How to Fix and Prevent Prototype Pollution

To fix prototype pollution, stop writing untrusted keys into objects that inherit from `Object.prototype`. Use a `Map` or `Object.create(null)` for userkeyed data, copy only allowlisted fields, validate input with strict schemas, patch vulnerable dependencies and add automated checks to code review and CI. Blocking the string `proto` alone is not a reliable fix.

Finding a prototype pollution bug is satisfying. Fixing it so that it stays fixed is the harder job. Teams often patch the one function named in a report, ship the change and meet the same flaw again six months later in a different helper or a newly added dependency. Durable prevention needs more than a clever patch. It needs safe defaults in design, disciplined review, automated detection and tests that fail when someone reintroduces the problem.This guide is organized around the software lifecycle rather than a list of tricks. We start with the fix itself, then move through design, code review, tooling, pipeline checks and runtime hardening. If you need the underlying theory first, our reference article on prototype pollution explains how the bug works. Here we focus on making it go away.



Understand What You Are Fixing

Object property pollution happens when untrusted keys reach a recursive merge, clone, or path assignment function and land on a shared prototype. Two routes matter `proto` and

`constructor.prototype`. A fix that closes only one route is incomplete and a fix that filters spellings rather than removing the capability tends to fail under a determined tester.

That insight shapes every recommendation below. Prefer changes that make the dangerous write impossible, such as objects with no prototype, over changes that try to guess every malicious string.

How to Fix Prototype Pollution Three BeforeandAfter Examples

Fix 1 A recursive merge

Vulnerable version

javascript

```javascript
function merge(target, source) {
  for (const key in source) {
    if (typeof source[key] === object && source[key] !== null) {
      if (!target[key]) target[key] = {};
      merge(target[key], source[key]);
    } else {
      target[key] = source[key];
    }
  }
  return target;
}
```

The line target[key] returns Object.prototype when the key is proto, so the recursion walks straight into the shared prototype. A safer version rejects dangerous keys, iterates only own properties and never treats an inherited value as a place to write
javascript

```javascript
const BLOCKED = new Set([proto, constructor, prototype]);

function safeMerge(target, source) {
  for (const key of Object.keys(source)) {
    if (BLOCKED.has(key)) continue;
    const value = source[key];
```

```
    if (value !== null && typeof value === object && !
Array.isArray(value)) {
      if (!Object.hasOwn(target, key) || typeof target[key] !==
object) {
        target[key] = {};
      }
      safeMerge(target[key], value);
    } else {
      target[key] = value;
    }
  }
  return target;
}
```

Treat the blocklist as a safety net. The stronger design is to avoid deep merging untrusted data at all, or to merge into Object.create(null).Handson practice in web application hacking labs and web security CTF challenges builds that intuition and the web application CTF skill builder gives a structured path.

Fix 2 A pathsetting helper

A function that sets values at usersupplied paths like `a.b.c` has the same flaw. Validate each path segment before walking it

javascript

```JavaScript
function setPath(obj, path, value) {
  const keys = path.split(.);
  if (keys.some(k => k === proto || k === constructor || k ===
prototype)) {
    throw new Error(Invalid path);
  }
  let current = obj;
  for (let i = 0; i < keys.length  1; i++) {
    if (!Object.hasOwn(current, keys[i])) current[keys[i]] = {};
    current = current[keys[i]];
  }
  current[keys[keys.length  1]] = value;
```

```
}
```

Rejecting the request outright is better than silently skipping keys, because the error becomes a signal you can log and alert on.

Fix 3 Replace the structure

When keys come from users, such as tags, counters, or form field names, avoid ordinary objects entirely

javascript

```JavaScript
const counts = new Map();
counts.set(userKey, (counts.get(userKey) ?? 0) + 1);
```

A Map stores keys in an internal table that never touches the object property namespace, so a key named proto is just a string. The tradeoff is ergonomics Map does not serialize directly with JSON.stringify, so convert at the boundaries.

Phase 1 Design Secure Defaults

Prevention is cheapest at design time. Agree on a few team rules before anyone writes a merge helper

- Allowlist over merge. When the input has a known shape, copy only expected fields into a new object, for example { name input.name, email input.email }. No attacker key ever becomes a property name.
- Prototypefree containers for dynamic keys. Use Map or Object.create(null) whenever keys are not fixed in code.
- Explicit permission values. Authorization should compare against allowlisted strings, such as role === editor and should never depend on a property being absent or falsy.
- Schema validation at every boundary. Reject unknown keys on request bodies with a validator such as Ajv, using strict settings like additionalProperties false. Apply it in shared middleware, not route by route, so no endpoint is forgotten.

These rules overlap with the defenses against mass assignment, which is no accident. Both bugs come from binding external keys onto objects without a filter, so one validation layer helps with both.

Phase 2 Secure Code Review

Human review catches what tools miss, especially handwritten helpers and trust assumptions. Give reviewers a short, specific checklist.

Review question	Why it matters
Does this function recurse over keys it did not create?	Deep merge, clone and pathset functions are the usual sinks
Can an attacker influence the keys of the source object?	Control of keys, not just values, enables pollution
Are `__proto__`, `constructor` and `prototype` all handled?	Partial blocklists leave routes open
Does the code read optional properties that control access or behavior?	These reads are gadgets once the prototype is polluted
Is `Object.hasOwn` used where ownership matters?	It stops inherited values from satisfying checks
Does a new dependency perform deep merges or compile templates from options?	Third party sinks and gadgets are invisible in your own diff

Practice improves review speed. Working through source code review labs trains you to spot unsafe key handling quickly and teams can use the same labs to calibrate reviewers before a security sensitive release.

