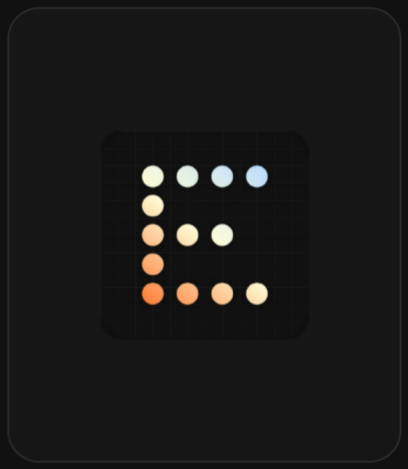
RUNTIME ENFORCEMENT FOR CI/CD

# In The Kernel, Where Code Can't Hide

## A Runtime Enforcement Architecture for CI/CD

Scanners list what your build declares.
Provenance attests to where it came from.
Endura controls what it does.

This paper covers the threat model, the enforcement mechanism in the Linux kernel, the overhead, and how it behaves when a component fails.



WP-001 Rev A / 2026-10

For security architects and platform engineers

© 2026 Infrared Security, LLC. d/b/a Endura Security

▸ CONTENTS_

# Table of Contents


| | | |
|---|---|---|
| 01 | What This Paper Covers<br>ABSTRACT_ | 3 |
| 02 | The Build Is The Breach<br>THE PROBLEM_ | 4 |
| 03 | What Endura Defends, And What It Does Not<br>THREAT MODEL_ | 5 |
| 04 | Two Components, One Loop<br>ARCHITECTURE_ | 6 |
| 05 | Enforcement At The Kernel Boundary<br>ENGINEERED FOR ENFORCEMENT_ | 7 |
| 06 | Policy Derived From Real Behavior<br>POLICY LIFECYCLE_ | 9 |
| 07 | What It Costs To Run In The Kernel<br>OVERHEAD_ | 10 |
| 08 | Failure Behavior<br>FAILURE MODES_ | 11 |
| 09 | Necessary, But Not Sufficient<br>THE CONTROL STACK_ | 12 |
| 10 | Secure The Build. Show The Evidence.<br>COMPLIANCE_ | 13 |
| 11 | Runs Where Your Builds Already Run<br>COMPATIBILITY_ | 15 |
| 12 | Selected Policy Primitives<br>APPENDIX_ | 16 |

▸ ABSTRACT_

## What This Paper Covers

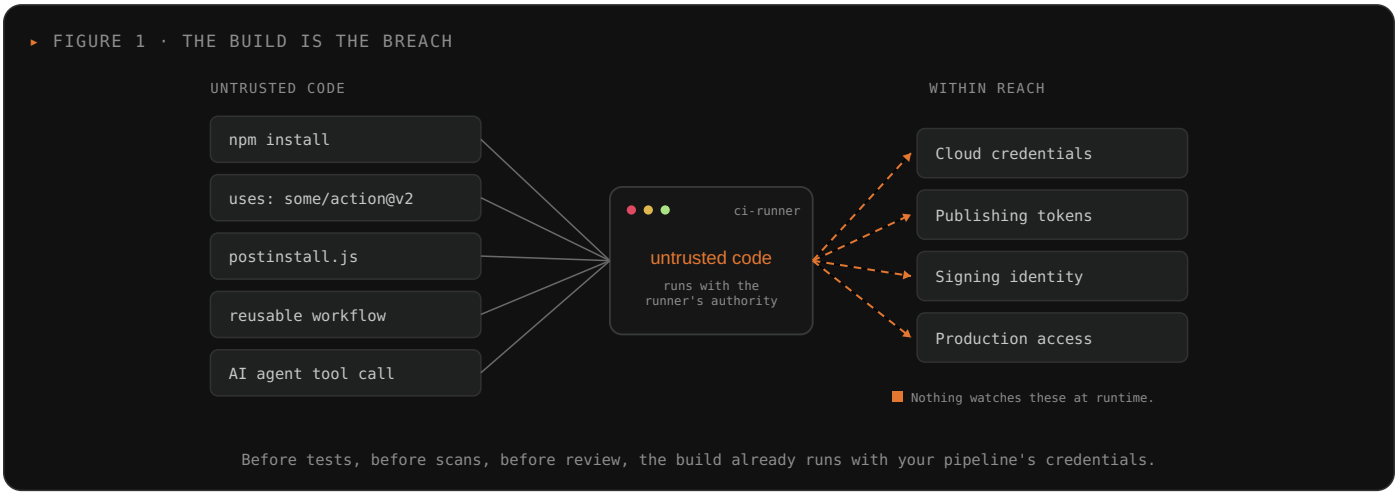
Modern CI/CD pipelines run code that no one on your team fully reviewed. Dependencies, third-party actions, reusable workflows, and AI agents all execute inside the build, and they execute the moment they are pulled in, before tests, before scans, before review.

