// DEF CON · Workshop

# Hacking IDE Extensions

A hands-on intro to finding bugs in VS Code extensions.

-- Nick Copi

// 00 / Setup

# Before we begin.

## You need

- VS Code 1.95+ — or any fork that loads `.vsix` extensions (Cursor, VSCodium, Windsurf)
- Chrome (or any Chromium browser) — for the DevTools we'll attach
- A few minutes to install the lab extension
- Willingness to break things on purpose

## Grab the lab

turb0.one/files/nopilot.vsix

Grab it now while the wifi holds. If it dies, flag me — we'll pass it around locally.

## Install in one line

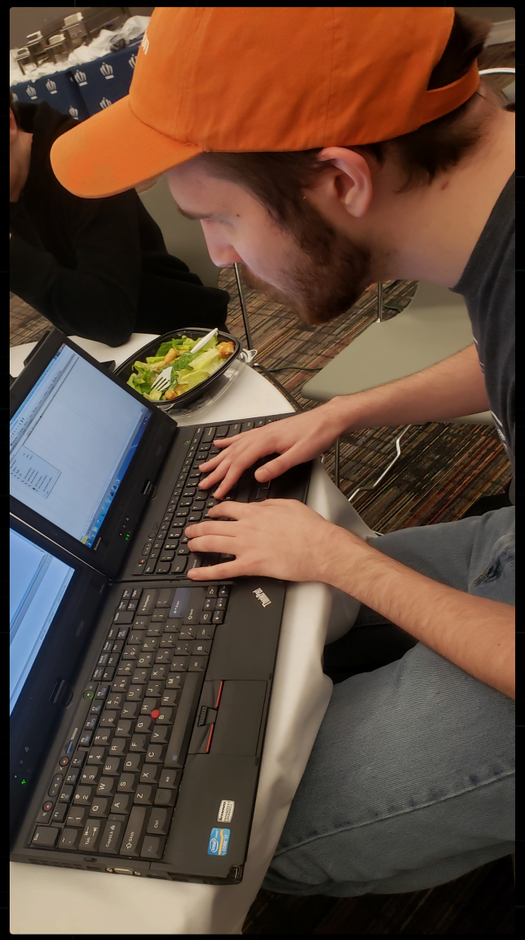
```
# from the terminal
code --install-extension nopilot.vsix

# or: Extensions view (Ctrl+Shift+X)
# → "..." menu → Install from VSIX…
```

Verify: open any folder, the Nopilot icon appears in the activity bar and the chat focuses. Type `hello` — the mock assistant answers.

> The whole workshop runs locally against this one extension. No internet needed, nothing to get in trouble for.

// $ whoami

# Nick Copi



- Full Time Bug Bounty Hunter (recovering AppSec engineer)
- Richmond, Virginia — born and raised
- God loving American patriot
- Certified Research Reviewer (CRR)
- Thinkpad Dual Wielder
- nickcopi.com · @7urb01 on X

More hours spent debugging JavaScript in the last year than in direct sunlight.

// 00 / Plan

# What we're doing today.

## The arc

- **Understand** — what an extension actually is, and why it's a juicy target
- **Dissect** — anatomy, the two execution contexts, what a `.vsix` really is
- **Tool up** — install, manage, and *debug* extensions like a researcher
- **Hack** — turn the tooling loose on a deliberately broken extension

## Format

Short talk, then you're at the keyboard. The slides exist so you can follow along and refer back.

If you fall behind on setup, flag me — don't suffer in silence. The interesting part is the hacking, not the install.

> Goal: by the end you can pull any extension apart and know where to look first.

// 01 / Framing

# What *is* an IDE extension?

## It depends.

In VS Code it's usually a combination of two things running together:

- extension-host code — a privileged Node.js process with filesystem and process access
- client-side code — webview HTML/JS rendering UI inside the editor

The value of any given extension as a target depends entirely on its implementation. Context is everything.

### Why you should care

Node host ↔ postMessage ↔ webview

- Great attack surface — two worlds talking across a bridge
- The host has real privilege; the webview renders untrusted content
- A ton of AI IDEs and assistants are built as exactly this

> Copilot, Cursor, Continue, Cline — coding assistants ship as extensions, render model output in webviews, and run commands.

