



Reverse Engineer Anything with Claude Code

The guide that goes with the reel. Connect REA to Claude Code, get a written report on how an app you have installed is built, then have Claude Code build your own version of its main feature. Tested on Windows. The Mac steps are marked untested.

START HERE

Who this is for, and what you will have

You watched Claude Code rebuild a paid app's main feature in the reel. You use Claude Code and know the basics. You have never added an MCP server or a skill. This guide is written for that reader, and it explains both terms before you need them.

- ❑ REA connected to Claude Code: an MCP server that gives Claude Code tools for inspecting software, and a skill that tells Claude Code how to use them
- ❑ REPORT.md for one app you have installed: what it is built from, which open-source projects it ships, and how its main feature works, with the evidence for each finding
- ❑ A plain statement of what REA could and could not see on your computer, and a list of what could not be determined
- ❑ A way to tell whether a build is worth trying, taken from what made the app in the reel a simple case
- ❑ Optional: your own version of the main feature, built from new code and open-source parts, with self-tests and an end report that names what is untested

What the prompts build

The prompts work on one app that is installed on your computer. First they produce a report on how that app is built. That report is the outcome of this guide.

The last prompt is optional. It has Claude Code build your own version of the app's main feature: a new program, made from new code and open-source parts. The build prompt tells Claude Code to put none of the original app's code, artwork or text in your version. The prompts do not unlock a paid app, and they tell Claude Code to stop if the feature depends on a license check or a sign-in.

What was tested

One run, on 12 October 2026: one Microsoft Store app on one Windows PC, with Claude Code v2.1.296 in auto mode and REA 6.4.0. This guide calls it Jason's run. It is the run in the reel.

Jason typed one short prompt, and Claude Code made eight judgment calls to fill the gaps. The prompts in this guide state those calls up front. They were written from Jason's run and have not been run again as written. Mac is untested.

THE TOOL

What REA is

REA stands for Reverse Engineer Anything. It is an MCP server and a skill for Claude Code. Its home is github.com/morluto/rea, and its license is MIT.

The two parts

The MCP server. An MCP server is a program that gives Claude Code extra tools to call. REA 6.4.0 adds 170 tools for inspecting software. In a session, the name of each tool starts with `mcp__rea__`.

The skill. A skill is a set of written instructions for Claude Code. REA's skill tells Claude Code when and how to use the tools. The skill alone adds no tools.

REA's README describes the loop this way:

"Your agent calls REA through MCP to inspect the target and trace relevant code. REA returns findings with their evidence. The agent uses them to ask follow-up questions, explain the behavior, or write and test an implementation."

On 12 October 2026 the repository had 147,982 stars on GitHub (GitHub's API). REA changes fast: versions 6.0.0 to 6.4.0 all shipped between 8 and 11 October 2026. If a screen or a tool name differs from this guide, go by what REA shows you.

What REA needs for each kind of app

Kind of app	What REA's README says it needs
Native binaries	"Hopper, Ghidra or IDA"
JavaScript and Electron	"Node.js and npm"
.NET assemblies	"None (static only)"
Websites	"A Chrome-family browser"

Hopper, Ghidra and IDA are decompilers. A decompiler turns compiled code back into code a person can read. REA calls each one a provider. The app in the reel was made of native binaries, so the first row decided how much REA could read.

On Windows, expect no decompiler

REA's setup cannot install Hopper on Windows, and REA's Hopper tools do not run there. REA can use a Ghidra that you installed yourself: version 12.1.x, with a 64-bit JDK 21 or later. REA's docs mark that route experimental and read-only. IDA Pro is commercial software.

Jason's PC had none of the three set up. The report prompt in this guide is written for that case: it uses REA first, then falls back to what the app's files show without a decompiler.

Where your files go

"REA analyzes targets locally. Your agent receives the tool results, and its model provider has its own data policy."

That is from REA's FAQ. Your agent here is Claude Code.

What REA is for

"REA is for lawful research, education, interoperability, software preservation, and analysis of software you own or are authorized to examine."

That is from REA's README. REA's disclaimer lists uses that the project does not support. Two of them are "copying proprietary software or reproducing its protected code and assets" and "circumventing DRM, license checks, anti-cheat systems or other technical protection measures". The prompts in this guide do neither. The same disclaimer says: "Reverse engineering may be restricted in your jurisdiction or by the license of the software you analyze."

