URADICAL

The Dependency Debt

What your package manager isn't telling you about supply chain risk

WHITE PAPER

May 2026

uradical.io

Contents

Executive Summary

Every dependency you add is a trust decision. You are not just importing code; you are importing the security posture, the maintenance discipline, and the operational decisions of everyone who contributed to that package — and everyone who contributed to its dependencies.

The software industry has spent a decade pretending this isn't true. Package managers made acquisition frictionless. Ecosystems celebrated large dependency trees as a sign of thriving communities. Developers learned to reach for `npm install` or `go get` before asking whether they actually needed outside help.

The bill is arriving.

Supply chain attacks — where malicious actors compromise a dependency to compromise its downstream consumers — have moved from theoretical concern to operational reality across every major ecosystem. Node.js, Python, Ruby, and Go have all had documented, impactful incidents. In February 2025, a backdoored typosquat of the widely trusted Go database module `BoltDB` was discovered to have been active and cached by the Go Module Proxy for over three years. The GoSurf research project, evaluating 500 of the most widely used Go packages, found exploitable attack vectors in every single one.

This paper makes three arguments:

First, the cost of dependency sprawl is not zero, and most organisations have never modelled what it actually is. We provide a framework for doing so.

Second, the mitigations offered by modern package managers — checksums, lock files, immutable proxies — are necessary but not sufficient. They protect against some attacks; they are silent about the attack surface you accepted when you pulled the dependency in the first place.

Third, the most effective supply chain security measure available to a development team is also the least popular one: use fewer dependencies. Go's own design philosophy has been making this argument since 2012. The ecosystem data is now catching up.

1. The Free Lunch Fallacy

When a developer runs `npm install left-pad` — or any equivalent in any ecosystem — the mental model is simple: I get functionality, I pay nothing.

This model is wrong. It has always been wrong. The cost is just deferred and diffuse enough that it rarely shows up in a sprint review.

The actual costs of each dependency are:

Security surface. Every package you import is code you are responsible for running but did not write and cannot fully audit. It can contain malicious code placed there intentionally, vulnerabilities introduced through negligence, or both. These risks apply not just to the package itself but to every package it transitively depends on.

Maintenance burden. Packages have breaking changes. Maintainers abandon projects. Security advisories require emergency response. The 2022 `colors` and `faker` deliberate sabotage — where a maintainer intentionally corrupted their own widely used npm packages — demonstrated that even packages with high download counts and long track records can become operational liabilities overnight.

Compliance exposure. Dependency licensing is a legal problem. GPL-licensed transitive dependencies in commercial software are a real liability. Most teams have no accurate inventory of what licences they have accepted.

Build and runtime overhead. Large dependency trees increase build times, binary sizes, container image sizes, and startup latency. These are engineering costs that compound across a team and a CI/CD pipeline.

Cognitive overhead. Every dependency is a system a developer must understand to debug when it behaves unexpectedly. This cost scales non-linearly: a flat dependency of five packages is manageable; a graph with a hundred transitive nodes is archaeology.

None of this appears in the thirty seconds it takes to add a package. It accumulates silently until an incident makes it visible.

The industry term for accumulated technical liability is "technical debt." Dependency sprawl is a specific, measurable category of that debt. We call it **dependency debt**: the aggregate future cost — in security remediation, maintenance engineering, compliance work, and incident response — of dependencies accepted without adequate evaluation.

2. The Attack Surface, Ecosystem by Ecosystem

Supply chain attacks are not hypothetical. They are documented, recurring, and increasing in sophistication across every major package ecosystem. This section is not an exercise in language criticism. No ecosystem is uniquely negligent, and no language community has failed to take the problem seriously. The point is simpler and less comfortable: the problem is universal, the mitigations are imperfect everywhere, and the dependency model itself is the common root cause.

2.1 npm: The Largest Target

Node.js's npm registry is the world's largest package repository and its most actively exploited. The attack patterns are well-established.

Typosquatting places a malicious package at a name one character from a legitimate one. The IconBurst campaign of 2022 deployed twenty-four such packages designed to skim data from form inputs in web applications. Packages with legitimate-looking names accumulated thousands of downloads before detection.