// 02 / Thesis

# Why does this matter for bug bounty?

### It depends

Context is key. The value of an extension depends on its implementation — but the upside, when it's there, is big.

### Good bugs

High-quality, impactful vulnerabilities that are genuinely rewarding to discover. Real chains, not low-hanging info leaks.

### Pays well & fun

Lucrative bounties combined with the thrill of technical problem-solving. The best kind of work.

### Rapid feature shipping

These platforms ship features fast — especially the AI ones — creating constant new attack surface to mine.

### Complex integration

Custom features wired into complex platforms widen the scope for technical exploits considerably.

### Privileged victims

An extension runs as the developer. Developers hold SCM tokens, cloud creds, and session cookies. One compromise cascades.

> Mature web-app bug classes, freshly ported into editors that few specialized hunters are looking at.

// Part 1

# How extensions are built.

You can't hack what you can't read. Let's take one apart.

// 03 / Structure

# How a VS Code extension is structured.

## The manifest: `package.json`

- `main` — entry point into the host code, e.g. `./out/extension.js`
- `activationEvents` — *when* the code is loaded and run
- `contributes` — commands, menus, views, settings, languages
- `engines.vscode` — minimum API version targeted
- `capabilities` — workspace-trust / untrusted-workspace behavior

## The lifecycle

- `activate(context)` — runs on an activation event; registers everything
- `deactivate()` — cleanup; often empty
- `context.subscriptions` — disposables tidied up on unload

```
// package.json
{
  "name": "nopilot",
  "publisher": "turb0",
  "main": "./out/extension.js",
  "activationEvents": ["onStartupFinished"],
  "contributes": {
    "commands": [{ "command": "nopilot.open" }]
  }
}

// out/extension.js
const vscode = require('vscode');
function activate(context) {
  // your attack surface starts here
}
module.exports = { activate, deactivate };
```

> Read the manifest first — it tells you what the extension can do and when its code starts running.

// 04 / Structure

# Two worlds, one bridge.

// webview

Sandboxed iframe. Renders HTML/JS for custom UI. No Node, no `vscode.*`.

`acquireVsCodeApi()` — once per session — gives it `postMessage` and nothing else.

↔ postMessage

// the bridge

`webview.postMessage()` down, `onDidReceiveMessage()` up.

Untrusted data crosses here. This is where chains are born.

↔

// extension host

A Node.js process. Full `fs`, `child_process`, network, and the entire `vscode` API.

Runs with the user's privileges. No sandbox.

## The mental model

The webview is the browser; the extension host is the server. Treat messages from the webview like requests from the internet — because once there's XSS in the webview, that's exactly what they are.

## The classic chain

DOM XSS in webview → postMessage → host handler → `executeCommand` / `spawn` / `openExternal`

If the host trusts what the webview sends, the sandbox is decorative.

// 05 / Structure

# A `.vsix` is just a ZIP.

The whole extension ships in one file. Unzip it and read everything — there's no DRM, no obfuscation you can't undo.

```
$ unzip nopilot.vsix -d nopilot/
$ ls nopilot/
  extension/            # the actual code
  extension.vsixmanifest
  [Content_Types].xml

$ ls nopilot/extension/
  package.json          # read this first
  out/extension.js      # host entry
  media/                # webview html/js
```

## Researcher first moves

- Read `package.json` → `main`, `activationEvents`, `contributes`
- Open the host entry — usually bundled/minified by esbuild or webpack; beautify it
- Grep for sinks:

```
$ grep -rnE \
  'child_process|exec|spawn|sendText|\
createTerminal|openExternal|eval|\
executeCommand|innerHTML|postMessage' \
  nopilot/extension/
```

Source maps, if shipped, hand you the original TypeScript.

> Every extension you install is sitting in `~/.vscode/extensions` right now, fully readable.

// 06 / Structure

# Activation = code execution.

An extension's code runs the moment an activation event fires. Often that's just opening a folder — no click required.

## Common triggers

- `onStartupFinished` — every launch fires it
- `workspaceContains:**/glob` — a matching file exists in the folder
- `onLanguage:python` — a file of that type is opened
- `onCommand:*` — command invoked (auto since VS Code 1.74)
- `onUri:*` — a deeplink handler fires
- `onView:*` — a side-panel view becomes visible

## Why an attacker cares

- The attacker controls what files are in a repo
- So the attacker picks which extensions activate
- Open a hostile repo → extension code runs with your privileges
- Extension code is trusted by default; the workspace isn't — the prompt gates the wrong boundary

> The set of installed extensions *is* the effective attack surface, whether the user thinks of it that way or not.

