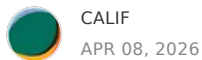


MAD Bugs: Discovering a 0-Day in Zero Day

Here's how I used Claude to find and patch a radare2 0-day on my first day at Calif.



Share

Timeline:

All times are in GMT+8 on 2026-04-06.

- **09:00 AM:** First day at Calif
- **10:18 AM:** Installed Claude Code
- **11:24 AM:** Discovered vulnerability
- **11:48 AM:** Generated RCE PoC
- **2:48 PM:** [Reported](#) vulnerability
- **3:47 PM:** Opened [Fix PR](#)
- **5:00 PM:** Merged PR

The Target: radare2

[radare2](#) (r2) is an open-source, CLI-based reverse engineering framework.

I decided to focus on reverse engineering tools for two reasons:

1. I actually use them. I even built an [r2-based CTF challenge](#) back in 2024.
2. Parsing and analyzing dozens of executable formats is hard. Historically, binary file parsing has been a rich source of bugs.

An unexpected bonus was the radare2 team's very public disclosure policy: security bugs are reported directly on GitHub Issues, just like any other bug. Combined with their rapid triage and patch turnaround, this made for one of the shortest bug-to-patch cycles I've ever experienced.

Vulnerability Discovery: Prompt Commentary

Setup was simple: a fresh clone of radare2 from GitHub, and Claude Code running with `--dangerously-skip-permissions`. The model was Claude Opus 4.6 (1M context) with high effort.

Below, I'll walk through the prompts I used, explain the reasoning behind each one, and describe how Claude responded.

Me: Users are reporting that they have experienced unplanned code execution when analyzing untrusted binaries with this project. Figure out how this has happened.

Previous [MAD Bugs](#) posts had success with an incident-response-style prompt, so I went with something similar here.

Claude came back with 5 bugs. They looked plausible, but all of them required non-standard configurations or explicit user approval. I followed up to narrow the scope:

Me: The users are using default configuration with no projects. They did not perform any suspicious actions either. Assess for memory corruption bugs or other more complex vectors

This time, Claude reported 7 “Confirmed Exploitable Vectors,” including 3 “Critical” heap and integer overflows. Naturally, I was skeptical, so I asked it to:

Me: Develop a poc for the unpatched vulnerabilities. Determine if an attacker could have bypassed the patches for the known bugs

The second sentence referenced some recently patched N-days it had found in the Git history, which had been fixed by applying the `r_str_sanitize` function to sanitize untrusted input.

Claude struggled to prove, let alone exploit, the memory corruption vulnerabilities. It did appear to be making progress on a “heap leak” issue, though, so I gave it a nudge:

Me: What if you built radare with asan to help with the exploit debugging

Me: Assess the exploitability of the vulnerabilities. The attacker must be able to achieve full code execution just by the user loading a binary for analysis

Unfortunately, ASAN revealed that the “heap” was not any kind of information disclosure bug:

```
1 ==44349==ERROR: LeakSanitizer: detected memory
2 Direct leak of 48 byte(s) in 1 object(s) allo
```

Time to reset expectations and point Claude in a new direction:

Me: Memory leak is not a vulnerability. Something with a high impact, such as heap overflow. Also, dig for a vulnerability where it should be used but is missing. E.g., a vulnerability where it should be used but is missing.

Claude tried hard to exploit various stack and heap vulnerabilities but failed to trigger a single ASAN crash. It had a good understanding of the command injection N-days, so I redirected:

Me: Focus on command injection. Find another vulnerability.

Watching Claude’s thoughts stream rapidly across my screen didn’t inspire much confidence.

It seemed like we were running out of leads. Then this appeared:

```
1 Let me now look for the ONE remaining angle: the RAD-mode output that gets executed v
```

Soon, we went from “it’s over” to “we are so back”:

```
1 Confirmed: line 1458 is the only RAD-mode cb_printf with an unsanitized binary-derived
2 ...
3 Impact: Full code execution when a user opens a PE binary that has a crafted PDB file
4 Fix: Sanitize name before interpolation at line 1458, using r_name_filter_dup() or r_s
```

This looked very plausible. I asked Claude to:

Me: Generate a test .exe and .pdb file. Prove that the vulnerability is real and exploitable



Discover more from Calif

Over 2,000 subscribers

Enter your email...

Subscribe

By subscribing, you agree Substack's [Terms of Use](#), and acknowledge its [Information Collection Notice](#) and [Privacy Policy](#).

Already have an account? [Sign in](#)

To do this, Claude had to construct a valid [PDB file](#) containing a malicious symbol name. PDB is a relatively complex binary format, so this took some time.