That code runs with the authority of the runner: cloud credentials, publishing tokens, signing identity, and production access. Endura sets a boundary on what each build is allowed to do and enforces it in the Linux kernel. Runtime Sensor attaches eBPF programs to Linux Security Module (LSM) hooks, evaluates security-relevant operations before they complete, and in enforce mode denies the covered operations that fall outside the policy. The policy is not written from a template. It is derived from what your pipelines actually do, refined into least privilege, validated in `observe` mode, then enforced.

The sections that follow describe the enforcement mechanism, the threat model and its explicit limits, the performance overhead, and the failure behavior. Endura denies the out-of-policy operations it covers, in `enforce` mode, before they complete. It does not stop an upstream maintainer account from being compromised, and where eBPF LSM is unavailable it falls back to monitoring and terminating the offending process after detection.

# The Build Is The Breach

Every control a mature team runs on its supply chain answers a question about identity or origin. None of them governs what the code does once the pipeline runs it.




Software composition analysis (SCA) lists the dependencies a build declares and flags the ones with known vulnerabilities. Static analysis (SAST) finds flaws in first-party source. An SBOM inventories what shipped. Signing and SLSA provenance attest to which workflow produced an artifact and that it was not altered in transit. Each of these is necessary. Each answers a question that was worth asking. The question left open is what the code is permitted to do while the build is running, with the build's credentials, on the build's host.

That gap is where the largest attacks of 2026 landed. In the May 2026 TanStack compromise, attackers published 84 malicious versions across 42 npm packages through the project's own release pipeline, and those releases carried valid SLSA provenance, the first documented case of malicious npm packages shipping with valid attestation. The identity was real, the workflow was trusted, and the attestation verified. The payload still ran, because attestation records where code came from, not what it is allowed to do. The same pattern recurs across ecosystems and years. Three build-time compromises show its shape:

- `XZ Utils, 2024.` A patched `build-to-host.m4` spawned a decoding pipeline during `./configure`, read shell stages hidden in test fixtures, and linked a prebuilt object into `liblzma.so.5`. The backdoor lived in the build of a C library, not in its reviewed source.
- `The Shai-Hulud npm worm, September 2025.` A trojanized package ran a `preinstall` script during `npm install`, swept the filesystem and environment for npm, GitHub, AWS, and Vault credentials, pushed the tokens to an attacker-controlled repository, and republished itself across more packages.
- `A re-enabled action, 2026.` Repositories disabled in May 2026 were re-enabled in September with the malicious code still in place. Any workflow pinned to a moving version tag ran it on the next build. There was no new version, no CVE, and no changed digest.

Sources: XZ Utils (CVE-2024-3094, disclosed March 2024); the Shai-Hulud npm worm (CISA, September 2025); the re-enabled actions-cool GitHub Actions and the TanStack provenance compromise (Socket, StepSecurity, and Palo Alto Unit 42, 2026).

In every case the compromise executed inside a build, with the build's authority, before any scanner reached a verdict. A control that reads manifests or attestations cannot answer this class of attack, because the artifact under attack is behaving exactly as a legitimate build step would until the moment it reads a credential or opens a socket.

# What Endura Defends, And What It Does Not

Two questions decide whether a control belongs in the critical path: what it protects, and what it assumes. Stating the limits plainly is part of the design.

## Assets under protection

- CI/CD credentials: cloud roles, publishing tokens, signing identity, and production access the runner can reach.
- Source and build artifacts produced on the host.
- Runner compute and the trust other systems place in its output.

## Entry points in scope

- A malicious dependency with an install hook.
- A compromised or re-enabled third-party action.
- A poisoned reusable workflow.
- A hijacked maintainer release.
- An over-permissioned AI agent tool call.

## Trust boundaries and assumptions

Endura trusts the integrity of the Linux kernel, root on the build host, Team Server, and the operators who run them. It treats all build code, and everything that build code invokes, as untrusted. Runtime Sensor starts and stops as a root operation while build steps run unprivileged, so a running build cannot disable the control that governs it.

## In scope, denied in `enforce` mode

Unauthorized file reads, writes, and execution. Outbound connections to hosts outside policy. Process injection and tracing. Direct access to another process's memory. Privilege escalation through kernel capabilities. Loading of kernel modules and eBPF programs. Container execution and runtime socket access. Persistence through startup files and scheduled tasks.

