Here's a practical, **enterprise-safe** sequence to reduce **1 moderate, 4 high, 15 critical** in a Next.js repo without randomly breaking the dependency graph.

The goal is to **(a)** identify what's actually runtime-reachable, **(b)** update direct deps first, **(c)** use overrides only when necessary, **(d)** avoid "fix everything" chaos.

---

0) Make a safe checkpoint

git checkout -b chore/security-audit
git status

---

1) Confirm your baseline + reproduce counts

Run these in order:

A) Full audit (everything)

npm audit

B) Runtime-only audit (what matters most)

npm audit --omit=dev

C) Only high+ (noise reduction)

npm audit --audit-level=high
npm audit --omit=dev --audit-level=high

Treat --omit=dev as your "production risk" view.

---

2) Identify the exact packages causing critical/high

A) Get machine-readable output (best for triage)

npm audit --json > audit.json

B) If you want quick text details for highs/criticals

npm audit --audit-level=high

For each critical/high item, note:

- package name
- which dependency chain pulls it in
- fix available? (yes/no)
- requires breaking change? (often shown as --force)

---

3) Fix the easy / safe ones first (non-breaking)

This applies patches/minor bumps where npm knows it won't break semver constraints:

npm audit fix

Re-check:

npm audit --omit=dev --audit-level=high

If your critical/high are still present (common), proceed.

---

4) Update your direct dependencies (most effective)

Most critical/high issues come from old transitive trees.

A) See what's outdated

npm outdated

B) Update direct deps conservatively

npm update

Then:

npm audit --omit=dev --audit-level=high

If still present, move to targeted fixing.

---

5) Targeted fixing: find *who* pulls the vulnerable package

When npm audit shows a vulnerable package (say glob, minimatch, serialize-javascript, etc.), do:

A) Explain the chain

npm explain <package-name>

Example:

npm explain glob

B) List versions installed

npm ls <package-name>

Now you can decide the correct remedy:

- **Update the top-level package** that pulls it in (best)
- **Override** the transitive version (when upstream isn't updated yet)

---

6) Use overrides surgically (don't shotgun it)

If the vulnerable package is **transitive** and the parent hasn't released a fix, add an override.

Example pattern in package.json:

```
{
  "overrides": {
    "glob": "^10.4.0",
    "minimatch": "^9.0.0"
  }
}
```

Then clean reinstall:

```
Remove-Item -Recurse -Force node_modules
Remove-Item -Force package-lock.json
npm install
npm audit --omit=dev --audit-level=high
```

Rules of thumb for overrides

- Only override packages that appear in critical/high output
- Prefer overrides that stay within semver expectations of parents
- After each override, run:
    - npm test (if you have it)
    - npm run lint
    - npm run build
    - npm run dev smoke test

---

7) Only use npm audit fix --force as a last resort

This can bump major versions and break Next/tooling.

If (and only if) you're stuck with criticals and the report says "fix available via --force":

```
npm audit fix --force
```

Immediately follow with:

```
npm run build
npm run lint
npm test
```

If it breaks, **revert** and go back to targeted overrides / direct dependency upgrades.

---

8) Separate dev-only criticals from runtime criticals

If you still see many criticals in full audit but runtime audit is clean-ish:

- Accept some dev-only issues temporarily
- Or bump dev tools (jest, ts-jest, msw, eslint plugins, etc.)

This is common: dev tooling trees pull in old packages.

---

9) When you think you're done: verify "real risk"

Run:

npm audit --omit=dev
npm run build

If runtime audit is clean (or only low/moderate) and build passes, you're in a good place.

Practical recommendation for your case

Given you had **~15 critical** after installing and you're on a corporate setup:

1. npm audit --omit=dev --audit-level=high
2. npm audit fix
3. npm update
4. For remaining critical/high:
    - npm explain <pkg>
    - update the direct parent if possible
    - otherwise add minimal overrides
5. Re-audit and run build

If you paste ONE thing, I can be very concrete

Paste the output of:

npm audit --omit=dev --audit-level=high

(or upload a screenshot). Then I'll tell you exactly:

- which ones to ignore (dev-only),
- which parent packages to upgrade,
- which overrides to add (and what versions),
- and the safest order to apply them.