Dependency confusion attacks exploit the way some package managers resolve private versus public package names. By publishing a public package with the same name as an organisation's internal package at a higher version number, an attacker can cause internal build systems to pull the malicious version.

Protestware — deliberate sabotage by a legitimate maintainer — emerged as a category in 2022. The `node-ipc` package, widely used in Vue CLI tooling, was modified to overwrite files on systems with Russian or Belarusian IP addresses. This was not a compromise; it was an intentional act by the registered owner. The package had millions of weekly downloads.

The shared characteristic across these attacks: they succeed because teams added the dependency without modelling what happens when that trust is violated.

2.2 PyPI: Infrastructure-Scale Risk

Python's Package Index has seen systematic abuse by actors seeding it with crypto miners, credential stealers, and exfiltration tools — often dozens of packages in a single campaign. The attack economics are well understood: a single widely-used package, compromised, can reach millions of machines within hours of a new version being published, before any advisory is issued.

PyPI's challenge is structural. Anyone can publish; there is no pre-publication review; package names are case-insensitive in ways that enable squatting. The Trust and Safety team operates reactively, responding to reports rather than preventing publication.

2.3 Maven Central: The Enterprise Blind Spot

Java is the dominant language in enterprise software — banking systems, healthcare infrastructure, government services, insurance platforms. Where Java goes, Maven goes with it. Maven Central is the primary repository for JVM-based packages, and it operates under the same optimistic trust assumptions as every other registry: publish freely, audit reactively.

The structural risk is compounded by how the JVM resolves classes at runtime. Researchers at KTH Royal Institute of Technology — the same group behind GoSurf — documented a novel attack class they named **Maven-Hijack**. The attack exploits the order in which Maven packages transitive dependencies and the way the JVM's classloader resolves class names. If a malicious dependency contains a class with the same fully-qualified name as a legitimate one, and that dependency is packaged earlier in the classpath, the JVM loads the malicious version. The legitimate class is silently shadowed. No library names change. No checksums fail. The main codebase is untouched.

The researchers demonstrated this by compromising the **Corona-Warn-App** — Germany's widely used open-source COVID-19 contact tracing system — using a corrupted JSON validation library buried in the transitive dependency tree. The result was full control over the application's database connection logic. A small, deeply nested dependency; a complete database takeover.

The implications for enterprise Java deployments are significant. Applications with deep, complex Maven dependency trees — which is to say, most of them — have no reliable runtime mechanism to detect that a class they are calling is not the class they expect. The JVM's classpath model was not designed with adversarial dependency trees in mind. Sealed JARs and Java Modules mitigate the risk but are not yet standard practice.

Compounding this, research into Maven Central build reproducibility found that approximately 84% of the most commonly used Maven artifacts are not built using a transparent CI/CD pipeline. There is no verifiable chain of custody between the source code and the published binary for the majority of enterprise Java dependencies. What you download is not demonstrably what was built from what you can read.

The enterprise context matters here. A Java application at a bank or hospital may have hundreds of direct dependencies and thousands of transitive ones. The remediation economics from Section 3 apply with greater force: larger engineering estates, higher regulatory exposure, longer patching cycles, and a dependency culture historically dominated by frameworks — Spring, Hibernate, Apache Commons — whose own transitive graphs are substantial.

2.4 RubyGems

The GemStuffer campaign of May 2026 demonstrated an underappreciated attack pattern: using the package registry not to distribute malware but as an exfiltration channel, packaging scraped data into junk gems published from throwaway accounts. RubyGems is a vector, not just a target.

2.5 Go: Not Immune

Go's supply chain design includes mitigations that other ecosystems lack. The `go.mod` file pins exact dependency versions. The `go.sum` checksum database creates a globally consistent, append-only record of module hashes. The Go Module Proxy provides immutable cached copies. There are no post-install hooks; fetching and building code cannot execute it.

These are meaningful protections. They are also incomplete.