## Explicit non-goals

Endura limits what build code can do when it runs in a protected environment. It does not prevent an upstream maintainer account from being compromised, does not stop malicious publishing at the source, and does not fix the vulnerable code itself. For an attributed job, a covered operation with no matching rule is a violation and is denied. Coverage depends on the Sensor being attached to the relevant hook, so unattributed operations, and operations without an attached supported hook, fall outside enforcement. Compromise of the kernel or of root on the host is outside the model. Endura complements SCA, SAST, SBOMs, provenance, signing, and dependency cooldowns. It does not replace them.

## Residual risk

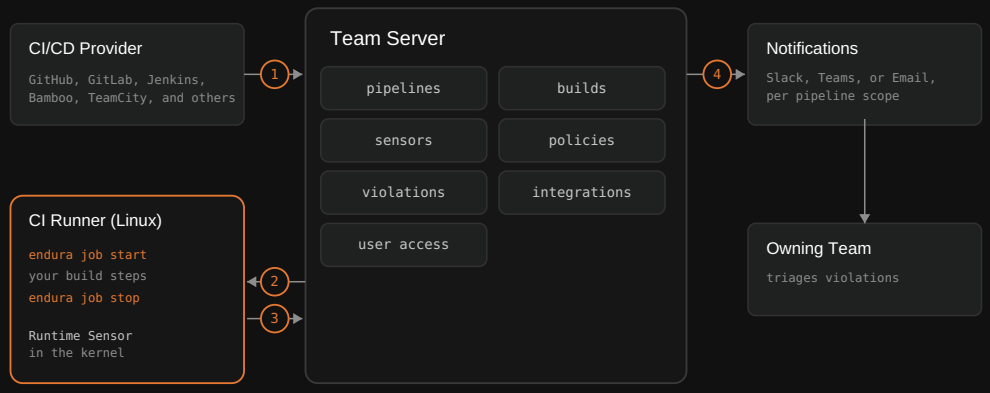
Where eBPF LSM is unavailable, Endura falls back to monitoring and terminating the offending process after detection, which is a weaker guarantee than inline denial and carries a race window. Policy derived from observed behavior cannot capture a valid branch that did not run during derivation, and over-broad refinement widens the boundary. These are discussed in the policy and failure sections below.

▸ ARCHITECTURE_

# Two Components, One Loop

Runtime Sensor enforces on the runner. Team Server holds policy, evidence, and routing. Team Server never connects into a build host; the Sensor initiates every Sensor-to-Team-Server request.

▸ FIGURE 2 · TWO COMPONENTS, ONE LOOP




1 Team Server finds pipelines. 2 Sensor fetches its policy. 3 Sensor reports violations. 4 Alerts post to the matching Slack, Teams, or email destinations.

`Runtime Sensor` runs on the build host as the enforcement point, one Sensor per host or node. It loads eBPF programs, evaluates operations before they complete, and depending on mode derives rules, records violations, or blocks unauthorized behavior. It installs as a native package, as a Docker or Podman container, or as a Kubernetes DaemonSet.

`Team Server` is the management plane. It discovers CI/CD pipelines through provider integrations, distributes policy to Sensors, records violations with their context, and controls access through configured OIDC providers or optional local credentials, with personal access tokens for API clients, and applies Viewer, Operator, and Administrator roles and resource scope. It runs managed by Endura or self-hosted on Docker, Podman, or Kubernetes, backed by PostgreSQL. Scheduled jobs handle statistics, reconciliation, and historical-data pruning.

The two components form one loop. Team Server discovers pipelines. Runtime Sensor fetches the policy for the job it is protecting. The Sensor reports violations and runtime context back. Violations are grouped by build or workload and delivered through Slack, Microsoft Teams, or email to the destinations configured for the matching pipeline or deployment.

## Connectivity

Sensor traffic is outbound to Team Server over HTTP(S); production deployments should expose Team Server over HTTPS, through ingress TLS or its built-in TLS. Team Server never opens a connection into a build host, so the Sensor adds no inbound attack surface. Certificate verification is on by default for HTTPS, with optional pinning.

## Operating modes

In standalone mode the Sensor runs with only a license key set, and policies and results stay on the host; no data is sent to Team Server, though resolving hostname policies may make outbound DNS or RDAP lookups. With Team Server configured, the Sensor adds a Team Server URL and a sensor token, and Team Server then distributes policy and receives status, logs, and violations.

## What leaves the host

When Team Server is used, the Sensor sends registration and status, host and build identity, build heartbeats, violations, and logs, with deployments and long-running workloads reporting their own status. Standalone mode sends none of it to Team Server.