BEFORE YOU START

What to have, in the order to learn it

The first five rows are for every reader. The last four are for the optional build.

What	Why you need it	Where to learn it
Claude Code, signed in	It runs every prompt in this guide. You type one slash command, <code>/mcp</code> , in step 1. REA's site says Claude Code needs a paid Claude plan.	graniteai.co/tools/claude-code-commands , our reference to the slash commands
MCP servers	REA's tools reach Claude Code through an MCP server. Setup adds it for you in step 1.	code.claude.com/docs/en/mcp , Anthropic's page on MCP in Claude Code
Skills	REA's skill is a folder of instructions in your Claude Code skills folder. Setup adds it for you in step 1.	graniteai.co/tools/clean-up-claude-code-skills , our guide to the /skills list. Anthropic's page: code.claude.com/docs/en/skills
Permission modes	In the default mode, Claude Code asks you to approve most tool calls. Jason used auto mode, and Claude Code asked him nothing.	code.claude.com/docs/en/permission-modes , Anthropic's page on permission modes
Node.js with npm	REA runs on Node.js 22.19 or later, 24.11 or later, or 26 or later, and does not run on 23 or 25. REA's setup does not install Node.js.	nodejs.org/en/download
Build only: a compiler	Claude Code did not install one in Jason's run, and REA never installs one. For a C++ app on Windows, that means Visual Studio Build Tools 2019 or later with the C++ workload.	visualstudio.microsoft.com/visual-cpp-build-tools
Build only: git	Jason's build fetched its open-source parts with git.	git-scm.com/downloads
Build only: disk space and internet	Jason's build used about 3 GB for source code, dependencies and build output.	No primer needed

Build only: the real device

For Jason's app: an iPhone, iPad or Mac on the same network as the PC, a network set to Private, and the right to allow a program through Windows Firewall.

Step 6

Check Node.js

Print your Node.js version. Run in a terminal.

```
node -v
```

Windows: install Node.js if it is missing or too old. Run in PowerShell. Then close PowerShell, open it again and run `node -v`.

```
winget install OpenJS.NodeJS.LTS
```

Windows: if PowerShell says "running scripts is disabled on this system" when `npx` runs, run this once and try again.

```
Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

The two Windows commands come from REA's own setup page, rea.tools/get-started. On a Mac, use the installer from nodejs.org.

The app

The app must be installed on your computer, and it must be an app you own or are authorized to examine. Step 2 finds its folder.

STEP 1

Install REA and connect it to Claude Code

This prompt is REA's own. It has Claude Code run REA's setup, show you the setup plan before anything changes, and check the result.

Paste into Claude Code

```
Install REA and connect it to this coding agent using npx rea-agents@latest setup. Show me the setup plan for approval, then verify the installation.
```

Setup writes two things:

- One entry named **rea** in Claude Code's configuration file. On Windows that file is **C:\Users\<you>\.claude.json**. On a Mac it is **~/.claude.json**. REA's docs say setup keeps your other entries and makes a backup of the file first.
- The REA skill, in your Claude Code skills folder. On Windows that folder is **C:\Users\<you>\.claude\skills**. On a Mac it is **~/.claude/skills**.

Your part: approve the commands Claude Code asks to run, read the setup plan, and approve it. If setup asks which agents to connect, choose Claude Code. Then exit Claude Code, make a new, empty folder for this work, and start Claude Code again inside that folder. REA's tools do not appear in a session that was already open.

Check that REA is connected. Type in Claude Code, and look for **rea** in the list.

```
/mcp
```

Or run setup yourself. Run in a terminal. Setup asks which agents to connect, shows the plan and waits for your approval.