Also is there any cmd to see which dependencies are actually not used ? We can remove such dependencies , would that help ?

Excellent question — this is exactly how mature repos reduce attack surface.

Yes, removing unused dependencies **does help** because:

- Every direct dependency pulls a subtree of transitive packages.

- Each subtree increases your audit surface.
- Removing 1 unused package can eliminate 20–200 transitive packages.

But you must do it carefully in a Next + TS + shadcn setup.

✅ **Step 1 — Find unused dependencies**

🔷 **Recommended tool: depcheck**

Install temporarily:

npm i -D depcheck

Run:

npx depcheck

It will show:

- Unused dependencies
- Unused devDependencies
- Missing dependencies (important)
- Files where usage was detected

⚠️ **Important: depcheck is NOT 100% accurate for Next.js**

Because:

- Dynamic imports
- shadcn code generation
- Next plugins
- Tailwind config usage
- Jest setup files
- MSW setup

So you must manually validate before removing anything.

✅ **Better tool (more accurate for TS + Next)**

Use:

npx knip

If not installed:

npm i -D knip

Then:

npx knip

knip is more modern and understands:

- TypeScript
- Next.js
- Jest
- config files

I recommend knip over depcheck for enterprise TS repos.

✅ **Step 2 — Remove unused safely**

For each reported unused dependency:

1. Confirm it's really unused:

npm ls <package-name>

2. Search codebase:
    - VSCode global search
    - Or:

Select-String -Path .* -Pattern "<package-name>" -Recurse

3. If truly unused:

npm uninstall <package-name>

Then:

npm install
npm run build
npm run dev
npm run lint

🚗 **Do NOT remove these (even if depcheck flags them)**

Common false positives in Next projects:

- next
- react
- react-dom
- typescript
- tailwindcss

- postcss
- autoprefixer
- @types/*
- eslint-*
- jest, ts-jest
- msw
- packages referenced only in config files

✅ **Step 3 — Check for duplicated packages (very important for audit)**

Sometimes you don’t have unused packages — you have duplicated versions.

Run:

npm ls

Look for:

deduped
invalid
multiple versions of same package

To automatically dedupe:

npm dedupe

Then re-run audit.

✅ **Step 4 — Check for outdated but unused transitive trees**

Sometimes you don’t directly use a package, but it's pulled by something outdated.

Example:

glob@7.x
minimatch@3.x
serialize-javascript@2.x

If npm explain shows it comes from:

- old Jest
- old MSW
- old ESLint plugin

Then upgrading that parent removes the vulnerable subtree.

□ Real-world impact expectation

Removing unused dependencies:

- Reduces audit count
- Speeds up installs
- Shrinks Docker images
- Reduces CVE surface
- Improves cold build time

But:

- Most critical vulnerabilities are usually in deep transitive trees
- The biggest win is upgrading major parents (Next, Jest, etc.)

□ Best minimal sequence for you

Run:

```
npx knip
npm dedupe
npm audit --omit=dev --audit-level=high
```

Then paste:

- knip output
- audit high/critical output

And I'll tell you:

- which deps to remove
- which ones to upgrade
- which overrides to apply
- and which ones are safe to ignore

1) Generate an audit report you can actually triage

JSON report (best)

```
npm audit --omit=dev --json > audit-prod.json
```

Now you have a file you can grep/search easily.

---

2) Extract the "top offenders" (without running explain repeatedly)

A) Quick human workflow (no scripts)

Open audit-prod.json in VS Code and search for these keys:

- "name":
- "severity": "critical"
- "severity": "high"
- "fixAvailable":

You are looking for patterns like:

- the **same vulnerable package** appears again and again (example: semver, cookie, minimatch, glob, json5, tar, undici, etc.)
- the **same "via" chain** repeats (same parent pulling it)

Usually you will find **3–6 parents** causing almost everything.