# Enforcement At The Kernel Boundary

Runtime Sensor runs as eBPF below the build and everything it invokes. File, network, process, and memory operations pass through the kernel to happen at all, so that is where policy is enforced.

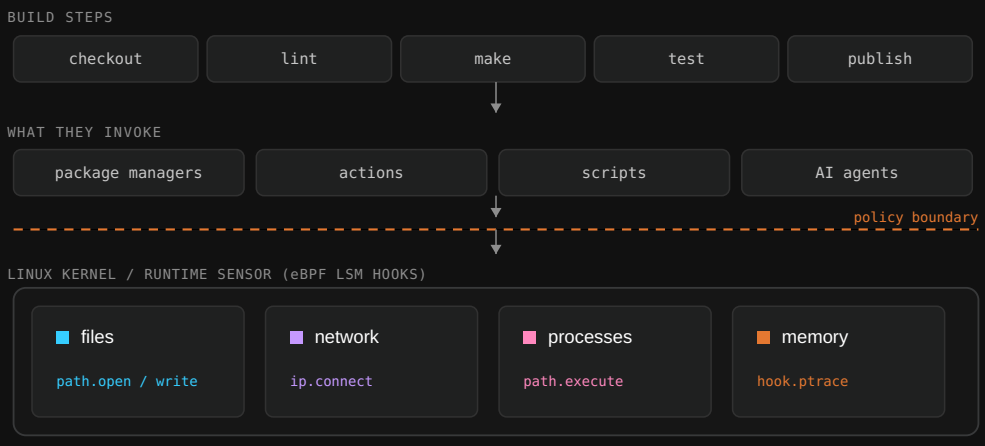
| 50+ | 40+ | <5% | 20+ |
|---|---|---|---|
| kernel-level security hooks | threat techniques prevented | typical build overhead | supported Linux distributions |

## The LSM mechanism

The Linux Security Module framework defines decision points inside the kernel where access control is applied: the places where a program is executed, a file is opened, a socket connects, or one process reaches into another's memory. Since Linux 5.7, the BPF LSM facility lets an eBPF program attach to those hook points and return a binding decision; Endura requires Linux 5.11 or later. For path and file hooks, the hook fires after the kernel has resolved the path and the inode, so it sees the real object rather than a user-supplied argument an attacker can swap after a check. Socket hooks act on the connection's address, and task-control hooks on the target task, each at the kernel's decision point for that operation. The return value is enforcement, not a log line: for an attributed job, a covered operation is permitted, or it is denied in the kernel before it completes and, by default, the offending process is also signaled with SIGKILL.

This is why enforcement lives in the kernel rather than in a userspace wrapper, a syscall shim, or an `LD_PRELOAD` library. Unprivileged build steps cannot bypass it, it observes every child process, interpreter, and tool the build invokes, and it needs no cooperation from the workload. Where BPF LSM is unavailable, a fentry probe cannot refuse a completed operation; it terminates the offending process when kill-task behavior is enabled, which is the default. That fallback is detection plus termination rather than inline denial, a weaker guarantee with a race window, and the kill behavior is configurable.

▸ FIGURE 3 · THE POLICY BOUNDARY




File, network, process, and memory operations pass through the kernel to happen at all, so that is where policy is enforced.

## What Endura mediates

| OPERATION CLASS | EXAMPLES | POLICY PRIMITIVES |
|---|---|---|
| File access | opens, reads, writes, creation, deletion, mounts | `path.open, path.create, path.write, path.delete, path.execute, path.mount, and more` |
| Networking | IP, Unix, packet, raw, netlink, and vsock operations | `ip.connect, and related` |
| Process control | signals, scheduling, resource limits, tracing | `hook.ptrace, hook.traceme, and related` |
| Kernel and memory | eBPF, modules, IOCTL, W+X, memory mapping | `hook.wx, mmap.file, and related` |
| Containers | image execution and runtime socket access | `see specification` |

Primitives shown are representative, not exhaustive; the policy model also covers process control, containers, and IPC. Full specification: docs.endurasecurity.com/security-policies/specification.
50+ hooks and 40+ techniques derived from the Runtime Sensor source.