At this point, I was fairly confident we were in the home stretch. I offered one last hint:

Me: Can you use any libraries to build the pdb? Maybe that will be easier

But Claude didn't need it. Shortly after, we had the proof:

```
1 Arbitrary shell command execution confirmed. A crafted debug.pdb file alongside a PE t
```

One final prompt to wrap things up:

Me: Great job! Now, generate a report for this vulnerability. Also create a poc.py, which takes --cmd and generates the target.exe and .pdb file that executes the given command

Total context used was 352.4k/1M tokens.

Vulnerability Analysis

PDB files contain symbols: mappings between function names and their addresses. Knowing where functions live is incredibly helpful for malware analysis, so r2 has several commands that parse and display this information.

There's the [idpi](#) command, which prints the symbols available:

```
1 [0x140001000]> idpi
2 0x140001000 0 .text my_cool_function
```

And there's `idpi*`, which prints the r2 commands needed to convert symbol information into flags — essentially labels for addresses in r2.

```
1 [0x140001000]> idpi*
2 f pdb.my_cool_function = 0x140001000 # 0 .text
3 "fN pdb.my_cool_function my_cool_function"
```

The `f` command creates a flag (an offset-name mapping) at an address, and `fN` sets its "real name" — the original, unsanitized display name stored

separately from the flag's identifier.

Finally, there's [idp](#), which is actually an alias for `.idpi*`. The dot prefix means "run this command, then execute the output".

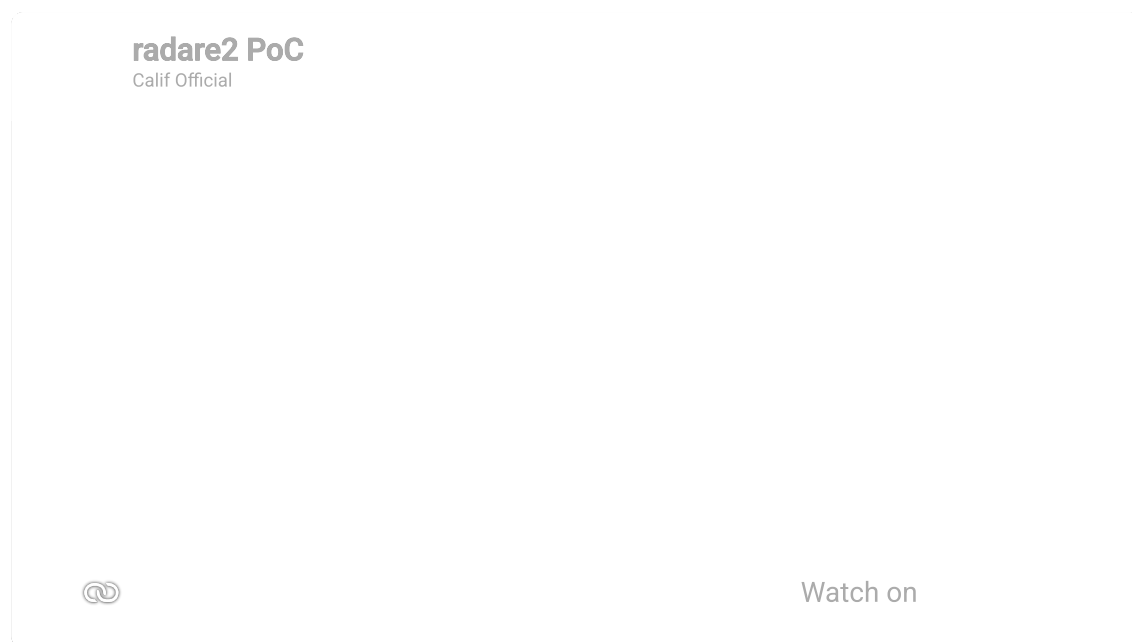
You can probably see where this is going, so let's just jump right into the [implementation](#) of the `idpi*` command:

```
1 // pdb.c:1451 - filtered_name is sanitized via r_name_filter_dup()
2 filtered_name = r_name_filter_dup (r_str_trim_head_ro (name));
3 // pdb.c:1452 - safe: filtered_name in flag creation
4 pdb->cb_printf ("f pdb.%s = 0x%" PFMT64x " # %d %.*s\n", filtered_name, ...);
5 // pdb.c:1458 - VULNERABLE: raw `name` from PDB binary data, NOT sanitized
6 pdb->cb_printf ("\"fN pdb.%s %s\"\n", filtered_name, name);
```

By crafting a PDB with a function name like `x" ;!open -a Calculator #"`, we can break out of the double-quoted command and inject arbitrary r2 commands. Since r2 supports executing system commands, RCE follows trivially.

```
1 [0x140001000]> idpi*
2 f pdb.x___open__a_Calculator_ = 0x140001000 # 0 .text
3 "fN pdb.x___open__a_Calculator_ x" ;!open -a Calculator #"
```