```
npx rea-agents@latest setup
```

Mac, untested: setup can also install Hopper, a commercial decompiler, after you approve it. REA's README says you must legally obtain a commercial tool and be authorized to use it.

STEP 2

Find the app's install folder

This prompt asks Claude Code to look up one installed app, report its name, its version and its folder, and change nothing.

Windows

For a Microsoft Store app, the folder is **C:\Program Files\WindowsApps** followed by the package's full name. On Jason's PC both lookup commands ran without administrator rights.

Paste into Claude Code, with one word from the app's name swapped in

Find where one app is installed on this PC. Change nothing.

One word from the app's name: <one word, for example AirPlay>

1. Run this command in PowerShell. Put my word where AirPlay is. Show me the output:

```
Get-AppxPackage *AirPlay* | Select-Object Name, Version, InstallLocation
```

2. That command matches package names, and a package name can differ from the name on the Start menu. If the output is empty or shows a different app, run this and show me the output:

```
Get-StartApps | Where-Object Name -like "*AirPlay"
```

The part of AppID before the exclamation mark is the package family name. Then run:

```
Get-AppxPackage | Where-Object PackageFamilyName -eq "" | Select-Object Name, Version, InstallLocation
```

3. If both lookups are empty, say so. Then look for an app that did not come from the Microsoft Store, and tell me how you found it.

4. Tell me three things: the app's name, its version, and the full path of its install folder.

5. List the files at the top level of that folder, with their sizes.

Do not copy, move, change or delete any file. Do not install anything.

Mac (untested)

REA's open_binary tool accepts a macOS app bundle, so the next prompt needs the bundle path and nothing more.

Paste into Claude Code, with one word from the app's name swapped in

Find where one app is installed on this Mac. Change nothing.

One word from the app's name: <one word from the app's name>

1. Find the app bundle whose name contains my word. Tell me how you found it.

2. Tell me three things: the app's name, its version, and the full path of its bundle.

3. List the files and folders at the top level of the bundle, with their sizes.

Do not copy, move, change or delete any file. Do not install anything.

Your part: swap in one word from your app's name, then copy the name, the version and the path from the answer. You paste all three into the next prompt. Jason's run covered one Microsoft Store app, and other kinds of app are untested.

STEP 3

Get the report

This prompt tells Claude Code to study the app with REA first, say which decompilers REA can use on your computer, and write its findings to **REPORT.md** with the evidence for each one. It builds nothing and leaves the installed app alone.

If REA has no decompiler, Claude Code falls back to three things it can read from the app's files without one:

- **Imports:** the functions a file takes from other files.
- **Exports:** the functions a file offers to other files.
- **Strings:** the readable text inside a file, such as messages and names.

Paste into Claude Code, with the three blanks filled in

Use REA to study one app that is installed on this computer. Write a report on how the app is built. This is a read-only study: build no program. You may write small helper scripts in the current folder to read the app's files.

THE APP

- Name: <app name>
 - Version: <version>
 - Install folder: <full path of the install folder or app bundle>
- I own this copy and I am allowed to examine it.

WHERE TO WORK

- Work only in the current folder. Put every file you write here.
- Never write into the app's install folder. Never change, move or delete the installed app.
- Read the app's files where they are. If a tool needs its own copy of a file, put the copy in a subfolder named evidence and tell me. That subfolder stays private.
- Install nothing. If a tool you want is missing, say so and use what is already on this computer.

USE REA FIRST

1. Check that you have REA's tools. Their names start with mcp__rea__. If you have none, stop and tell me that REA is not connected to this session.
2. Call REA's binary_session tool. Tell me which decompilers REA can use on this computer. REA calls them providers: Hopper, Ghidra and IDA. Say plainly whether REA can decompile native code here.
3. Open individual files with REA. Its open_binary tool accepts a file or a macOS app bundle and rejects any other folder.
4. Use the REA tools that fit these file types. You do not need to try every tool.
5. Record each open question with REA, then export REA's evidence bundle into the current folder.

THEN THE FALLBACK

If REA cannot decompile native code here, keep going. For each main file, list its imports, its exports and its readable strings with ordinary tools. Label every finding with its source: "REA" or "ordinary tools". Only REA output counts as REA evidence.

WRITE REPORT.md WITH THESE SECTIONS

1. What REA could do on this computer, and what it could not see.
2. The files: each main file, its size and its job.
3. What the app is built from: languages, frameworks and libraries.