## How a rule is evaluated

In `enforce` mode, a covered operation by an attributed job without a matching effective rule is a violation and is denied. Omitted sections inherit the Sensor's built-in baseline, so a mode-only policy is not an empty allowlist. A rule carries one to three operation-specific fields, so the arity and meaning of each field depend on the operation, and a whole-rule `all` is also supported. For example, the `ip.connect` rule `/usr/bin/curl|api.example.com|{80,443}`, read as executable, destination, and port, permits `curl` to reach that host on port 80 or 443. Path arguments use directory-prefix matching, except for identity-style fields defined in the specification. Rules can use the `all` wildcard, brace expansion such as `{a,b}`, identity variables including `%workspace%`, `%home%`, `%hostname%`, `%uid%`, and `%user%`, and `<anonymous>` for anonymous or memfd-backed execution where supported. Team Server also supports reusable system and policy variables so a shared value can be defined once and overridden per policy.

## How a job is attributed

One Sensor protects every job on the host at once, each under its own policy. A job is anchored at start by its workspace path and its mount namespace, and is selected afterward by a 16-digit hex identifier or by workspace path, with the identifier resolving across mount namespaces. The Sensor authorizes a job by resolving the calling process through its peer credential and the caller's procfs root, so a non-root user may start a job only for a workspace they own. Root bypasses ownership, not existence. Containerized Sensors run with the host PID namespace and the host procfs path and mount the bpf and kernel filesystems, which is how attribution holds across container boundaries.

# Policy Derived From Real Behavior

Endura recommends deriving policy from real behavior and validating it before anything is denied, though a policy can be assigned in any mode. Observed does not mean approved.

▸ FIGURE 4 · DERIVE, REFINE, OBSERVE, ENFORCE




Policy is derived from real behavior and validated before anything is denied. Observed does not mean approved.

The lifecycle has four stages. In `derive`, the Sensor records the covered file, execution, network, and other kernel operations a representative build performs, including accesses to credential files, then emits a policy that adds observed rules, removes unobserved baseline entries, and simplifies the result. Derivation records observed behavior, not intended behavior: it cannot capture a valid branch that did not run, and it can capture one-time setup or unwanted behavior that did.

In `refine`, that record becomes an intentional boundary. Remove one-time operations. Replace dynamic filenames with the narrowest stable parent directory. Consolidate equivalent values with brace expansion. Broaden an address only after confirming it. Derivation prefers an observed hostname and its reverse resolution, and falls back to `/32` or `/128` when no name is available; a hostname rule itself resolves at load into one or more address ranges, which can be wider than a single host. Add legitimate branches that were not exercised. Validation caps how large a single expression may grow, so refinement also keeps a policy within those limits.

In `observe`, the refined policy runs without blocking and records violations for review. In `enforce`, only the mode changes, and the Sensor denies covered operations outside the boundary. Policy is versioned JSON, assigned per pipeline, and reviewed like any other code. Precedence is the local policy file, then the built-in derive policy, and Team Server can replace either once the provider and pipeline match.

policy.json

```
{
  "mode": "enforce",
  "ip": {
    "connect": [
      "/usr/bin/curl|api.example.com|{80,443}",
      "all|140.82.112.0/20|443"
    ]
  }
}
```

This `ip.connect` example reads executable | destination | port. Covered operations that are not listed are denied. Field arity is operation-specific; see the specification for the full schema.

# What It Costs To Run In The Kernel

A control in the critical path has to be cheap. Several design choices keep the added cost small and bounded.

Evaluation happens at security-relevant LSM hooks rather than on every syscall, so most activity is never intercepted. A policy decision is a lookup against rules already loaded in the kernel, which keeps the per-operation cost small. Telemetry to Team Server is batched and sent asynchronously, so enforcement is never blocked waiting on the network. By default the Sensor is sized at 1 GiB of memory with a 2 GiB limit, and event delivery uses a configured 128 MiB of ring buffers (four 32 MiB shards).

The figure Endura publishes, under 5% typical build overhead with enforcement active, holds for common build profiles. A credible number depends on its conditions, so the benchmark that supports it reports the workload profile, host specification, count of active hooks, added latency at the median and the 99th percentile per mediated operation, and the Sensor's memory footprint. Syscall-heavy builds carry more hook evaluations than compute-bound ones, and a very large policy is bounded by validation limits rather than by runtime throughput.

## Measured overhead

| WORKLOAD PROFILE | HOST | ACTIVE HOOKS | ADDED LATENCY P50 / P99 | SENSOR MEMORY |
|---|---|---|---|---|
| Compute-bound | i9-9900KF, 8C/16T, 31.3 GiB, Linux 7.2.5 | 75 / 75 | 0.58 µs / 0.62 µs | 379 MB RSS |
| Syscall-heavy | i9-9900KF, 8C/16T, 31.3 GiB, Linux 7.2.5 | 75 / 75 | 1.88 µs / 3.23 µs | 379 MB RSS |

Measured with Endura's benchmark suite across compute-bound and syscall-heavy build profiles, all 75 hooks attached. Added latency is the per-operation cost at the median and the 99th percentile; Sensor memory is resident set size.

