DevOps, DevSecOps, and Not Getting Burned by Software You Bought

Mon, Sep 21, 10:00 AM EDT · https://scottslab.io/posts/devops-devsecops-and-acquired-software



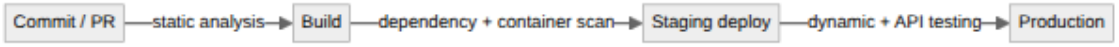
TL;DR

"Shift left" is a slogan until it's a specific gate at a specific pipeline stage with a specific decision attached: block the build or just log the finding. Infrastructure as code is a security control and a new attack surface at the same time: the same script that eliminates config drift can also propagate one bad permission to every environment that imports it. And the software you didn't write is still your risk to manage: an SBOM tells you what's inside it, a contract and an escrow agreement tell you what happens when the vendor can't support it anymore, and even a SaaS product leaves you holding the half of the risk that lives in your own account settings.

DevOps exists to resolve an old tension: developers are rewarded for shipping fast, operations for keeping things stable, and left alone those incentives fight. Automation and shared ownership close the gap: the same pipeline, the same on-call rotation, so speed doesn't get bought by quietly degrading stability. DevSecOps is the obvious next step, folding security into that same pipeline as code instead of leaving it as a separate gate someone has to remember to open.

Shift left is a claim about your pipeline, not your calendar

"Shift left" gets used as a synonym for "think about security earlier," which is true but not actionable. The actual requirement is that security checks become automated gates *inside* the pipeline stages that already exist, not a separate review that happens to run before deployment instead of after. Static analysis runs on every commit or pull request, before merge. Dependency and container scanning run on every build, because a base image or a library can go from clean to vulnerable overnight without a single line of your own code changing. Dynamic and API testing run against a deployed staging environment as part of the same release candidate that would otherwise go straight to production. None of that is "earlier" in a vague sense. Each one is bound to a specific event in the pipeline that already fires on every change, which is the only way a check survives being someone's job.



The decision that actually determines whether any of this matters is blocking versus reporting. A gate that only logs a finding is a dashboard nobody reads under deadline pressure. A gate that blocks every finding on day one, at whatever severity the scanner happens to flag, gets an exception request within a week and a standing bypass within a month. That's not hypothetical, it's the default outcome of turning on a scanner with its factory settings. The practice that actually holds is to block on a severity and exploitability bar set deliberately low at first, report everything else, and ratchet the blocking bar up as the team clears the backlog, so the gate is always slightly ahead of where the codebase already is, never wildly ahead of it. A blocking gate nobody can pass isn't a security control, it's a change-management problem wearing a security control's badge.

Infrastructure as code is a control and a target

Provisioning from versioned scripts instead of hand-configuring servers is a real security improvement. It makes environments reviewable, and it enables immutable infrastructure, where you don't patch a running box, you rebuild it from an updated image and replace it. That kills the class of incident caused by someone logging in to "quickly fix" a server and leaving it in an undocumented state.

The part that gets skipped is that the same script is now an attack surface. A hardcoded credential or an over-permissioned IAM role written into a shared module doesn't stay a local mistake. It gets imported into every environment that pulls that module, so one bad review propagates the error at the speed of `terraform apply` instead of staying contained to one box. State files themselves are also a target: they routinely contain resource attributes and sometimes secrets in plaintext, and

they're exactly the kind of file that ends up in a public bucket by the same misconfiguration reflex that causes everything else on this list. The fix is treating infrastructure code like application code for review purposes: policy-as-code scanning in the same pipeline, catching a public S3 bucket or a wildcard IAM policy before `apply`, not after an audit finds it live.

The software you didn't write is still yours to secure

Buying or adopting software instead of writing it doesn't transfer the risk. It just changes what tools you have to manage it, and the biggest one is that you usually can't read the source. That split runs along a real technical line. Interpreted languages such as Python, Ruby, and R stay human-readable at runtime, so a security review can read the actual logic. Compiled languages such as C, C++, and Java ship as binaries, so there's no source to read at all: understanding what one of them actually does without vendor cooperation means sandboxing it and watching what it does, or decompiling it, not sitting down with a code review. That distinction is what determines how you can even assess a third-party component before it runs in your environment, not just how comfortable you are reading its output.

Since May 2021, U.S. federal Executive Order 14028 has pushed a real, concrete answer to that gap: a Software Bill of Materials, an inventory of exactly what's inside a piece of software. NTIA's minimum-elements guidance built on the order defines what a usable SBOM actually contains: baseline data fields identifying each component, machine-readable automation support so it can be generated and checked continuously rather than produced once for a sales call, and defined practices for when and how it gets regenerated. Ask a vendor for one before you adopt their product, not after an incident, because that's the only point in the relationship where you have real leverage to ask.

The rest of vendor risk management is less exotic but no less necessary: assess what a new piece of software touches against the CIA triad before it's in production, not after; run it through the same network and web-application scanning you'd run against anything you built; patch it promptly, because published disclosures are a race and attackers read them the same day you do; and don't let "it's open source" become an excuse to skip patching, since community components rarely announce updates the way a vendor's account team does. Software escrow (a neutral third party holding a copy of the vendor's source, released to you if they go bankrupt or discontinue the product) covers business continuity, but it's a snapshot of code, not the vendor's build pipeline, their institutional knowledge, or a support contract; treat it as a last resort, not a substitute for vendor due diligence. And even fully-managed SaaS doesn't erase your half of the job: you still own account security, privilege assignment, and IP allow-listing on your side of that relationship. "We bought it" has never meant "someone else owns the risk." It means the risk got split, and the half that's yours is still yours whether you wrote a line of the code or not.