4. Open-source projects the app ships or is built on. For each one: its name, its license, and the strings or names that point to it.

5. How the main feature works, step by step, as far as the evidence goes.
6. License checks, accounts and servers: say whether the app has each one. Report what you see and bypass nothing.
7. What a rebuild of the main feature would need: the parts that exist as open source, the parts that are missing, and the build tools.
8. What could not be determined. For each item, say what would settle it.

RULES FOR EVERY FINDING

- Mark it "observed" or "inferred".
- Give the evidence: the file name, plus the string, import, export or REA record behind it.

End your reply with a five-line summary and the full path of REPORT.md.

Your part: fill in the name, the version and the install folder from step 2, then approve what Claude Code asks to run. Expect REA to report no decompiler on Windows. That is the default there, and it was the state of Jason's PC: REA read two metadata files, and every other finding came from imports, exports and strings.

Mac, untested: REA's docs list more for a Mac than for Windows. They list Hopper, Ghidra support without the experimental label, and an open_binary tool that accepts an app bundle.

STEP 4

Read the report and decide

Read **REPORT.md**. Start with three sections: what REA could and could not see, the open-source projects the app is built on, and what could not be determined. The report is the outcome of this guide. Steps 5 and 6 are optional.

The report says	What it means	What to do
An open-source project sits behind the main feature	The hard part exists as source code. Jason's app was this case: its engine was a port of UxPlay, an open-source AirPlay server.	Go on to step 5.
The main feature depends on a license check, an account or a server	Jason's app depended on none of the three, so this case is untested. The build prompt tells Claude Code to stop at a license check or a sign-in.	Stop at the report.
Little or nothing about how the main feature works	A thin report. The app's core is proprietary and REA had no decompiler, so there is no source to build from and no code to read.	Stop at the report, or add Ghidra (see the fixes) and run step 3 again.

Your part: decide whether to go on. One more thing made Jason's app a simple case: his PC already had a C++ compiler.

STEP 5, OPTIONAL

Build your own version

This prompt has Claude Code build your own version of one feature as a new program, from new code and open-source parts, in a new subfolder. Claude Code checks for a compiler before it starts, installs nothing system-wide, runs its own tests, and ends with a report on what it verified and what it did not.

Paste into Claude Code, with the four blanks filled in

Build my own version of one feature of the app described in REPORT.md. Read REPORT.md in the current folder first.

THE TARGET

- Original app: <app name>, version <version>
 - Install folder: <full path of the install folder or app bundle>
 - The feature I want: <the main feature, in one sentence>
- I own this copy of the original app and I am allowed to examine it.

WHAT "MY VERSION" MEANS

- A new program that you build from source code. It runs on its own. It does not repack, load or depend on the original app's files.
- Put nothing from the original app in the result: none of its binaries, none of its artwork and none of its text.
- If REPORT.md shows that the original app is built on an open-source project, get that project from its own source repository and build it at a pinned commit. Write the commit in the README.
- Do not bypass or remove a license check, a sign-in or copy protection. If the feature depends on one, stop and tell me.

WHERE TO WORK

- Make a new subfolder named my-version in the current folder. Work only there.
- Never write into the original app's install folder. Never change the installed app.
- Use REA first when you need to look at the original app again. Open individual files: REA's open_binary tool accepts a file or a macOS app bundle and rejects any other folder.

BEFORE YOU WRITE CODE

1. Check that a compiler for this kind of program is installed. If it is missing, stop. Tell me what is missing and how to install it. Do not install it yourself. Build tools that can be downloaded into the my-version folder, such as CMake or Ninja, count as dependencies.
2. Install nothing system-wide. Download dependencies into the my-version folder only, and tell me how much disk space they use.
3. Tell me your plan in a few lines: what you will take from open source, what you will write, and what you will leave out.

WHILE YOU BUILD

- Build the feature I named first. Leave the other features out and list them.
- Each time you make a judgment call for me, add one line to ASSUMPTIONS.md: what you assumed, why, and what changes if you are wrong.
- When something fails, test one cause at a time. After three failed attempts at the same fault, stop and show me what you tried.
- Do not publish, push or upload anything. Ask me before any other action that sends something off this computer.

WHAT "DONE" MEANS