# Failure Behavior

Endura is not a kill switch for your pipeline. Running builds are never killed wholesale. The strongest thing a build can trigger is denial of its own out-of-policy operation, which by default also terminates the offending process. When the Sensor cannot enforce a given operation, it fails open rather than stop your build.

| SCENARIO | BEHAVIOR |
|---|---|
| An operator stops the Sensor | Running builds are not killed. Enforcement ends for the remainder of those builds, and `endura job stop` then exits non-zero, so the pipeline surfaces it rather than finishing quietly. |
| A hook fails to load or attach | Nonfatal. The Sensor keeps running, and any operation that depends on the unattached hook is not enforced, which fails open rather than stopping the build. |
| A build tries to stop the Sensor | Starting and stopping the Sensor are root operations while build steps run unprivileged, so nothing inside a running build can stop it. |
| Team Server is unreachable | Local enforcement continues on the job's selected policy. Heartbeats and telemetry resume when connectivity returns, but failed batches are not durably queued or replayed and may be lost. |
| Config or TLS tampering | The environment file is root-owned, so an unprivileged build cannot alter it. With certificate pinning enabled, a missing or empty CA file fails closed rather than falling back to an unverified connection. |
| Sensor and Team Server version mismatch | The two compare major versions. Matching majors are compatible, and minor or patch differences are allowed. On a major mismatch the Sensor keeps enforcing locally but suppresses normal Team Server submissions, rechecks every 15 seconds, and keeps posting its version; Team Server reports the Sensor as incompatible. |
| eBPF LSM unavailable | A fentry fallback cannot refuse a completed operation. It terminates the offending process when kill-task behavior is enabled, the default, so this path is detection plus termination rather than inline denial. |
| Build cancellation | When a process outside every protected job sends SIGINT, SIGTERM, or SIGKILL to a protected CI build recognized from provider signals, the Sensor allows the signal, records the cancellation, and keeps every other control active, then retires the job after a configurable timeout that defaults to 15 minutes. |
| Delayed or out-of-order telemetry | Graceful shutdown drains in-memory batches on a best-effort basis. Event and export queues are bounded and drop under overload, so abrupt termination or a failed submission can lose undelivered evidence. |

# Necessary, But Not Sufficient

The controls below are not wrong. Each answers a real question. None was designed to govern what code does at runtime, inside the build. Endura adds that boundary and complements the rest of the stack.

| CATEGORY AND EXAMPLES | QUESTION IT ANSWERS | WHEN IT ACTS | POSTURE | NEEDS PRIOR KNOWLEDGE OF THE THREAT | GOVERNS BUILD RUNTIME BEHAVIOR |
|---|---|---|---|---|---|
| SCA<br>Snyk, Mend, Black Duck | Which declared dependencies have known vulnerabilities | Before the build, in CI | Detect and advise | Yes, vulnerability databases | No |
| SAST<br>Semgrep, Checkmarx, SonarQube | Where first-party source has flaws | Before the build | Detect and advise | Yes, rules and patterns | No |
| SBOM and provenance<br>Syft, Anchore, FOSSA; Sigstore, SLSA, Chainguard | What is in the artifact and where it came from | At build and publish | Inventory and attest | Not applicable | No |
| Package behavior screening<br>Socket.dev | Whether a package looks suspicious before install | Before install | Detect, advise, gate installs | Heuristics on package metadata and capabilities | No, gates installs rather than governing runtime behavior |
| CI/CD runner hardening<br>StepSecurity, Harden-Runner | What the runner reaches on the network, with tamper detection | During the build | Prevent egress, detect broader activity | No, egress allowlist | Partial, network-centric |
| Pipeline posture and ASPM<br>Legit Security, Cycode, Arnica | Where the pipeline is misconfigured or risky | Continuous, before the build | Detect and advise | Configuration analysis | No |
| Production runtime security<br>Falco, Tetragon, KubeArmor, Aqua, Sysdig | What production workloads do at runtime | In production | Mostly detect, some enforce | No, behavior-based | Governs production, not the build |
| Endura<br>Runtime Sensor, Team Server | What the build is allowed to do | During the build, in the kernel | Prevent, deny-by-default, inline | No, behavior-based, no signature required | Yes, for attributed jobs, across the file, network, process, memory, kernel, and container operations its hooks cover |

The nearest neighbor is CI/CD runner hardening, which also uses eBPF at build time and is strongest at controlling network egress and detecting tampering. Endura differs in that it enforces deny-by-default across the file, network, process, memory, and kernel operations its hooks cover, rather than mainly the network, derives policy per pipeline from observed behavior, runs across many CI providers, and can be fully self-hosted. Production runtime tools act in the workload after release and are mostly detection; Endura acts in the build and denies inline.

