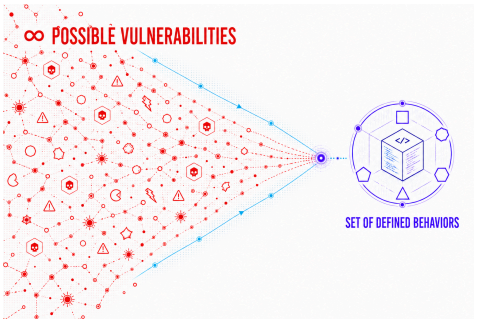
A DETERMINISTIC APPROACH TO SOFTWARE SUPPLY CHAIN SECURITY

Your application is assembled largely from code you did not write, some of it arriving at runtime. Every control that has to know the threat first leaves that exposed.

praedictio.ai · September 2026

A modern application is a composite: first-party code, open-source libraries, dynamically loaded modules and third-party services, all executing in the same runtime context. A compromise at any one of those layers becomes operational risk in production.



Threats are unbounded. Behavior is not. The set of possible vulnerabilities has no limit. The set of things your application is supposed to do is finite, and you already know it.

The supply chain widens the surface in ways scanning does not reach. Deliberately malicious code, AI-assisted development errors, opaque transitive dependencies and human misconfiguration can each introduce behavior nobody intended and nobody documented.

Most supply chain security remains anchored to scanning for known vulnerabilities and enumerated threat vectors. That work is necessary, and it is structurally limited: it assumes risk can be anticipated from prior knowledge. Zero-day exposure is not an anomaly in that framework. It is a predictable consequence of it.

THREAT	HOW IT GETS PAST TRADITIONAL CONTROLS
Supply chain compromise	Malicious code is embedded in source or metadata, or fetched and executed at runtime.
AI-generated zero-day exploits	No prior CVE or signature exists, so detection models have no reference pattern.
Insider-driven backdoors	Payloads sit in rarely exercised or obfuscated code paths that static analysis may not understand.
Dynamic loading and de-obfuscation	Code is downloaded or decrypted at runtime, after the static SBOM was written.

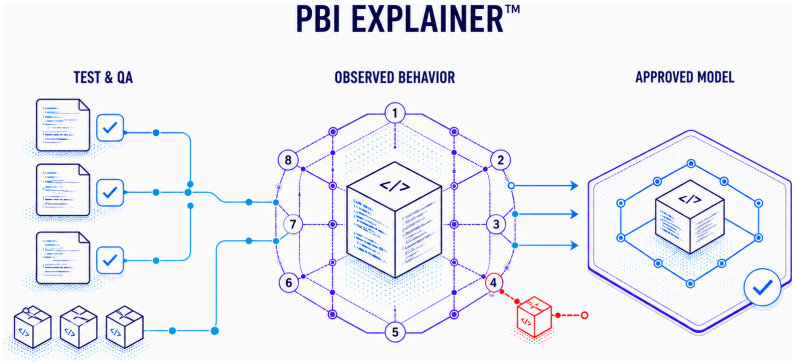
From enumeration to governance

Program Behavior Intelligence™ (PBI™) answers a different question. Rather than anticipating what may be malicious, it builds a deterministic model of how the program behaves during legitimate execution. Every file operation, network request and library invocation is captured together with the causal chain that produced it, which makes the model an auditable account of what the application actually does.

PBI Explainer™ builds that model and renders it for people and AI agents to read, so teams can interpret execution patterns, confirm expected behavior and find what they did not expect. **PBI Enforcement™** applies the same model in production, where anything outside it is prevented from executing, whatever its origin and however novel.

How the model is built

Explainer establishes the baseline during a controlled execution phase: the program runs against its full automated test suite together with a representative workload. Runtime activity is consolidated into an auditable model tied to that specific build, which can then be explored, rendered and compared across versions before anything is promoted to production.



What Explainer sees. Your tests drive the application, Explainer records what actually runs, and that observed behavior becomes the approved model.

Where teams use it

Onboarding new components. Documented functionality is not runtime behavior. Run the program with the new dependency in place and the model shows what it does. A hidden payload appears as a system-call sequence that is in no upstream documentation, found before the build is promoted.

Library upgrades and patch validation. An upgrade meant to remediate a CVE brings new code with it, and review processes tend to look at version numbers and patch notes. Comparing models across versions makes added, removed and altered execution paths explicit: new network connections, file writes,

spawned processes. That is what stops a compromised fork or a patched release carrying something extra.

A behavior-based SBOM. A conventional SBOM enumerates what shipped. A **PBI SBOM™** records what executed, including dynamically loaded components, so you can show not only what is in a release but what it did. Dormant paths can be pruned rather than shipped.

Test coverage. Compare execution driven by your formal test suite against behavior observed during controlled operational use. The difference is your coverage gap, and each newly observed path converts into a test case. Over time the baseline becomes a record of validated functionality rather than a static artifact.

AI-generated code. Generative AI produces code through stochastic processes, so provenance and architectural intent cannot be recovered from the artifact. PBI does not try to predict whether that code is malicious. It establishes whether it behaves within defined expectations, and enumerates the ways in which it does not.

Redundant and obsolete code. Unused functions and legacy integrations remain executable, and therefore exploitable. Making real execution explicit lets teams retire them, which cuts attack surface, lowers runtime overhead and simplifies what an auditor has to read.

What changes

WHERE	TODAY	WITH PBI
Supply chain malware	Found weeks or months later, after the incident	Prevented at the point of execution
False positives	5 to 15 percent with static composition analysis	A flagged behavior means missing test coverage, not noise
Transitive dependencies	70 to 80 percent, depending on SBOM quality	Everything that actually executes
A new automated test	Hours or days of manual analysis	The missing case is named for you
Compliance evidence	Manual mapping of code to controls	Exported from the model, traceable to each control

> **ONE HONEST LIMIT**
>
> PBI describes what ran, so the model is only as complete as the execution you give it. Where test coverage is thin the picture is thin. The model names that gap rather than papering over it, which is why a flagged behavior is worth reading as a coverage question first.

What it adds up to

The software supply chain has become a dynamic execution environment rather than a dependency inventory. Static SBOMs, vulnerability scanners and signature-based detection remain necessary and are not sufficient on their own, because malicious or unintended behavior can arrive through a third-party component, dynamic loading, AI-assisted development or insider modification, and may surface only under specific runtime conditions.

A behavioral model changes what you can govern. It gives visibility into runtime behavior without depending on source code analysis, shows execution changes introduced by routine updates, and surfaces code paths nobody had looked at. Built into development and release workflows, it becomes a continuous control rather than a point-in-time assessment.

Risk you cannot enumerate is still risk you can bound.

UNDERSTAND EXACTLY WHAT YOUR SOFTWARE DOES. ENFORCE THAT AT RUNTIME.

SEE IT ON YOUR SOFTWARE