- Scripts that fetch the dependencies, build the program and package it, starting from an empty folder.

- A program folder that runs on its own on this computer.
- A log file that the program writes, and the path to it.
- A README: how to build the program, how to run it, and the license of every part.
- Self-tests that run without the real device or account. Tell me exactly which tests passed.

YOUR END REPORT, IN THIS ORDER

1. What you built.
2. How you built it.
3. What you verified, and how.
4. What you did not verify. Name the one test that only I can do, and give me the steps.
5. Guesses: each value or behavior you could not read from the original app.
6. What you left out.
7. The license of everything the program links, and the license of the result.
8. Firewall state: each rule or prompt the program needs.
9. Side effects on my desktop during your tests: windows, sounds, anything left running.
10. Disk space used.
11. What REA contributed, and what it could not do.

Your part: name the one feature you want, then watch the run. Jason used auto mode, and Claude Code asked him nothing across 193 tool calls. In the default permission mode, Claude Code asks you to approve most tool calls.

Each line of the prompt comes from a point where Jason's run needed a guess or a recovery. One example: his first build reused three FFmpeg library files that shipped with the original app. FFmpeg is open source, and a second build compiled it from source to replace them. The prompt rules that out from the start.

STEP 6, OPTIONAL

Test it yourself, with the real device

Claude Code's self-tests run without the real device or account. So the end report names one test that only you can do. For Jason's app that test needed an iPhone: start the program, open Control Center on the phone, choose Screen Mirroring, and pick the PC's name.

Your part: run that test, then check every item the end report lists as not verified. If Windows asks whether to allow the program on your network, allow it on a private network, or the device cannot find the PC. If the test fails, tell Claude Code what you saw and point it at the program's log.

In Jason's run the log shows video from his iPhone 53 minutes after the prompt. Sound from the phone was never tested. That test ran on his first build, and no device has connected to the second build. That test ran on his first build, and no device has connected to the second build.

What Jason's run showed

One run, on 12 October 2026. These figures come from the session's own records and from the log of the program it built.

Three things the reel leaves out

- A free, open-source AirPlay receiver already existed. It is named UxPlay. Jason's version is that engine inside a new Windows program.
- The 47 minute build carried three FFmpeg files copied from the original app's package. A later build compiled FFmpeg from source and holds no code, binaries, artwork or text from the original app. No device has connected to that later build.
- The result is GPL-3.0, because it links UxPlay. Anyone who hands the program to someone else owes them the source.

Measure	What happened
The prompt	One line: use REA to rebuild the app in one install folder as a standalone app. It had a typo, and the folder path was pasted twice. Claude Code had to decide what "rebuild" meant.
The setup	Claude Code v2.1.296 with Opus 5.5 at xhigh effort, in auto mode, and REA 6.4.0. REA was first set up on that PC three days earlier. A C++ compiler, git and Python were already there.
Time	47 minutes 2 seconds from the prompt to a built program. At that moment no iPhone had connected. The first iPhone connection came 53 minutes after the prompt.
Tool calls	193. REA made 8 of them. Ten calls returned an error, and Claude Code recovered from each one without Jason.
What REA did	It reported that it had no decompiler: Ghidra and IDA were not set up, and Hopper is not supported on Windows. It read two metadata files, logged two open questions and exported an evidence bundle.
The rest of the study	Claude Code listed imports and exports with dumpbin, and it wrote a small script to pull strings.
What the app is	A Microsoft Store app. Messages in its protocol engine appear word for word in the source of UxPlay, so Claude Code inferred that the engine is a port of UxPlay. It found no license check, and it bypassed none.
What it built	A new Windows program. Claude Code compiled UxPlay from unmodified source at a pinned commit, and wrote new code for video, sound and the window.

Proven	Video from an iPhone. The program's log shows two sessions, 29 seconds and 90 seconds long, with H.264 video at 498 x 1080.
Not proven	Sound from the phone. The phone never opened an audio stream. H.265 video from a device and recovery after a dropped connection are also untested.
Left out	Recording to MP4, casting of video links, the now-playing panel, the on-screen stats overlay, translations, the Store extras and Xbox support.
Guesses	Two defaults: 60 frames per second, with VSync on. Claude Code could not read the original app's defaults.
Judgment calls	Eight. Claude Code put none of them to Jason as a question. It logged each one with the assumption tool from Terminal Desk . The build prompt asks for the same list in ASSUMPTIONS.md.