▸ COMPLIANCE_

# Secure The Build. Show The Evidence.

Most security frameworks ask two things of your pipeline: lock down what runs in the build, and prove it stayed locked down. Scanners tell you what is in a build. Provenance tells you where it came from. Endura governs what the build is allowed to do, and records what it actually did, which is the part auditors keep asking for.

Endura is a runtime enforcement layer. It does not generate provenance, sign artifacts, produce an SBOM, or scan source. It works alongside the tools that do. The mappings below are deliberate about where Endura is a primary control, where it supports a requirement another control owns, and where it produces evidence you can show an assessor.

■ DIRECT Endura is a primary control for the requirement. ◧ SUPPORTING Endura strengthens a requirement owned mainly by another control. □ EVIDENCE Endura produces artifacts that help you demonstrate the requirement.

## NIST SSDF and SLSA

The Secure Software Development Framework (NIST SP 800-218) is the reference most US federal and private programs measure against, and SLSA defines the build-integrity levels teams target. Endura's strongest fit is the secure-environment practice and the hardened-build intent.

| FRAMEWORK | REQUIREMENT | ROLE | HOW ENDURA APPLIES |
|---|---|---|---|
| NIST SSDF<br>SP 800-218 | PO.5<br>Secure environments | ■ DIRECT | Runtime Sensor enforces least privilege on the build host: which files a build reads and writes (`path.open`, `path.write`), which binaries it executes (`path.execute`), and which destinations it reaches (`ip.connect`). PO.5.1 and PO.5.2 call to separate, harden, and monitor build environments; Endura does this in the kernel, and `observe` mode records activity without blocking. |
| | PS.1<br>Protect code | ◧ SUPPORTING | File-access policy during the build limits unauthorized changes to source and artifacts while the build is handling them. |
| | PO.3<br>Supporting toolchains | ◧ SUPPORTING | Endura secures the environment the toolchain runs in and constrains what each tool in the chain is allowed to do. |
| | RV.1<br>Identify vulnerabilities | ◧ SUPPORTING | Policy violations surface runtime behavior a build was not expected to produce, which feeds detection and triage. This is not vulnerability scanning. |
| SLSA<br>v1.0 Build L3 | Tamper-resistant build | ◧ SUPPORTING | Build L3 exists to protect a build from tampering while it runs. A step that reaches for `~/.aws`, opens an unexpected socket, or executes an unplanned binary is denied before it completes. Endura hardens the environment the build runs in and complements the provenance the platform generates. |

Endura does not produce or sign provenance, so it does not by itself raise your SLSA level. Code review (PW.7), testing (PW.8), and release signing (PS.2) sit outside its scope; your SAST, DAST, and signing tools own those.

> Federal note: in January 2026, OMB memorandum M-26-05 rescinded the government-wide secure-software attestation mandate (M-22-18 and M-23-16) and the CISA Common Form, moving agencies to their own risk-based approaches. SSDF remains a widely used reference, and agencies may still request a self-attestation or an SBOM. Endura's PO.5 enforcement and its runtime records support those requests where they apply. Confirm current federal requirements before relying on them.

▸ COMPLIANCE_

# Crosswalks And The Evidence Auditors Ask For

Because Endura is a kernel-level control that keeps its own records, it also crosswalks to the general catalogs your auditors use. These are the controls where enforcement, or enforcement paired with violation logging, carries weight.

| FRAMEWORK | CONTROLS | ROLE | WHAT IT COVERS |
|---|---|---|---|
| NIST<br>SP 800-53 | SI-7, CM-7 | ■ DIRECT | Software and information integrity, and least functionality. |
| | SI-4, AC-6, CM-5 | ◧ SUPPORTING | System monitoring, least privilege, and access restrictions for change. |
| SOC 2<br>Trust Services | CC6, CC7, CC8 | □ EVIDENCE | Enforcement paired with violation logging produces evidence for logical access, system operations and monitoring, and change management. |

## The evidence auditors ask for

A framework is easier to pass when you can show the control working, not just describe it. Endura produces three artifacts an assessor can review.

01 / ACTIVITY

### Per-build runtime record

Every file, process, and network destination each build touched, per pipeline and per build.

