



MAY 2026

WHITEPAPER

DevSecOps for Secure Mission Environments

Evidence Is the Product

PUBLICATION DETAILS

AUTHORS	Polyrhythm Software, LLC
PUBLISHED	May 18, 2026
UPDATED	May 25, 2026
KEYWORDS	devsecops, software engineering, authorization, supply chain security



ABSTRACT

Secure software in secure mission environments must survive integration, review, authorization, deployment, incident response, and sustainment. This paper frames DevSecOps as the disciplined production of software and the evidence needed to trust it, with an implementation pattern aligned to RMF, cATO, CMMC, and DFARS.

For secure mission environments, DevSecOps has two products: working software and the proof that lets that software stay trusted.

Executive Summary

Secure software does not just need to run. In secure mission environments, it has to survive integration, security review, authorization, deployment, incident response, and long-term sustainment.

That requires evidence.

For commercial software, a passing build may be enough to ship. For secure mission environments, a passing build is only one part of the story. A release also needs to show what changed, why it changed, and who approved it. It needs to show how it was built and what dependencies it carries. It needs to show what tests passed, what vulnerabilities remain, what risks were accepted, where it was deployed, and how it is behaving in operation.

That is the central idea behind this paper:

Software is the mission product. Evidence is the assurance product that allows the software to cross boundaries, support authorization, and remain trusted in operation.

This is especially important in Department of Defense environments. The DoD Software Modernization Implementation Plan continues to emphasize resilient software delivery at the speed of relevance. ([U.S. Department of Defense CIO](#)) DoD DevSecOps guidance describes a software factory as people, tools, and processes that deliver software value through automation. The guidance also points to pipelines, monitoring, artifact repositories, configuration management, and controlled promotion gates. ([U.S. Department of Defense CIO](#))

The practical conclusion is straightforward: DevSecOps is not just CI/CD. In secure mission environments, DevSecOps is the disciplined production of software **and** the evidence needed to trust that software.

The Polyrhythm View

Polyrhythm works where software meets real mission constraints: mission systems, flight test, modeling and simulation, sensor integration, open architectures, and secure delivery environments.



Our open-architecture position is simple: open systems only matter if they integrate cleanly and ship predictably. That means clear interfaces and schema-controlled exchanges. It also means contract tests, reproducible builds, signed artifacts, security scans, and promotion paths that work across enclaves.

The same principle applies to DevSecOps.

A pipeline that produces software but not evidence is incomplete. A dashboard that shows status but not provenance is incomplete. A control mapping that exists outside the engineering workflow is fragile. A release package that depends on manual reconstruction is already late.

For Polyrhythm, the goal is not compliance theater. The goal is an engineering system where the artifacts needed for security, authorization, integration, and sustainment are generated as part of normal work.

Why Secure Mission Environments Are Different

Secure mission environments create conditions that ordinary software delivery models do not handle well.

They may involve multiple authorization boundaries, disconnected networks, CUI, classified enclaves, and export-controlled information. They may also involve lab environments, operational platforms, hardware-in-the-loop testing, and deployment targets that cannot be treated like ordinary cloud services.

DoD guidance recognizes these realities. Software factories may operate at different classification levels. They may run in cloud, on-premises, or disconnected environments. They may also produce software that is deployed outside the factory boundary, such as into a weapon system or another separately authorized production environment. ([U.S. Department of Defense CIO](#))

That means evidence has to support boundary crossing.

When software moves from development to test, every handoff asks the same questions. The same is true when it moves from test to integration, from integration to an operational platform, or from one enclave to another:

- What exactly is this artifact?
- Where did it come from?
- What was it built from?
- What changed since the last approved version?
- What security checks were performed?
- What known risks remain?
- Who accepted those risks?
- What operational constraints apply?



If those questions require manual archaeology, the delivery system is not mature enough.

Evidence Is Not Paperwork

Evidence is often confused with documentation. That is a mistake.

Documentation explains. Evidence proves.