B) One command that groups roots (works on Windows PowerShell)

Run:

```
npm audit --omit=dev --json | node -e "
const fs=require('fs');
let data=''; process.stdin.on('data',d=>data+=d);
process.stdin.on('end',()=>{
 const j=JSON.parse(data);
 const vulns=j.vulnerabilities||{};
 const roots=new Map();
 for (const [pkg,v] of Object.entries(vulns)){
  if(!['high','critical','moderate'].includes(v.severity)) continue;
  // collect 'via' and count
  const via=(v.via||[]).map(x=>typeof x==='string'?x:x.name).filter(Boolean);
  for(const r of via.length?via:[pkg]){
   roots.set(r,(roots.get(r)||0)+1);
  }
 }
 const arr=[...roots.entries()].sort((a,b)=>b[1]-a[1]).slice(0,25);
 console.log(arr.map(([k,c])=>`${c}\t${k}`).join('\n'));
});
"
```

This prints a ranked list of packages that appear most in the vulnerability chains — **your real targets**.

---

3) Then fix by "root parent", not by every leaf

For each top offender printed, do **exactly one**:

npm explain <name>

Not 40 times — usually 5–8 times.

That tells you what top-level package brings it in.

---

4) The usual resolution patterns

Pattern A — fix requires upgrading a parent package

Example outcomes:

- some-old-lib -> glob@7 -> …
  Fix:
- upgrade some-old-lib to a version that no longer depends on that chain

Command:

npm i <parent>@latest

Then:

npm audit --omit=dev --audit-level=high

Pattern B — fix exists only via overrides

If the parent won't update soon (or your corp registry blocks), use overrides.

Example:

```
"overrides": {
  "glob": "^10.4.0",
  "minimatch": "^9.0.0"
}
```

Then clean reinstall:

```
Remove-Item -Recurse -Force node_modules
Remove-Item -Force package-lock.json
npm install
npm audit --omit=dev --audit-level=high
```

Pattern C — npm audit is flagging something that's technically "prod" but not runtime reachable

Some packages end up counted as prod due to dependency placement (badly categorized upstream). You still fix if severity is critical/high, but this explains why lists feel huge.

---

5) Why your list is "long" even with --audit-level=high

Important nuance: --audit-level=high filters *what fails the command*, but npm can still print a lot depending on advisory grouping and dependency graph.

That's why the JSON + grouping approach is the sane way.

---

6) The correct sequence from here (do this exact order)

1. Create the prod JSON:

npm audit --omit=dev --json > audit-prod.json

2. Run the grouping command above (prints top 25 offenders)
3. For the top ~5–8 names:

npm explain <name>

4. Decide:
    - upgrade parent package(s), OR
    - add overrides
5. After each change:

npm install
npm run build
npm run lint
npm audit --omit=dev --audit-level=high

---

7) One more thing in your screenshots: .npmrc always-auth

You have:

- npm warn Unknown project config "always-auth"

This is coming from a **project .npmrc** entry always-auth=true. Newer npm versions are getting stricter and may deprecate that key.

It won't cause vuln count issues, but keep in mind:

- your corp registry config may affect which patched versions are available
- if your registry caches old versions, you may need npm cache clean --force after overrides/upgrades

---

What I need from you to give the exact overrides/upgrade list

Paste **only the first ~60 lines** of:

npm audit --omit=dev --audit-level=high

Specifically the part that shows the *first few advisories* with package names and "fix available" notes.

Or even better: paste the **output of the grouping command** (top offenders). That single list will let me tell you:

- which 3–6 roots to target
- the safest overrides entries to add

- which parent packages to upgrade (and likely candidates in your stack: msw, jest/ts-jest, eslint ecosystem, certain chart libs, etc.)