

Look inside the apps you like with REA

REA (Reverse Engineer Anything) is an open-source tool that works out how an app does something without its source code. Installing it in Claude Code, asking the first question, a quick terminal scan, the Wispr Flow example, and bringing what you learn into your own project.

TIME	LEVEL	STEPS	PROMPTS
25 min	Intermediate	5	3

You see a feature in an app and wonder how they built it. You don't have the source code. This guide shows you how to connect REA to Claude Code and look inside an app. I tried it on Wispr Flow; below you'll find what I found and, honestly, where REA fell short. Hit **Copy** in the top right of any command or prompt, paste it, keep going.

What we're making

We'll pick a desktop app and answer "how does this feature work?". Then we'll build the same idea in your own project, with your own code.

What is reverse engineering? Taking a finished product apart to work out how it was made. In software, that means looking at an app's files without its source code and finding which part does what.

Every app is really a folder. On a Mac, right-click an app and choose **Show Package Contents** to see its files. Apps built with Electron, like Slack, Obsidian, VS Code, Claude and Wispr Flow, keep their code in that folder as JavaScript. It's hard to read (minified), but readable.

What does REA do? [REA](#) is an open-source (MIT) tool. It connects to your agent over MCP and can inspect:

- Electron and JavaScript apps: modules, imports, the app's internal messaging (IPC)
- Websites: page structure, scripts, network requests
- Compiled programs: through Hopper, Ghidra or IDA if you have one installed
- .NET, Android APKs, SQLite databases and a few other formats

Analysis runs on your machine. Every finding comes with its evidence (file, location, relationship) and what REA couldn't determine.

You'll end up with three things:

1. A step-by-step picture of how the feature you picked works
2. Evidence for every step (which file, which call)
3. A plan to build the same idea in your own project with your own code

INGREDIENTS

- [Claude Code](#). Codex, Cursor and Gemini CLI are supported too.
- Node.js 22.19 or newer, plus npm. Check with: `node -v`
- The app you want to look into. The example here is on a Mac; REA also works on Windows and Linux.

TOOLS

01 Install REA

In your terminal:

TERMINAL

```
npx rea-agents setup
```

Setup asks which agents to connect. Pick **Claude Code**. Before changing anything it shows which file it will write and asks for approval, and it backs up your existing configuration. It also offers to install Hopper; that's optional. If you only look at JavaScript/Electron apps, you don't need it.

When setup finishes, **quit and reopen Claude Code**.

You can also let Claude do the install:

PROMPT

Install the morluto/rea repo (REA: Reverse Engineer Anything) into Claude Code. The official install command is `npx rea-agents setup`. Show me which files setup will change before writing anything, and don't write without my approval. Tell me if I need to restart afterwards.

02 Ask your first question

After reopening Claude Code, use this template. Swap in your app and the feature you're curious about:

PROMPT

Use REA to inspect the Wispr Flow app (/Applications/Wispr Flow.app). Find out how, when I finish speaking, the text gets typed into the box where my cursor is.

1. Lay out the flow step by step: which part calls which.
2. Show evidence for every step: file path and the relevant function or message name. Don't copy long code; 2-3 lines is enough.
3. List separately anything you're unsure about or that REA couldn't see.
4. If there's a separate program running inside the app, point it out.

The narrower the question, the better the answer. Instead of "how does this app work?", ask "what happens when I press this button?".

03 Quick scan from the terminal (optional)

If you want the big picture before asking Claude, you can run REA straight from the terminal. In Electron apps the code lives in `app.asar`:

TERMINAL

```
npx -y rea-agents@latest analyze-javascript-application  
"/Applications/Wispr Flow.app/Contents/Resources/app.asar" --  
json > wispr.json
```

The output is large (59 MB for Wispr Flow). To see a summary:

TERMINAL

```
jq -c '{source: {kind: "inline", evidence: .}, view: {kind: "summary"}}' wispr.json > summary-request.json  
npx -y rea-agents@latest inspect-analysis-view summary-request.json
```

For me the scan took about 82 seconds and reported 167 JavaScript files, 1,428 modules and 207 internal calls (IPC).

04 What I found: how Wispr Flow types

In Wispr Flow you hold a shortcut key and speak; when you let go, the text appears in whichever app you're in. Inside, this is the flow:

1. **You speak.** The audio becomes text.
2. **AI cleans up the text.** If that step fails, the raw text is used (`PrePastePolish` in the code).
3. **Is the cursor in an editable box?** The app checks this first (`getTextBoxInfo`). If not, it shows the text as a notification instead of pasting.
4. **A small hidden program types the text.** Electron itself doesn't paste. Inside the package, in `swift-helper-app-dist`, there's a separate app written in Swift; it receives a `PasteText` request. If that program has stopped, it's relaunched (`pasteTextAndRelaunchHelperIfNeeded`).

5. **Did it actually land?** The text in the box is checked before and after pasting to confirm it really appeared.

You can see it yourself: Finder → Applications → right-click Wispr Flow → **Show Package Contents** → Contents → Resources → `swift-helper-app-dist`. There's a second "Wispr Flow" app in there.

Where REA fell short. REA's feature-trace command (`trace-application-feature`) didn't find this flow; all 6 starting points I tried returned 0 matches. REA itself said in its output that the map it built for this app was incomplete. We found the chain above by searching the packaged code for function and message names after REA's scan. So REA doesn't hand you the answer on its own; it's useful together with an agent. Check every step the agent gives you against its evidence.

05 Bring it into your own project

Once you understand the flow, build the idea with your own code:

PROMPT

Design the flow we found in Wispr Flow (prepare the text → check the target box is editable → have a small separate helper program do the typing → confirm it landed) for my own project.

1. Don't copy Wispr Flow's code; only use its architecture as a reference.
2. Suggest the simplest solution that fits my project's language and structure.
3. List separately which steps need an OS permission (e.g. Accessibility on a Mac).
4. Show me the plan before writing code.

This isn't only for speech-to-text apps. You can work out the architecture behind a Duolingo exercise, an Instagram flow, or a feature in your favorite desktop tool the same way.

First: what can you look into?

REA reads apps you installed on your own computer. Still:

- Use it to **learn**. Don't copy the code you find into your product; understand the idea and write your own.
- Check the app's terms of use. Some explicitly forbid reverse engineering.
- Don't use it to get around passwords, license checks or security measures.

TIPS

artifact_integrity_mismatch error. Some files inside an app can be signed or changed after packaging; REA notices and stops. If it's an app you installed and trust, add `--integrity-policy record-and-continue` to the command. REA records the difference and continues.

Node version error. REA needs Node.js 22.19 or newer. Check with `node -v` and update if it's older.

The app's code isn't JavaScript. Apps like Safari or Notes are compiled to machine code. For those, REA needs Hopper, Ghidra or IDA. You can accept the Hopper install during setup or point REA at an existing Ghidra.

The trace command returns 0 matches. That doesn't mean the feature isn't there; REA says so in its output. Ask your agent to search the packaged code

for function and message names after REA's scan.

The answer is too vague. Narrow the question: one button, one shortcut, one screen.

DONE?

- ☐ Node.js 22.19+ is installed
- ☐ I connected REA to Claude Code with `npx rea-agents setup` and restarted Claude Code
- ☐ I checked the terms of use of the app I'm inspecting
- ☐ I narrowed my question to one feature
- ☐ I saw the evidence (file, function, message name) for every step
- ☐ I noted separately what REA couldn't see
- ☐ I built the idea in my own project with my own code, without copying

If you get stuck, message me on Instagram and I'll add it to the guide.