The Go ecosystem has two documented structural vulnerabilities that no checksum database addresses. First, **repo-jacking**: when a GitHub user deletes their account or renames their repository, the namespace becomes available for registration. VulnCheck identified more than 15,000 Go repositories in this hijackable state — packages whose import paths resolve to a namespace any attacker can now claim. Second, **typosquatting remains effective** despite Go's nominal decentralisation, because developers searching for a package by name can be misled before the module system's integrity checks have any opportunity to intervene.

Both vulnerabilities have been exploited in the wild. The widely used Go `cli` package was compromised via a malicious fork that introduced a hidden `init` function to collect private system information and exfiltrate it to a remote host. The attack vector was the same one GoSurf classifies as I2: initialisation hooks that execute silently before `main`, regardless of whether the package is directly invoked.

In February 2025, Socket Security researchers discovered a malicious package — `github.com/boltdb-go/bolt` — that had been active in the Go Module Proxy for over three years. The package was a typosquat of `github.com/boltdb/bolt`, a database module trusted by organisations including Shopify and Heroku, with more than 8,300 downstream packages depending on it.

The attack exploited Go's immutability guarantee against itself. The threat actor published a backdoored version to GitHub, allowed the Go Module Proxy to cache it, then rewrote the Git tag on GitHub to point to a clean version. Anyone auditing the GitHub repository would see legitimate code. Anyone installing via the Go CLI would receive the cached malicious version. The proxy's caching design — a genuine security feature — became the persistence mechanism.

The backdoor was a full remote access implant. An `ApiInit()` function, triggered by the legitimate-looking `Open()` database call, established a persistent TCP connection to a command-and-control server at `49.12.198.231:20022`. The C2 address was obfuscated by encoding it as arithmetic on constants named `MaxMemSize`, `MaxIndex`, and `MaxPort`, distributed across multiple files. Standard static analysis tools did not flag it. It remained in active operation for three years.

When Socket reported the finding, analysis of the malicious package's GitHub page showed zero stars, zero forks, and no pull requests — and only two recorded imports, both from a single cryptocurrency project with seven followers. Go does not track download metrics, so the true exposure is unknown. The package was nevertheless served from the proxy to any developer who happened to request it. Obscurity and low adoption are not the same thing as safety; the infrastructure risk is independent of uptake.

Following publication of the research, Google confirmed removal of the module from both the Go Module Proxy and GitHub, and added it to the Go vulnerability database. Google cited Capslock and deps.dev as ongoing mitigations — tools that post-date the three years during which the backdoor was available.

This was not a failure of Go's security model; it was a demonstration that any trust assumption — including "this package appears clean on GitHub" — can be exploited, and that a compromised package can persist in distribution infrastructure long after any reasonable window for detection.

Beyond this specific incident, the GoSurf research project from KTH and the University of Naples Federico II evaluated 500 of the most widely imported Go modules for supply chain attack surface. They identified twelve language-specific attack vectors — covering code execution opportunities at pre-build, initialization, and execution phases of the package lifecycle. Every single one of the twelve attack vectors was found in real-world popular Go packages. Across 500 modules, there were 490,250 interface polymorphism instances, 241,794 constructor methods, 116,141 testing functions, and 642,996 unsafe pointer usages — each a potential vector for hidden malicious code in a compromised package.

The research team also tracked five Kubernetes releases (v1.26 through v1.30) and observed that the attack surface is not static: it shifts with every update, as new features introduce new vectors or refactoring removes old ones. Kubernetes v1.29 showed 90,901 interface instances; v1.28 showed 89,849. The attack surface of your dependencies changes when they do, without your involvement.

2.6 Convergence: March 31, 2026

On March 31, 2026, three attack streams converged on the same target — developers — in a three-hour window between 00:21 and 03:29 UTC.

The most directly relevant to this paper is the Axios compromise. Axios is the most widely used JavaScript HTTP client in existence: over 100 million weekly downloads, present in approximately 80% of cloud and code environments. A North Korean threat actor designated UNC1069 compromised the npm account of the Axios maintainer and published two backdoored versions — `axios@1.14.1` and `axios@0.30.4`. The malicious versions introduced a hidden transitive dependency, `plain-crypto-js`, containing a post-install script that deployed WAVESHAPER.V2, a cross-platform backdoor, across Windows, macOS, and Linux. No user interaction required. Any automated CI pipeline that ran `npm install` in the window may have installed it.