Problems	Eleven. Among them: the doubled path in the prompt, a Visual Studio install that CMake could not find, and a crash on first start. Claude Code got past each one.
Tokens	34.26 million in the 47 minutes. 98% of them were cache reads.
Disk	About 3 GB after the second build.
Published	Nothing. No repository, no upload and no signing.

When something goes wrong

What you see	What it means	The fix
No rea in /mcp	The session was open before setup ran, or your Node.js is a version REA does not run on.	Exit Claude Code and start it again. Run <code>node -v</code> . REA runs on Node.js 22.19 or later, 24.11 or later, or 26 or later, and does not run on 23 or 25.
REA reports no decompiler	The default state on Windows. REA's message reads: "Set GHIDRA_INSTALL_DIR to an extracted Ghidra 12.1.x release directory." Hopper shows as not supported on Windows.	Carry on. The report prompt falls back to imports, exports and strings. To read native code, see the Ghidra note under this table.
REA's doctor says environment_healthy: false	Ghidra and IDA are not set up. Jason's setup showed the same. REA's docs say doctor "checks files and prerequisites, not the live agent connection."	Type <code>/mcp</code> . If rea is in the list, go on.
"This target is a directory"	REA's open_binary tool accepts a file or a macOS app bundle. It rejects a Windows install folder.	The prompts tell Claude Code to open individual files. If the message still appears, tell Claude Code to open the files one at a time.
The report is thin	The app's core is proprietary and REA had no decompiler. There is no source to build from and no code to read.	Stop at the report. Or add Ghidra, then run the report prompt again in a new folder.
The build stops: no compiler	The build prompt checks first and stops. Claude Code did not install a compiler in Jason's run, and REA never installs one.	For a C++ app on Windows, install Visual Studio Build Tools 2019 or later with the C++ workload. The installer needs administrator rights and several GB. Then paste the build prompt again.
CMake: "could not find any instance of Visual Studio"	Visual Studio's own locator reports the Build Tools install as incomplete. Jason's PC did this.	Tell Claude Code what worked on Jason's PC: load <code>vcvars64.bat</code> into the shell, then use the Ninja generator with cl as the compiler.
The first build takes a long time	The build downloads open-source projects from GitHub and compiles them. In Jason's second build, FFmpeg alone took 5.6 minutes.	Let it run. If a download fails, check your internet connection.

The program crashes on first start	Jason's first full build did this: an access violation at start. Claude Code's first idea of the cause was wrong, and its second was right.	Give Claude Code the error text. The build prompt tells it to test one cause at a time and to stop after three failed attempts.
The device cannot find the PC	Windows Firewall blocks the program, the network is set to Public, or the device is on another network. Firewall rules follow the file path, so a copy of the program in another folder asks again.	Allow the program on a private network. Put the device and the PC on the same network.
An approval prompt on almost every tool call	The default permission mode. Jason's run made 193 tool calls in auto mode, and Claude Code asked him nothing.	Approve the calls as they come. Anthropic's page explains the modes: code.claude.com/docs/en/permission-modes
A usage limit	The build is a large job. Jason's run used 34.26 million tokens in 47 minutes, and 98% of them were cache reads.	Jason did not hit a limit, so no fix was tested. The files Claude Code already wrote stay in the working folder.
"Windows protected your PC"	SmartScreen shows this for an unsigned program that was downloaded. Jason did not see it on his own PC.	It applies only when you share the program. Read the next row first.
Before you share your version	A version that links a GPL-3.0 project is GPL-3.0, and handing it to someone means giving them the source. A name can matter too: "AirPlay" is an Apple trademark. REA's disclaimer says an analysis report "does not grant rights to the original program".	Read the license list in the end report.

Optional: Ghidra on Windows

To let REA read native code on Windows, install Ghidra 12.1.x and a 64-bit JDK 21 or later yourself. Then set **GHIDRA_INSTALL_DIR** to the Ghidra folder. REA's docs mark this route experimental and read-only. They give Ghidra up to 330 seconds to start by default, so the first query can be slow. Jason's run did not use Ghidra, so this route is untested here. The steps are in REA's installation docs: github.com/morluto/rea/blob/main/docs/installation.md

To remove REA

Ask Claude Code to run REA's uninstall command, `npx rea-agents@latest uninstall`. REA's docs say it "removes only REA-owned agent registrations and skill files". Your REPORT.md and your version stay in the working folder.

KEEP GOING

The guide page

Every step and prompt in this PDF is on the page too, with a Windows track and a Mac track:

graniteai.co/tools/reverse-engineer-anything

See what GraniteAI makes: graniteai.co/tools/reverse-engineer-anything#for-sale