

---

# What is serverless security?

When you shift to serverless, it feels like infrastructure worries vanish—but unseen functions can harbor hidden risks. Forgotten endpoints, unchecked triggers, or overly broad permissions may silently expose data or inflate your bill, and you might not realize until it's too late.

Suppose you wake to a surprise invoice after a bot swarm hammered an unprotected endpoint overnight, or discovering a misconfigured function leaked sensitive information because its trigger was left open. Without clear visibility and guardrails, small oversights can cascade into major incidents, eroding trust in your serverless deployments and draining resources to investigate and fix.

By adopting a focused serverless security approach—starting with visibility into deployed functions, embedding core hardening practices, and integrating checks into your workflows—you catch issues early and prevent surprises. A phased, iterative strategy helps you balance agility with protection, so you can leverage serverless benefits confidently.

## How can you uncover hidden serverless risks?

When functions appear and disappear on demand, it's easy to lose sight of what's running and how it's accessed. Overlooking a stray test endpoint or an unchecked trigger can let data slip or cause unexpected costs.

### Tracking deployed functions

If you haven't kept an updated list of all functions, you might discover later that a leftover function is publicly accessible. For example, a debug function meant for internal testing could quietly accept external requests. To avoid surprises, periodically query your environment (via provider APIs or simple scripts) to list active functions and note their triggers. You don't need a formal registry at first—just a shared document or dashboard where you jot down each function's role and exposure. Over time, this habit becomes a safety net against forgotten endpoints.

### Observability and tracing

When an error or unusual activity shows up, reconstructing what happened across several short-lived functions can feel like chasing shadows. Imagine an unexpected data change: without consistent logs and a way to follow a request's path, you spend hours hunting for the source. To guard against that, make sure each function emits structured logs including a request or correlation ID, and send them to a central place. Enabling distributed tracing (e.g., with OpenTelemetry or your cloud provider's tracing service) means you can click through the sequence of invocations. Even a small initial setup—adding a consistent log format and a basic dashboard—pays off when you need to diagnose problems quickly.

### Monitoring costs and anomalies

A sudden surge in invocations can balloon your bill before anyone notices. If there's no baseline, you may not spot an unusual spike until invoices arrive. Suppose a public endpoint gets hit by a bot swarm overnight; without alerts, costs escalate silently. Start by viewing invocation counts, error rates, and average durations over time. Set a simple alert for when invocations exceed typical peaks. Many cloud consoles let you tie budget alerts to spend thresholds. Even a basic dashboard that you glance at weekly helps you catch odd patterns early and investigate before

---

it becomes a crisis.

## What core practices reduce serverless vulnerabilities?

Core hardening practices prevent many common issues. Rather than overwhelming details, focus on a few essentials that fit naturally into development and deployment.

### Designing small, focused functions

When a single function handles multiple tasks—authentication, data processing, notifications—a flaw in one area can expose the rest. For instance, if your notification code accidentally logs sensitive identifiers, it could leak data tied to payment or user records. Splitting functionality into clear, single-purpose functions confines risk. As you refactor, document each function’s trigger, inputs, and outputs in a simple note.

**Over time you build clarity:** each function’s purpose stays obvious, and permission scopes remain tight.

### Applying least-privilege permissions

Broad permissions are tempting for convenience (“just grant full access so it works”), but a compromised function with excessive rights can do serious damage. Imagine a function that only needs to read a storage bucket but was given write or delete privileges; if someone exploits it, they could corrupt or remove data. To resist that, observe what resources a function actually touches (e.g., through test runs or logs) and grant only those actions. Integrate a lightweight check in your pipeline that warns when a role seems broader than necessary. Even a manual review for critical functions improves safety.

### Validating inputs early

Unchecked inputs lead to injection or logic errors. If a function expects a numeric ID but accepts arbitrary text, an attacker might inject malicious payloads or cause crashes. Enforcing validation as soon as data arrives—ideally at the API gateway or event ingestion point—rejects bad payloads before they reach business logic. For example, define a simple schema for JSON fields or check file size/type for uploads. In code, use parameterized queries and safe parsing libraries. This habit cuts off many [attack vectors](#) at the boundary.

### Managing dependencies and secrets

Including third-party libraries boosts productivity, but a vulnerable package used in several functions can spread risk widely. If a [CVE](#) appears, manual patching may miss some functions. Embed automated dependency scanning in your CI/CD so you’re alerted immediately when known issues surface. Keep manifests and lockfiles up to date, and prune unused libraries to [shrink the attack surface](#). For secrets, avoid hard-coding credentials—instead, use managed secret stores. Grant each function access only to what it needs at runtime, retrieve secrets securely, and clear them after use. Automating rotation and audit logging helps spot anomalies.