The maintainer was not careless. He was the subject of a months-long, state-sponsored social engineering campaign: a fake company identity, a cloned founder likeness, a real Slack workspace built over time to establish credibility, and eventually a Microsoft Teams call designed to execute the final payload. The attack was tailored to one individual, specifically because that individual held publish access to a package with 100 million weekly downloads. One account. Three hours. A potential foothold in the majority of cloud infrastructure globally.

Claude Code depends on Axios. Any developer who updated Claude Code via npm in the exposure window was potentially in the blast radius of a North Korean supply chain attack through a transitive dependency they had never heard of.

In a separate, simultaneous campaign, a second North Korean-nexus group designated UNC6780 poisoned PyPI packages associated with Trivy, Checkmarx, and LiteLLM, deploying the SANDCLOCK credential stealer. Multiple state-sponsored groups are now systematically targeting open-source package registries — not as opportunistic operations, but as a sustained, coordinated strategy with an understood return on investment: one compromised maintainer account, held briefly, reaches an extraordinary proportion of the global developer estate.

Two things distinguish March 31 from earlier incidents and make it directly relevant to the dependency debt framework.

First, the post-install script mechanism. npm allows arbitrary code execution at package installation time. This is the attack surface that Go's design explicitly eliminates — the Go blog's observation that "fetching and building code cannot execute it" is not an aesthetic preference; it is a direct response to the attack class that UNC1069 deployed. The presence or absence of post-install hooks is a design decision with measurable security consequences.

Second, the maintainer as the terminal vulnerability. No checksum, no lock file, and no SBOM protects against a legitimate maintainer whose account is compromised. The `plain-crypto-js` dependency was introduced by a valid publish from a valid account. It passed every automated integrity check. The trust model of open-source package distribution ultimately terminates in a human being — and state-sponsored

adversaries have concluded that human beings are the most cost-effective attack vector available.

The dependency debt is not only technical. Every package in your tree is a trust relationship with a person.

3. Quantifying the Debt

Most organisations have no model for the cost of a dependency compromise. They have incident response processes. They do not have pre-incident cost models. This is backwards.

The following framework provides a starting point for modelling dependency debt before an incident, not after.

3.1 Discovery Time

The most important variable in supply chain cost is how long between compromise and detection. The BoltDB attack ran for over three years. The average time to discover a data breach — across all breach types — consistently sits above 150 days in industry surveys. Supply chain compromises, which are specifically designed to be invisible within legitimate code, skew this figure upward.

A dependency in production for three years is a dependency that could have been delivering data to an attacker for three years.

Discovery cost model: For each dependency, estimate `P(compromise) Ã— days_to_discovery Ã— daily_data_value`. Even conservative estimates produce significant numbers for dependencies with access to sensitive data.

3.2 Blast Radius

When a dependency is compromised, every system that runs it is affected. This is the blast radius. In the BoltDB case, a database package with access to application data — exactly the packages you import for their privileged access — was the vector.

Blast radius is determined by:

- **Data access**: what data does this dependency touch?
- **Network access**: can it establish outbound connections?
- **Process execution**: can it spawn child processes?
- **Privilege level**: does it run as root, as a service account?

The GoSurf taxonomy provides a useful framework here. At execution time, Go packages can exploit constructors (called on every struct instantiation), reflection (dynamic method dispatch), interface polymorphism (runtime type substitution), unsafe pointers (arbitrary memory access), CGO (C code with no memory safety), assembly (low-level, difficult-to-audit code), dynamic plugins (externally loaded code), and external execution (shell commands). Each vector extends the blast radius of a compromised package.

A database package that also links C code via CGO — a pattern present in popular Go database drivers — has a substantially larger blast radius than a pure-Go alternative.

Blast radius score: For each dependency, rate it across the GoSurf vectors. Aggregate this score with data sensitivity to produce a risk-weighted dependency cost.