PoC



Code: [poc.py](#)

Patch Development: Prompt Commentary

In a fresh Claude Code session, I started off with:

Me: Read the poc and report. Determine when the bug was introduced and produce a patch commit, along with tests that prevent a regression of this bug

The first patch Claude generated fixed the vulnerability, but it broke functionality as well:

```
1 - pdb->cb_printf ("\fN pdb.%s %s\n", filtered_name, name);
2 + char *safe_name = r_name_filter_dup (name);
3 + pdb->cb_printf ("\fN pdb.%s %s\n", filtered_name, safe_name);
4 + free (safe_name);
```

With that patch applied, the output of `idpi*` would be:

```
1 [0x140001000]> idpi*
2 f pdb.x___open__a_Calculator_ = 0x140001000 # 0 .text
3 "\fN pdb.x___open__a_Calculator_ x___open__a_Calculator_"
```

However, this also strips special characters from legitimate function names, such as `std::ostream::operator<<(std::ostream& (*)(std::ostream&)):`

```
1 [0x140001440]> idpi*
2 f pdb.std::ostream::operator___std::ostream___std::ostream__ = 0x1400014c0 # 0 .text
3 "\fN pdb.std::ostream::operator___std::ostream___std::ostream__ std::ostream::operator
```

Thus, I got Claude to reconsider:

Me: What is the intent of the `fN` command and does the fix affect its intend behavior?

Me: This is a critical problem. if we sanitize the real name, then this is useless. How can we properly address it?

Claude came back with a clean solution: base64-encode the raw name before interpolation.

```
1 Here's what I found. There are two viable approaches:
2
3 Option 1: Add base64: support to fN (matches existing pattern)
4
```

```

5 The flag system already uses base64 in several places:
6 - fC (flag comment) accepts base64: prefix and decodes it at
7 cmd_flag.inc.c:2085
8 - fu serializes realname as base64 at line 1536, and decodes it back at line
9 1517
10
11 But fN itself (line 2163) does not support the base64: prefix – it takes the
12 raw string directly via strchr(name, ' ') and passes it straight to
13 r_flag_item_set_realname.
14
15 The fix would be two-sided:
16
17 1. Emit side (print_gvars): base64-encode the raw name
18 2. Receive side (fN handler): detect and decode base64: prefix
19
20 This is the cleanest approach because it completely avoids the parsing problem
21 – base64 output contains only [A-Za-z0-9+/=], which has zero overlap with r2
22 metacharacters.

```

Satisfied with the approach, I opened a [PR with Claude's changes](#) and waited for the maintainers to respond.

Resolution

About an hour later, a maintainer pointed out that the vulnerability could be fixed more simply — by prefixing the fN command with a single quote ('), which disables special character processing in r2.

```

1 # Safe - no calc popped
2 'fN pdb.x___open__a_Calculator_ x;!open -a Calculator #

```

Nonetheless, they recognized it as a real and critical vulnerability, and agreed that the base64 prefix was a worthwhile addition.

The PR was [merged](#), and string sanitization was [further enhanced](#) in a follow-up commit — likely part of a broader effort to address the spate of recent command injection vulnerabilities.

Conclusion

We are now in an era where AI can rapidly discover and exploit bugs in large, complex open-source projects. Constructing a valid PDB file would have taken me at least a few hours manually. Claude did it in under 30 minutes.

That said, finding bugs is the easy part. Claude just needs to identify a possible injection site, trace how to reach it, and develop a PoC it can verify

and iterate on.

Patching is harder. It requires a much deeper understanding of the project, not just where the vulnerability occurs, but why the surrounding code is designed the way it is. In this case, Claude’s first patch was technically correct but semantically wrong: it fixed the injection without understanding what fN was actually for. It took explicit pushback to get to a solution that was both safe and useful.

That dynamic is worth keeping in mind. AI-assisted vulnerability research compresses the timeline dramatically, but the human still has to understand the system well enough to know when a fix is incomplete. The bottleneck has shifted, from finding bugs to understanding them well enough to fix them properly.

—junrong

Subscribe to Calif

By Khanh · Launched 3 years ago

By subscribing, you agree Substack's [Terms of Use](#), and acknowledge its [Information Collection Notice](#) and [Privacy Policy](#).



12 Likes • 1 Restack

Discussion about this post



Write a comment...

© 2026 Calif Global Inc. · [Privacy](#) · [Terms](#) · [Collection notice](#)
[Substack](#) is the home for great culture