// 07 / Structure

# Workspace trust is the *whole* game.

When you open an untrusted folder, VS Code enters Restricted Mode. Extensions declare how they behave there via `capabilities.untrustedWorkspaces`: `true`, `false`, or `'limited'`. This one field decides how good your bug is.

## Why this is the lever for bug bounty

- VS Code's own model says: untrusted workspace = attacker-controlled, must not run code
- A bug that fires without trust = "just open this repo" → exploitation, zero clicks
- That violates the vendor's *stated* security boundary — triage can't wave it away
- A bug that needs trust granted first is far weaker: the user already opted in
- Most IDE / AI-assistant programs explicitly scope pretrust execution — it's a named target, not a gray area

## Where the gold is

- Extensions that declare `untrustedWorkspaces: true` for convenience — they run fully in Restricted Mode
- Activation events (`workspaceContains`, `onLanguage`) that fire before any trust check
- Sinks reachable in `'limited'` mode that the dev assumed were gated

First thing to grep in the manifest: `untrustedWorkspaces`. It tells you how much you get for free.

> Code exec, sensitive disclosure, or filesystem read pretrust = a security boundary clearly bypassed. Trust state *is* your evidence.

// Part 2

# Your tooling.

Install it, manage it, and — the part that matters — debug it.

// 08 / Tooling

# How to install extensions.

## Three ways in

- **Marketplace** — Extensions view, `Ctrl+Shift+X`, search & install
- **CLI** — `code --install-extension <id|path.vsix>`
- **From VSIX** — Extensions view → `"..."` → *Install from VSIX…*

Sideloading from a `.vsix` skips the marketplace entirely — exactly what we want for a target you've repackaged or a lab build.

```
# install the lab
code --install-extension nopilot.vsix

# same flag on the forks
cursor --install-extension nopilot.vsix
```

For today: download the `.vsix`, then either the CLI above or the Command Palette / UI — whatever's fastest. Both work.

> No `code` command? In VS Code: `Cmd/Ctrl+Shift+P` → "Shell Command: Install 'code' command in PATH".

// 09 / Tooling

# How to manage extensions.

## Day-to-day

- Enable / Disable — globally, or *Disable (Workspace)* for just this folder
- Uninstall — gear menu, or `code --uninstall-extension <id>`
- List — `code --list-extensions --show-versions`
- Profiles — named extension/setting sets you switch between

For research: isolate your target in its own profile so unrelated extensions don't pollute what you're observing.

## Profiles = your research sandbox

- `code --profile "lab"` — spin up a clean window with only the target installed
- Nothing else running means nothing else to confuse what you're observing
- Throw it away when you're done; your daily setup stays untouched

Run the lab in its own profile so the only behavior you're watching is Nopilot's.

> Disable everything you're not testing. A noisy editor hides the one behavior you care about.

// 10 / Tooling

# How to debug an extension.

Forget `F5` and `launch.json` — that's for an extension *you* wrote. As a researcher you want to debug the already-installed one. Launch the editor with an inspector on the extension host:

```
# expose the host's Node inspector
code --inspect-extensions=9229

# same for the forks
cursor --inspect-extensions=9229
codium --inspect-extensions=9229
```

## Then attach

- Chrome → `chrome://inspect` → add `localhost:9229` → *inspect*
- or a Node "attach" config pointed at port `9229`
- full Chromium DevTools on the live host process

## Now you own the host

- Breakpoint into any installed extension's real code
- Sources tab shows the on-disk JS from `~/.vscode/extensions`
- Pause on the message handler; inspect what crossed the bridge
- Evaluate live in the console — call its functions, poke `vscode.*`
- Edit the on-disk JS, reload the window, watch your change land

No source needed, no project to open. The host is running every extension you have installed — and you're attached to it.

> This debugs production extensions as they actually run — exactly the thing you're trying to break.

// 11 / Tooling

# Webviews are just *iframes*.

A webview isn't a separate process or window — it's an iframe embedded in the workbench renderer. So you debug it with the workbench's own DevTools; there's no special webview inspector.