02 / POLICY

### Least privilege as code

The boundary each build runs under, expressed as policy you can review, diff, and version.

03 / VIOLATIONS

### Violations with ancestry

A record of anything that fell outside policy, with the full process ancestry behind it.

Deploy Runtime Sensor on one pipeline in `derive` mode, and Endura will walk the output against the framework you answer to.

▸ COMPATIBILITY_

# Runs Where Your Builds Already Run

On supported x86_64 Linux today, with ARM64 on the roadmap, as SaaS or fully self-hosted.

Three capabilities are separate. Runtime Sensor enforces policy on a supported x86_64 Linux host running kernel 5.11 or later, in standalone mode with no Team Server, so enforcement does not depend on the CI provider. A Sensor adapter adds automatic CI context for a detected build. Team Server discovery adds automatic inventory and per-pipeline policy management. Each covers a different set of providers.

| PROVIDER COVERAGE | PROVIDERS |
|---|---|
| Discovered by Team Server and Sensor-adapted | Bamboo, GitHub, GitLab, Jenkins, TeamCity |
| Sensor-adapted, not Team Server-discovered | AWS CodeBuild, Azure Pipelines, CircleCI, Argo Workflows, Travis CI, CloudBees |
| Discovered, no Sensor adapter | Koji |

Discovery enables per-pipeline management; a Sensor adapter supplies automatic CI context. Where neither applies, a build can still be enforced as a generic workload, since enforcement does not depend on the provider.

## Platforms and requirements

### OPERATING SYSTEMS

- AlmaLinux 9, 10
- Alpine Linux 3
- Amazon Linux 2023
- Arch Linux
- CentOS Stream 10
- Debian 12, 13
- Fedora 44
- Gentoo
- Oracle Linux 8 UEK, 9, 10
- RHEL 9, 10
- Rocky Linux 9, 10
- SUSE Enterprise 15, 16
- Ubuntu 24, 25, 26

### HOST REQUIREMENTS

- x86_64 architecture
- Linux kernel 5.11+ with BPF LSM
- BTF at /sys/kernel/btf/vmlinux
- Root and required capabilities
- 1 GiB memory or more
- Outbound HTTP(S) to Team Server

### INSTALL

- Native RPM, DEB, TGZ
- Docker or Podman
- Kubernetes DaemonSet, Helm
- One Sensor per host or node
- SaaS or self-hosted

### AUTH AND CONFIG

- Root-owned environment file
- OIDC or local login
- PAT for API
- RBAC controls

# Selected Policy Primitives

> These are common examples, not the full set. Runtime Sensor mediates operations across files, networking, process control, memory, kernel, containers, and IPC. The complete, authoritative policy specification is maintained at `docs.endurasecurity.com/security-policies/specification`.

| PRIMITIVE | GOVERNS | RULE SHAPE |
|---|---|---|
| `path.open` | File opens (not filtered by mode), such as `~/.ssh`, `~/.aws`, or `~/.npmrc` | `process_path\|target` |
| `path.create / path.write` | File creation and modification | `process_path\|target` |
| `path.execute` | Execution chain by parent, interpreter, and target | `parent\|interpreter\|target` |
| `ip.connect` | Outbound connections by program, host or CIDR, and port | `process_path\|target\|qualifier` |
| `hook.ptrace` | Tracing of, or memory access to, another process | `tracer_path\|target_path` |
| *More operations* | *additional file, network, process, memory, kernel, container, and IPC operations not listed here* | *see full specification* |

Values include the `all` wildcard, `{a,b}` brace expansion, and identity variables such as `%workspace%` and `%user%`. Field meaning is operation-specific; see the full specification for every operation and its fields. Related primitives include `hook.wx` and `mmap.file`.

## Local inspection on the build host

| COMMAND | WHO CAN RUN IT |
|---|---|
| `endura sensor status, wait` | Any local user |
| `endura sensor start, stop, logs, hooks` | Root |
| `endura job start` | Workspace owner or root |
| `endura job get, list, status, stop; endura policy describe` | Job owner or root |
| `endura policy validate` | Any user with file read |

`endura sensor hooks` lists every loaded eBPF program and whether it attached. An unattached row means that specific program or hook path is unavailable; the event class may still have partial coverage through other rows.



▸ READY WHEN YOU ARE_

# See What Your Builds Are Doing

Deploy Runtime Sensor on one pipeline, run it in derive mode to record every file, connection, and process your build touches, then refine, observe, and enforce.

REQUEST A DEMO

endurasecurity.com

© 2026 Infrared Security, LLC. d/b/a Endura Security · 18376 Southampton Drive, Lewes, DE 19958.