Useful evidence is generated by the engineering process itself. It is structured, attributable, current, and tied to the artifact it supports. It should be good enough for engineers, cyber teams, program leaders, mission owners, incident responders, and authorizing officials. Those groups need to make risk decisions without rebuilding history by hand.

In secure mission environments, evidence should answer four basic questions:

1. **Can we trace the artifact?** Identify the source, configuration, dependencies, build environment, logs, signatures, hashes, and release lineage.
2. **Can we verify the artifact?** Show unit tests, integration tests, interface tests, regression tests, and performance tests. Add hardware-in-the-loop results, software-in-the-loop results, and mission-thread validation where needed.
3. **Can we assess the risk?** Show static analysis, dynamic analysis, dependency scans, container scans, secrets checks, and SBOMs. Include vulnerability triage, POA&M items, waivers, compensating controls, and residual risk decisions.
4. **Can we operate and defend it?** Show deployment records, configuration state, monitoring coverage, alerting, incident response hooks, drift checks, log access, and operational telemetry.

When these answers are generated continuously, evidence becomes part of the product lifecycle instead of an after-the-fact compliance scramble.

The Minimum Viable Evidence Package

A practical evidence-first DevSecOps approach should start with a minimum viable evidence package for every release.

That package should include:

1. Change Rationale

Every release should identify the requirement, defect, vulnerability, operational need, technical debt item, or mission thread that drove the change. This does not need to be heavy. It does need to be explicit.



2. Artifact Identity

Every artifact should have a stable identity: version, hash, build number, source revision, branch or tag, build timestamp, and signing status.

3. Build Provenance

The program should know how the artifact was produced. That includes build environment, compiler or toolchain version, container base image, build script, pipeline run, dependency sources, and generated logs.

4. Dependency and SBOM Data

Modern software depends on third-party packages, containers, libraries, operating system parts, and infrastructure definitions. A Software Bill of Materials should be generated and kept as part of the release record. NIST's Secure Software Development Framework emphasizes practices that reduce vulnerabilities in released software. It also addresses the impact of vulnerabilities that remain and the root causes behind them. ([NIST Computer Security Resource Center](#))

5. Test Results

The evidence package should include automated and manual test results that fit the system. For mission systems, this often means more than unit tests. It may include interface tests, model-based tests, scenario tests, timing tests, hardware-in-the-loop tests, and lab event results.

6. Security Scan Results

Security evidence should include SAST, DAST where applicable, dependency scans, container scans, infrastructure-as-code checks, secrets scans, configuration checks, and vulnerability triage. DoD cATO criteria recognize automated security scanning. They also recognize evidence generation across the lifecycle.

7. Risk Decisions

Findings are not the same as decisions. A useful release package shows which findings were fixed, deferred, rejected as false positives, mitigated with controls, or accepted by the right authority.

8. Deployment Record

The release should show where the artifact was deployed, who approved promotion, what environment received it, what configuration was applied, and what rollback path exists.



9. Operational Feedback

The release record should connect to runtime data: logs, alerts, vulnerability status, configuration drift, incident reports, performance data, and user or operator feedback.

This package does not need to be elegant at first. It needs to be consistent.

DevSecOps, RMF, cATO, CMMC, and DFARS: Different Evidence Consumers

One source of confusion in DoD software delivery is that several frameworks and authorities consume overlapping evidence.

RMF and ATO

The Risk Management Framework is the baseline process for categorizing systems, selecting controls, implementing controls, assessing controls, authorizing systems, and monitoring risk. NIST SP 800-37 describes RMF as structured but flexible. It integrates security and privacy risk management into the system lifecycle. ([NIST Computer Security Resource Center](#))

For ATO work, evidence supports control implementation, assessment, risk acceptance, and ongoing monitoring.

cATO

Continuous Authorization to Operate is not a shortcut around RMF. DoD cATO guidance says RMF steps must still be followed and that systems must already be in the Monitor phase before applying for cATO.