3.3 Remediation Engineering Cost

When a dependency is identified as compromised — or merely vulnerable — the remediation path is:

1. Identify all projects importing the affected package, directly or transitively
2. Assess which versions are affected
3. Identify safe alternatives or patches
4. Update and test across all affected projects
5. Deploy to production
6. Verify and close

For an organisation with dozens of services and typical dependency sprawl, this process takes weeks and involves multiple engineering teams. In emergency scenarios, it becomes the highest-priority item across the entire engineering function — displacing planned work, creating technical debt in other areas, and generating significant managerial overhead.

Engineering cost estimate: Count the number of services in your estate. Count the average number of direct and transitive dependencies per service. For each one, model the remediation cycle as: (services_affected × 4 hours average engineering time) + (2 days management overhead) + (QA and deployment costs). For a mid-sized organisation with 30 services and 200 average dependencies, a single ecosystem-wide incident affecting 10% of packages produces around 240 engineering hours of remediation work — roughly £30,000 at current UK contractor rates, before any breach costs.

3.4 Regulatory and Breach Exposure

Under UK GDPR and the Data Protection Act 2018, a personal data breach resulting from a compromised dependency must be reported to the ICO within 72 hours if it is likely to result in a risk to individuals' rights and freedoms. Failure to report carries fines of up to £17.5 million or 4% of global annual turnover. The breach itself carries similar exposure.

NIS2, which entered UK law in 2024, extends supply chain security obligations explicitly to software dependencies for operators of essential services. Cybersecurity regulators across energy, finance, and healthcare are now actively auditing software supply chain practices.

An organisation that has accepted thousands of transitive dependencies without an inventory or a risk model is not compliant with these obligations. It is not a question of whether they will face scrutiny; it is a question of when.

4. The Transitive Problem

The discussion above focuses on direct dependencies — packages you explicitly name in your `go.mod`, `package.json`, or `requirements.txt`. This understates the problem.

Transitive dependencies — packages pulled in by your dependencies — are typically invisible until a tool surfaces them. They are not reviewed before adoption. They are not listed in your `go.mod` as first-class entries. And they carry the same security risks as your direct dependencies, with less visibility and less accountability.

Consider the BoltDB case from a downstream perspective. If your service uses a database package that internally depends on `github.com/boltdb/bolt`, you are a transitive consumer. You may have no idea that `boltdb/bolt` is in your build at all. The threat actor behind `boltdb-go/bolt` was not just targeting organisations that use BoltDB directly; they were targeting every organisation that uses any package that uses BoltDB.

This cascade effect means the actual exposure of a compromise scales with the popularity of the compromised package. BoltDB has 8,300+ direct dependents. Each of those dependents has their own downstream consumers. The reach of a single compromised package in a widely used ecosystem is not linear; it is combinatorial.

Go's `go mod graph` command makes this visible, which is more than most ecosystems offer. Running it on a typical Go service reveals something most developers have never seen: the full, often surprising, dependency tree that their binary includes. Teams should make this exercise routine, not optional.

5. What Low-Dependency Looks Like in Practice

Go's official supply chain security documentation contains a line that deserves to be treated as engineering policy, not cultural observation:

> "A little copying is better than a little dependency."
>
> — Rob Pike, Go Proverbs, GopherCon 2015

This is not a beginner's heuristic. It is the considered position of the team that designed one of the most widely deployed languages in production infrastructure. It is also, notably, the design philosophy behind GoSurf itself: the researchers implemented their entire static analysis tool — parsing Go source, building ASTs, running twelve distinct analysers, and generating JSON reports — in 793 lines of Go using only standard libraries. The tool that maps the supply chain attack surface has no supply chain attack surface of its own.

The standard library and the `golang.org/x/...` packages provide an HTTP stack, TLS, JSON encoding, a cryptography suite, and a wide range of utilities. The design intent is that you can build complex, production-grade systems without a large external dependency graph.

What does this look like in practice? Consider three common dependency decisions:

HTTP routing. The default Go developer instinct is to reach for a router library — gorilla/mux, chi, gin. The standard library's `net/http` package supports path-variable routing natively as of Go 1.22. For the vast majority of services, it is sufficient. The dependency adds code, a maintenance relationship, and a supply chain vector; the standard library adds none of those.

