JavaScript Scope & Lifecycle Debugging Checklist

A triage checklist for var/let/const, hoisting, closures, and memory leaks — before you escalate.

1. var vs. let vs. const at a Glance

Keyword	Scope	Hoisting	Reassignable
var	Function	Yes — initialized undefined	Yes
let	Block	Yes — temporal dead zone	Yes
const	Block	Yes — temporal dead zone	No (binding only)

2. Before You File or Escalate a Defect, Check:

- ■ Is the value 'wrong on every row/callback'? Check for var inside a loop feeding async code (setTimeout, event handlers, API callbacks).
- ■ Is the console throwing 'Cannot access before initialization'? That's the temporal dead zone — confirm the variable is genuinely read before its let/const declaration line.
- ■ Is data bleeding between tabs, sessions, or components? Check whether the variable is module-level/global instead of scoped to the component instance.
- ■ Is a long-running dashboard or monitoring tab slowing down over hours? Open the memory profiler before assuming a data volume or server issue — look for growing listener/timer counts.
- ■ Was a var redeclared somewhere in the same file? Search for duplicate declarations — var allows silent redeclaration with no error.
- ■ Is const being relied on to prevent an object's contents from changing? const only locks the binding, not the object's internal state.

3. The Classic Loop-Closure Bug

Broken (var)

```
for (var i = 0; i < 3; i++) {
  setTimeout(() =>
    console.log(i), 100);
}
// Logs: 3, 3, 3
```

Fixed (let)

```
for (let i = 0; i < 3; i++) {
  setTimeout(() =>
    console.log(i), 100);
}
// Logs: 0, 1, 2
```

var is function-scoped — one shared variable for the whole loop. let creates a fresh binding per iteration.

Rule of thumb: if a defect looks like wrong data 'everywhere at once' from an async source (timers, callbacks, event handlers), suspect a shared var before you suspect the data source.