For cATO, evidence must show that the organization has mature monitoring, active cyber defense, secure software supply chain practices, and a DevSecOps platform that produces useful risk information. DoD's cATO implementation guide emphasizes pipeline guardrails, control gates, evidence collection, dashboards, feedback loops, and risk governance. ([U.S. Department of Defense CIO](#))

CMMC

CMMC is about verifying that defense contractors and subcontractors have implemented required cybersecurity measures. Those measures protect Federal Contract Information and Controlled Unclassified Information in applicable contractor systems. The CMMC Program rule became effective December 16, 2024. ([Federal Register](#)) The DFARS rule for CMMC contract requirements became effective November 10, 2025 and includes a phased rollout. ([GovInfo](#))



For CMMC, evidence supports assessment of required practices, contractor system security, and contract eligibility where CMMC requirements apply.

DFARS 252.204-7012

DFARS 252.204-7012 requires adequate security for covered contractor information systems. It also requires rapid reporting of cyber incidents that affect covered defense information or operationally critical support. The clause requires review for evidence of compromise and reporting within 72 hours. It also requires malicious software submission when applicable. Affected system images and monitoring data must be preserved for at least 90 days. ([Acquisition.gov](https://www.acquisition.gov))

For DFARS incident response, evidence supports reporting, containment, forensic analysis, subcontractor coordination, and damage assessment.

The same engineering system can support all of these needs, but only if evidence is intentionally designed into the workflow.

What Good Looks Like

A mature DevSecOps environment for secure software should be able to produce release evidence without a special meeting, a spreadsheet hunt, or a week of manual reconstruction.

For secure mission environments, this is a readiness test. The team should be able to prove what it shipped while the work is still fresh.

For any release, the team should be able to answer:

- What changed?
- Why did it change?
- Who reviewed it?
- What artifact was built?
- What dependencies are included?
- What tests were run?
- What scans were run?
- What findings remain?
- What risk decisions were made?
- What control evidence was updated?
- What environment received the artifact?
- What telemetry confirms the system remains within expected bounds?

The best version of this is not a binder. It is not a static compliance package. It is an evidence chain connected to the artifact.



Implementation Pattern

Programs can adopt this approach incrementally.

Step 1: Define the Boundary

Start with the system boundary, software factory boundary, deployment boundary, data types, authorization context, inherited controls, and external dependencies.

Do not skip this step. A pipeline without a boundary model will produce evidence that is difficult to use.

Step 2: Identify the Evidence Consumers

List who needs evidence and why. Typical consumers include engineering leads, cyber teams, ISSOs, ISSMs, authorizing officials, mission owners, and program managers. Configuration control boards, incident responders, and external assessors may need the same facts in different forms.

Each consumer needs different views of the same underlying evidence.

Step 3: Build the Evidence Register

Create an evidence register that maps:

- Evidence type
- Source system
- Owner
- Update frequency
- Associated control, requirement, or risk
- Retention requirement
- Review process
- Export or reporting format

This can start as a spreadsheet. It should eventually become part of the program's normal engineering system.

Step 4: Put Evidence Generation in the Pipeline

Automate what can be automated: builds, tests, scans, SBOMs, artifact signing, dependency checks, configuration checks, and deployment records.

The goal is not to eliminate judgment. The goal is to reserve human judgment for actual risk decisions.



Step 5: Connect Findings to Decisions

Security findings, test failures, waivers, POA&M items, accepted risks, and compensating controls should not live in disconnected systems. They should be traceable to the artifact and release decision they affected.

Step 6: Design for Boundary Crossing

If software will move between enclaves, labs, software factories, or operational platforms, define the promotion evidence required at each crossing.

This is where many programs fail. They automate the build but manualize the handoff.

Step 7: Close the Loop With Operations

Operational telemetry should feed back into the backlog. Incidents, drift, vulnerabilities, operator feedback, performance regressions, and mission-thread failures should become engineering inputs.

This is the difference between DevSecOps as a pipeline and DevSecOps as a mission sustainment model.

Common Failure Modes

Toolchain Without Governance

Buying tools does not create DevSecOps. Without risk tolerances, promotion rules, evidence ownership, and review cadence, tools create noise.