JSON handling. Third-party JSON libraries — particularly those providing struct tagging extensions or performance improvements — are common. For most workloads, `encoding/json` in the standard library is performant, correct, and dependency-free. The case for an external library must be made on specific, measured grounds, not habit.

Database drivers. This is where the case for external dependencies is strongest: database-specific wire protocols are complex and not covered by the standard library. But even here, the number of dependencies should be minimised. A team using PostgreSQL needs a driver; it does not need five packages around that driver for migrations, connection pooling, and query building that each add their own dependency trees.

The practice is not "never add a dependency." It is "the default answer is no, and you must make the case for yes." This inverts the industry norm, where the default answer is yes and the case against is rarely made.

For a team adopting this discipline seriously, the result is a dependency graph that can be read in a single terminal session, audited in a working week, and updated as a deliberate engineering decision rather than an automatic process.

6. A Decision Framework

Before adding any dependency, apply the following questions in order. The first question that produces a definitive answer determines the outcome.

Can the standard library do this? Check the language's standard library and, for Go specifically, the `golang.org/x` extended packages before looking externally. If yes, close the browser tab.

Can this be implemented in under 200 lines? The cost model changes when the alternative is a small, owned implementation. A small, purpose-built function with no external dependencies, living in your codebase under your control, is often the lower-risk choice. Apply this test seriously: a UUID generator, a retry wrapper, a simple token bucket — all of these have been the subject of widely used packages, and all of them can be written in an afternoon.

What is the blast radius of this dependency? Using the GoSurf taxonomy: does this package use CGO, unsafe pointers, reflection, dynamic plugins, or external execution? Does it need network access? Does it run with elevated privileges? Higher blast radius requires a higher justification threshold.

What is the maintenance posture of this package? When was it last updated? Is there a single maintainer? Has the repository changed owners? Are security advisories responded to promptly? A package that hasn't been touched in two years is either very stable or abandoned; you need to know which.

What is the license, and is it compatible with your commercial use? This is not an afterthought. Check it before the dependency is merged.

Is there a verified, pinned version with a checksum? For Go: is it in `go.sum`? For npm: is it in `package-lock.json` or `yarn.lock` with integrity hashes? For Python: is it pinned to an exact version with a hash in your requirements? If not, fix this before proceeding.

If a dependency passes all of these checks and is approved for use, the decision should be recorded — why this package, what alternatives were considered, what the acceptable risk is. This creates an audit trail and makes future dependency reviews tractable.

7. What To Do About It

Three things, in order of impact.

Invert the default. The industry norm is that adding a dependency requires no justification and removing one requires effort. This is backwards, and it is the root cause of the problem this paper has documented. Every dependency addition should require a recorded decision — what the package does, what alternatives were evaluated, what the blast radius is, who maintains it. Dependency removal should be treated as engineering work with a measurable positive return. This costs nothing to implement. It requires only that teams decide the default answer to "should we add this?" is no rather than yes.

Inspect what you build, not what you read. The BoltDB backdoor appeared clean on GitHub for three years while the cached version in the Go Module Proxy was malicious. The Axios compromise introduced a hidden transitive dependency that post-install tooling executed without the main package's manifest changing. Repository audits and lock file reviews are necessary but not sufficient. The attack surface is in the installed package — the bytes that actually execute on your infrastructure. Tooling that analyses installed content rather than declared content is not optional in the current threat environment; it is the minimum viable position.

Reduce the graph before the incident, not after. Run `go mod graph`, inspect your `node_modules`, or enumerate your Python environment. For most production services, the transitive dependency count will be a number nobody on the team can justify. For every dependency in that graph with significant data access, network access, or execution capability, ask whether the standard library, a `golang.org/x` package, or two hundred lines of owned code would serve the same purpose. The answer will be yes more often than expected. Treat each removal as a security improvement, a maintenance reduction, and a compliance simplification simultaneously — because it is all three.

