InvalidSchema`	✅ **reads local files**
`ftp://` protocol	❌	✅ supported

No URL validation exists: no scheme check, no private IP blocking, no allowlisting.

PoC

This is a **blind SSRF** — the server fetches the URL, but the response data is consumed locally for audio playback (passed to `wave.open()` then `sounddevice.sd.play()`). The attacker cannot read the response content. However, the server-side request itself is confirmed.

Attack scenario — LFI via `file://`:

1. A developer deploys an AgentScope application with a `ReActAgent` that processes audio content blocks. The `print()` method is called during agent reasoning.

2. An attacker supplies a message (or influences the agent's conversation) containing a malicious audio block:

```
malicious_msg = Msg(
    name="assistant",
    role="assistant",
    content=[
        AudioBlock(
            type="audio",
            source={
                "type": "url",
                "url": "file:///etc/passwd",
            },
        ),
    ],
)
```

3. When `_process_audio_block()` processes this block, `urllib.request.urlopen("file:///etc/passwd")` reads the local file contents into `audio_data`.

4. The data is then passed to `wave.open()`, which expects valid WAV format. For non-WAV files (like `/etc/passwd`), this raises an exception caught by the generic `except` handler. The error is logged server-side via `logger.error()`.

What the attacker CAN observe:

- Error-based side channels: different exceptions for `file:///etc/passwd` (exists, read succeeds but WAV parse fails) vs `file:///etc/nonexistent` (FileNotFoundError) leak file existence information.
- Timing differences for internal HTTP targets reveal port/host reachability.

What the attacker CANNOT do:

- ❌ Read the file contents — data goes to `wave.open()`, not back to the attacker
- ❌ Exfiltrate HTTP response bodies — data is consumed by `sd.play()` on server speakers
- ❌ Steal cloud credentials — metadata responses stay server-side

Minimal reproduction (simulates the sink in isolation):

```
import urllib.request

# This is what _process_audio_block() does internally
url = "file:///etc/passwd"
with urllib.request.urlopen(url) as response:
    data = response.read()
    print(f"Read {len(data)} bytes from {url}")
    # data is consumed by wave.open() next — NOT returned to caller

# For comparison, requests.get() blocks file:// entirely:
import requests
try:
    requests.get("file:///etc/passwd")
except Exception as e:
    print(f"requests.get blocked: {type(e).__name__}")
```

Impact

This is a **blind SSRF with `file://` protocol support**. The fetched data is NOT returned to the attacker (it's consumed by `wave.open()` → `sd.play()` for local audio playback). The impact is:

1. **Denial of Service via unbounded memory consumption (most severe)** — Because `response.read()` is called without a size limit, pointing it at `file:///dev/urandom` (an infinite random byte stream with no EOF) causes `.read()` to **never return**. Memory grows without bound until the Linux OOM killer terminates the process. This is a reliable, single-request DoS that requires no authentication:

   ```
   urlopen("file:///dev/urandom")
       → .read() starts consuming /dev/urandom
       → /dev/urandom never emits EOF
       → .read() never returns, memory grows ~100MB/s
       → OOM kill → service crash
   ```

   The same attack works with `file:///dev/zero` (infinite zero-byte stream). Verified locally: `.read()` exhausts a 50MB memory-limited subprocess within seconds.

2. **File existence probing via `file://`** — The attacker can confirm whether arbitrary files exist on the server through error differentials (`wave.Error` for existing non-WAV files vs `FileNotFoundError` for missing files).

3. **Internal network reconnaissance / port scanning** — Timing and error differences between "connection refused", "timeout", and "invalid WAV format" reveal internal host/port reachability.

4. **Side-effects on internal services** — GET requests to internal APIs that trigger state changes on receipt.

What this vulnerability does NOT allow:

- ❌ Direct data exfiltration (fetched content goes to `wave.open()` / `sd.play()`, not the attacker)
- ❌ Reading local file contents (non-WAV files cause `wave.Error`, caught and logged locally)
- ❌ Stealing cloud credentials from metadata endpoints

Affected products

- **Ecosystem**: pip
- **Package name**: agentscope
- **Affected versions**: <= 1.0.14 (all versions with `_process_audio_block()`)
- **Patched versions**:

Severity

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

Weaknesses

- **CWE**: CWE-918: Server-Side Request Forgery (SSRF)
- **CWE**: CWE-400: Uncontrolled Resource Consumption

Occurrences

Permalink
https://github.com/modelscope/agentscope/blob/5e2bf63/src/agentscope/agent/_agent_b L317
https://github.com/modelscope/agentscope/blob/5e2bf63/src/agentscope/agent/_agent_b L258
https://github.com/modelscope/agentscope/blob/5e2bf63/src/agentscope/agent/_react_ag