## Get to the right context

- `Developer: Toggle Developer Tools` — opens Chromium DevTools for the workbench window
- The webview content runs in a nested iframe served from `vscode-webview://`
- In the Console, switch the JS context dropdown to that frame; in Elements, find the iframe
- `Open Webview Developer Tools` lands you in the *same* DevTools — it's a shortcut, not a second inspector

## Once you're in the frame

- Inspect the DOM; set breakpoints in the webview's scripts
- Watch `postMessage` traffic crossing the bridge
- Craft and fire DOM XSS / CSP-bypass payloads right in the console

Gotcha: `acquireVsCodeApi()` runs once per page session. Calling it again throws — reuse the reference the page already grabbed.

> Host inspector ( `--inspect-extensions` ) + workbench DevTools = full visibility across the bridge.

// Part 3

# Where the bugs live.

Now that you can read and drive an extension — what are you actually looking for?

// 12 / Hacking

# Sources, sinks, and the bridge.

## Sources — attacker-controlled input

- file contents, file *names*, and paths in the workspace
- messages arriving from the webview
- `vscode://` URI deeplinks
- settings & config files in the repo
- instructions an AI agent reads from workspace files — indirect prompt injection

## Sinks — where it gets dangerous

- `child_process.exec / spawn` — fires pretrust
- `innerHTML` in the webview → XSS
- `terminal.sendText` / Tasks — often gated by trust
- `commands.executeCommand(...)`
- `env.openExternal(uri)` → OS default handler

Some sinks are blocked in Restricted Mode — check which actually fire pretrust. `child_process` usually does.

## The recurring footguns

- shell from workspace strings — `cd $path && ...`, branch/file names concatenated into commands
- glob "allowlists" — `ls *` matches `ls; evil` after expansion
- webview → command — messages routed straight into `executeCommand` as method names
- broad URI handlers — `open?url=` accepting any scheme
- unsanitized HTML — file paths interpolated into webview templates, blowing past the CSP `<meta>`

Every one of these has shipped in real extensions.

> AI is the accelerator. An agent reads attacker-controlled files and emits tool calls that bridge straight into those sinks. The confused deputy acts for whoever wrote the file.

> Find a source, find a sink, connect them across the bridge. That's a chain — and that's the lab.

// Part 4

# The lab.

Hands on keyboards. Let's break something.

// 13 / Lab

# Meet Nopilot.

A plausible-looking mock AI pair-programmer — every surface is a vulnerable design choice lifted from real shipping AI assistant extensions, variable names changed.

- Chat sidebar that renders markdown
- A custom `.nopilot` "skills" format — named recipes with *steps* and *triggers*, plus slash commands
- A context loader that auto-reads workspace files into the assistant
- An `autorun` toggle that gates whether tool calls execute

The "AI" is a regex echo bot — but the tool calls (shell exec, file read, file write) are real. It activates pretrust and reads its own gating from workspace settings.

> turb0.one/files/nopilot.vsix

## Your job

1. Install it; read the manifest cold — what activates, what's contributed
2. Unzip the `.vsix` (or read the source) and map the surfaces
3. Pop the chat webview's DevTools; learn the `postMessage` protocol
4. Attach to the host; breakpoint where messages and files get handled
5. Find a source, reach a sink, build the chain

Goal line: "victim opens a folder" → code execution, no further clicks. PoC or GTFO.

> Working session starts now — go hack. Challenges scale from "one primitive" to "full zero-click chain"; more than one bug lives in here. I'll be circulating.

// 14 / Regroup

# Alright — back together.

How'd it go? Let's walk a few of these out loud.

## If we've got time

- Live walkthroughs of what people found
- I'll walk a couple of the simpler chains end to end
- From "opened a folder" to a primitive, then to code execution

### Compare notes

Different people hit different surfaces — chat, skills, context loader, the custom editor. Pool what you saw.

Got a clean PoC? Show it. That's the whole point of the room.

> >_ switching to the editor…

// 15 / Fin

# That's a wrap.

Unzip it, read the manifest, debug both sides, trace a source to a sink.

Same recipe on every extension you'll ever look at — including the AI ones shipping new surface every week.

Now go find the good bugs. PoC or GTFO.

## Thanks

Nick Copi

nickcopi.com · @7urb01

questions welcome.