The technical controls — checksums, lock files, signed commits, SBOMs, vulnerability scanners — are all worth having. They do not substitute for a smaller graph. A well-audited list of fifty dependencies is more defensible than a well-scanned list of five hundred. The scanners only know about vulnerabilities that have already been discovered and documented. The attack surface you never accepted cannot be exploited.

8. Conclusion

Open-source dependencies are a documented, active, and compounding security liability across every major software ecosystem. A backdoored Go package was served from production infrastructure for three years before detection. A compromised npm maintainer account, held for three hours, exposed a potential backdoor across 100 million weekly downloads and 80% of global cloud environments. A JSON validation library buried in a transitive dependency tree was sufficient to take over a national health application's database connection. Academic analysis of 500 widely used Go packages found every one of twelve documented supply chain attack vectors present in real-world code. Multiple state-sponsored threat actors are now systematically targeting open-source maintainers as a primary strategy — not opportunistically, but because the return on investment is extraordinary.

NIS2 and UK GDPR make supply chain security a legal obligation for any organisation handling personal data or operating essential services. The ICO has fined organisations for precisely the security posture that dependency sprawl produces: no inventory, no visibility, no patching discipline. Regulatory exposure compounds breach cost. The remediation economics in Section 3 are conservative.

The most effective mitigation available requires no tooling budget, no vendor relationship, and no new process. Rob Pike articulated it as a Go Proverb in 2015: "a little copying is better than a little dependency." It is not an aesthetic preference. It is arithmetic. The question is not whether to reduce dependency debt. It is how much exposure to accumulate before doing so.

References

1. Cesarano, C., Andersson, V., Natella, R., and Monperrus, M. (2024). GoSurf: Identifying Software Supply Chain Attack Vectors in Go. SCORED '24, ACM. https://doi.org/10.1145/3689944.3696166
2. Reyes, F., Bono, F., Sharma, A., Baudry, B., and Monperrus, M. (2024). Maven-Hijack: Software Supply Chain Attack Exploiting Packaging Order. arXiv:2407.18760. https://arxiv.org/abs/2407.18760
3. Vitale, T. (2024, July 11). Six Software Supply Chain Security Concerns in Java Applications You Need to Know About. Systematic. https://systematic.com/int/careers/meet-us/tech-bytes/six-software-supply-chain-security-concerns-in-java-applications-you-need-to-know-about-and-how-to-address-them/
4. Boychenko, K. (2025, February 4). Go Supply Chain Attack: Malicious Package Exploits Go Module Proxy Caching for Persistence. Socket Security. https://socket.dev/blog/malicious-package-exploits-go-module-proxy-caching-for-persistence
5. Valsorda, F. (2022, March 31). How Go Mitigates Supply Chain Attacks. The Go Programming Language Blog. https://go.dev/blog/supply-chain
6. Pike, R. (2015). Go Proverbs. GopherCon. https://www.youtube.com/watch?v=PAAkCSZUG1c&t=9m28s
7. Baines, J. (2023). Hijackable Go Module Repositories. VulnCheck. https://vulncheck.com/blog/go-repojacking
8. Henriksen, M. (2021). Finding Evil Go Packages. https://michenriksen.com/archive/blog/finding-evil-go-packages/
9. Edwards, J. (2026, May 13). GemStuffer Campaign Abuses RubyGems as Exfiltration Channel. Socket Security. https://socket.dev/blog/gemstuffer
10. Ohm, M., Plate, H., Sykosch, A., and Meier, M. (2020). Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. DIMVA 2020.
11. Ladisa, P. et al. (2023). SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. IEEE S&P 2023.
12. Synopsys CyRC. (2024). Open Source Security and Risk Analysis (OSSRA) Report.
13. Jones, C. (2025, February 4). Poisoned Go programming language package lay undetected for 3 years. The Register. https://www.theregister.com/2025/02/04/researcher-sniffs-out-three-year-go-supply-chain-attack/
14. uRadical Engineering. (2026, March 31). What Happened on March 31, 2026 Should Scare Every Engineering Team. uRadical. https://uradical.io/latest-news/march-31-2026-should-scare-every-engineering-team/