### Encrypting data in transit and at rest

Unencrypted communication or storage can expose sensitive information if misconfigured. Picture a log bucket left unencrypted containing user details. Enforce HTTPS/TLS for all calls—client to API and function-to-service—and enable provider-managed encryption for storage (buckets, databases) with automated key rotation. Resist tempting shortcuts (e.g., disabling TLS

---

in dev) unless you're confident it never affects production. Mask or avoid logging sensitive plaintext. Regularly check encryption settings so nothing slips through the cracks.

## How to integrate serverless security into workflows?

Security that lives outside developers' normal flow often gets skipped. Embedding checks and [response plans](#) into workflows makes protection part of everyday work rather than an afterthought.

### Automating checks in pipelines

Waiting for manual reviews invites mistakes. A pull request adding a new dependency with a known vulnerability can slip through unless the pipeline flags it. Start by adding a dependency scan step; if it finds a critical issue, it fails the build, prompting an immediate fix. Next, include a simple IAM policy check—e.g., warn if a role has wildcards or seems broader than a baseline. Over time, you refine these steps, but even basic scans catch many problems early, saving headaches later.

### Preparing incident response for serverless

When something goes wrong—unauthorized invocations, unexpected costs, or data anomalies—teams need clear, practiced steps. Without a runbook, disabling the offending function or revoking its credentials can be chaotic. Draft a concise guide: how to identify the function in logs, how to disable its trigger or remove permission, where to fetch relevant logs/traces, and how to notify stakeholders. Run a quick tabletop exercise: simulate a scenario and walk through the steps. This preparation ensures that when real incidents occur, you react swiftly rather than scrambling.

### Fostering collaboration and culture

Security works best when everyone feels responsible. If developers see security checks in their IDE or pipeline feedback, they fix issues early rather than later. Share brief tips in team meetings: "We caught a bad dependency in CI yesterday—nice!" Encourage questions like "Have you validated this new endpoint's inputs?" Periodically hold informal workshops where dev, operations, and security discuss recent findings or emerging risks. This keeps security in mind without creating heavy processes.

## How to evolve serverless security as you scale?

As usage grows and environments multiply, you need practices that scale gracefully without overwhelming the team.

### Embracing specialized tools selectively

At small scale, simple scripts and manual reviews may suffice. But when functions proliferate, automated [detection of anomalous behavior](#) or permission drift becomes valuable. Pilot a runtime protection tool on a subset of functions to see if it surfaces unexpected patterns. Similarly, test a permission-analysis service that flags roles wider than typical. If these add genuine visibility without undue noise or cost, expand their use. The key is to choose tools that integrate smoothly into your existing workflows.

### Maintaining consistency across environments

---

If you deploy on multiple clouds or separate accounts, divergent configurations create weak spots. Use infrastructure-as-code modules (Terraform, Pulumi) to define uniform IAM roles, encryption settings, and logging setups. This way, a security setting updated in one place propagates elsewhere. Centralize logs and metrics so you can spot anomalies across all environments. Abstracting provider differences via shared templates or libraries reduces effort and risk.

## **Balancing performance optimizations with security**

Techniques like pre-warming functions or provisioned concurrency improve latency but mustn't skip critical initialization (loading current dependencies, fetching secrets, validating configs). Design warm-up routines that still perform necessary checks so you don't run outdated or insecure code. Monitor both performance and security metrics to find the right balance, adjusting settings as usage patterns change.

## **Embedding compliance early**

Handling regulated data retroactively is painful. From the start, map data flows: which functions process sensitive or personal data, and what controls they need (encryption, residency, retention, audit logging). Involve compliance teams in design so these controls are built in, not bolted on. For every new function touching regulated data, confirm it meets requirements before deployment. This practice avoids last-minute rework and reduces audit risk.

Securing serverless architectures demands visibility into deployed functions, disciplined hardening, integrated monitoring, and automated access governance to prevent stale credentials or privilege creep from undermining defenses. [Fidelis Elevate](#) complements these efforts by automating identity and access lifecycle processes—adjusting or revoking permissions tied to serverless functions and cloud resources in real time as team members onboard, change roles, or offboard—thereby enforcing least privilege and reducing the risk of unauthorized invocations. By integrating with your existing IAM and [DevOps](#) pipelines, Elevate streamlines access reviews, revokes unused service accounts promptly, and offers clear visibility into who can reach which serverless components. Ready to close access gaps and reinforce your serverless security posture? Schedule a Fidelis Elevate demo today to see how automated access governance can keep pace with your agility.

Give Us 10 Minutes - We'll Show You the Future of Security

*See why security teams trust Fidelis to:*

- Cut threat detection time by 9x
- Simplify security operations
- Provide unmatched visibility and control

[Book a Demo Now!](#)