Phase 3 Secure Code Review Tools and Automation

Automation covers the volume that reviewers cannot. Each category sees a different slice of the problem.



Tool type	What it finds	Strength	Limitation
Static analysis (SAST)	Unsafe merge, clone and path assignment patterns in source	Scales across the whole codebase, runs before deployment	Cannot always tell whether input is attackercontrolled
Software composition analysis (SCA)	Dependencies with published advisories	Catches known vulnerable packages and transitive copies	Reports presence, not reachability; misses your own code
Dynamic testing (DAST, proxybased)	Pollution reachable in a running application	Confirms real behavior	Needs an observable effect; a clean run is not proof
Unit and regression tests	Reintroduced bugs in your own helpers	Cheap and specific	Only covers functions you test

For static analysis, CodeQL ships queries for prototype polluting functions and Semgrep supports community and custom rules for the same patterns. A comparison of options and where they fit in a pipeline appears in our overview of the best SAST tools. For dependencies, npm audit, GitHub Dependabot and OSVScanner flag packages with known advisories, which historically include several lodash, minimist and jQuery issues such as CVE201910744, CVE202144906 and CVE201911358.

For dynamic checks, an intercepting proxy such as Burp Suite lets testers replay requests with prototype targeting keys and watch for behavior changes. Automated results are leads, not verdicts, so confirm each one manually and treat a clean scan as absence of evidence rather than proof of safety.

Phase 4 Pipeline Checks and Regression Tests

Add prevention to the build so it runs on every change.

1. Fail builds on known advisories. Run npm audit auditlevel=high or an SCA tool in CI and commit your lockfile so the audited versions are the shipped versions.
2. Run SAST on pull requests. Block merges on new prototype pollution findings and route false positives to a documented suppression process.
3. Force patched transitive dependencies. When a parent package has not released a fix, use npm overrides or Yarn resolutions, then reaudit.
4. Add a regression test for every fixed helper. A short test proves the fix and protects it from future edits.

Phase 5 Runtime Hardening as Defense in Depth

Runtime controls reduce damage when something slips through. None of them replace fixing the sink.

Object.hasOwn() in permission and option checks. It ensures that a value must belong to the object itself, so a polluted prototype cannot satisfy it. It requires a modern runtime that supports ES2022.
Object.freeze(Object.prototype). Freezing the shared prototype makes pollution attempts fail. It can break libraries and polyfills that legitimately extend the prototype and in strict mode failed writes throw errors, so test thoroughly. Apply it at application startup, before any dependency loads.
The Node.js disableproto flag. Using disableproto=delete or disableproto=throw removes or restricts the proto accessor for the whole process, including dependencies. It does not stop the constructor.prototype route and it may break packages that rely on proto, so confirm current behavior in the Node.js documentation before relying on it.
Request level alerting. Log and alert on request bodies or query strings that contain proto, constructor, or prototype as keys. Legitimate clients almost never send them, so alerts are high signal and they give you early warning of probing. Web application firewall rules can help here, but string matching can be bypassed by encodings and parser quirks, so use it as a speed bump.

Conclusion

Preventing prototype pollution is mostly about habits. Choose data structures that cannot reach a shared prototype, validate input at every boundary, review recursive helpers with a specific checklist and let tools and tests watch for regressions. Runtime controls such as Object.hasOwn, prototype freezing and the disableproto flag add useful depth, but they work best behind a real fix.

If you want your team to recognize these patterns on sight, train on realistic code. Start practicing at AppSecMaster, where hands-on labs and challenges let you break vulnerable code and then secure it.

Frequently Asked Questions (FAQs)

How do I fix prototype pollution in JavaScript?

Remove the unsafe writing. Replace recursive merges on untrusted data with allowlisted field copies, use a Map or Object.create(null) for usersupplied keys, reject proto, constructor and prototype keys and upgrade any vulnerable dependency. Then add a regression test that checks ({}).polluted stays undefined.

Is blocking proto enough?

No. The constructor.prototype route reaches the same shared object and string filters can be bypassed by parser differences or input that reassembles the blocked word. Block all three names as a supporting layer and rely on structural fixes such as prototypefree objects and strict schemas.

What tools detect prototype pollution?

Static analyzers such as CodeQL and Semgrep flag unsafe merge and assignment patterns, SCA tools such as npm audit and OSVScanner flag vulnerable dependencies and proxy-based testing confirms reachable behavior in running applications. Use them together, because each covers a different part of the problem.

Does Object.freeze(Object.prototype) prevent prototype pollution?

It makes writes to the shared prototype fail, which blocks the attack, but it can break libraries that extend the prototype and can throw errors in strict mode. Test it carefully, apply it before dependencies load and treat it as defense in depth rather than a replacement for fixing the vulnerable code.

How can I stop the bug from returning after a fix?

Add regression tests for every fixed helper, run SAST and dependency audits in CI, enforce schema validation in shared middleware and include prototype pollution questions in your code review checklist. Search the whole codebase for similar helpers after each fix instead of patching only the reported one.