Compliance Outside the Pipeline

When compliance evidence is collected after the release, it is usually stale, incomplete, or disconnected from the actual artifact.

Security Findings Without Risk Decisions

A list of vulnerabilities is not a risk posture. Findings need disposition: fix, defer, accept, mitigate, or reject as false positive.

Boundary Handoffs Without Provenance

Secure mission systems often fail during handoff. The artifact may be built correctly, but the receiving environment cannot verify enough about its origin, configuration, dependencies, or risk status.



Dashboards Without Traceability

A dashboard that shows red, yellow, and green status is not enough. The dashboard must connect back to artifacts, controls, findings, and decisions.

How Polyrhythm Helps

Polyrhythm focuses on the engineering boundaries where secure software succeeds or fails:

- Open interfaces and schema-controlled exchanges
- Modular software architectures
- Model-to-mission traceability
- Build, sign, scan, and deploy workflows
- Reproducible artifacts
- Multi-enclave promotion planning
- Secure-by-design implementation
- Control mapping and authorization readiness
- Testable integration paths from lab to field

This aligns with Polyrhythm's broader open-architecture and secure software posture: clear boundaries, portable services, predictable integration, and automated delivery discipline. It also aligns with Polyrhythm's current secure environment and authorization support work. That work includes control mapping, evidence organization, authorization readiness, secure-by-design engineering, platform hardening, monitoring alignment, and artifact-based validation.

The objective is not to create more paperwork. The objective is to create software delivery systems that produce proof as they produce capability.

Conclusion

DevSecOps in secure mission environments is not primarily a tooling problem. It is an evidence architecture problem.

A program does not become mature because it has a pipeline. It becomes mature when the pipeline, the people, the controls, the artifacts, and the feedback loop produce enough evidence to support risk decisions.

In that environment, evidence is not an afterthought. It is part of the release.

That is the standard secure software now requires: build the capability, prove the artifact, preserve the chain, and keep the mission moving.



References

1. [Cybersecurity Maturity Model Certification \(CMMC\) Program, Federal Register](https://www.federalregister.gov/documents/2024/10/15/2024-22905/cybersecurity-maturity-model-certification-cmmc-program) — <https://www.federalregister.gov/documents/2024/10/15/2024-22905/cybersecurity-maturity-model-certification-cmmc-program>
2. [DFARS 252.204-7012 Safeguarding Covered Defense Information and Cyber Incident Reporting](https://www.acquisition.gov/dfars/252.204-7012-safeguarding-covered-defense-information-and-cyber-incident-reporting) — <https://www.acquisition.gov/dfars/252.204-7012-safeguarding-covered-defense-information-and-cyber-incident-reporting>
3. [DoD Software Modernization Implementation Plan, Unclassified Summary](https://dodcio.defense.gov/Portals/o/Documents/Library/SW-Mod-I-Plan25-26.pdf) — <https://dodcio.defense.gov/Portals/o/Documents/Library/SW-Mod-I-Plan25-26.pdf>
4. [DoD Enterprise DevSecOps Fundamentals](https://dodcio.defense.gov/Portals/o/Documents/Library/DoD%20Enterprise%20DevSecOps%20Fundamentals%20v2.5.pdf) — <https://dodcio.defense.gov/Portals/o/Documents/Library/DoD%20Enterprise%20DevSecOps%20Fundamentals%20v2.5.pdf>
5. [NIST SP 800-218, Secure Software Development Framework \(SSDF\) Version 1.1](https://csrc.nist.gov/pubs/sp/800/218/final) — <https://csrc.nist.gov/pubs/sp/800/218/final>
6. [NIST SP 800-37 Rev. 2, Risk Management Framework for Information Systems and Organizations](https://csrc.nist.gov/pubs/sp/800/37/r2/final) — <https://csrc.nist.gov/pubs/sp/800/37/r2/final>
7. [DoD DevSecOps cATO Implementation Guide](https://dodcio.defense.gov/Portals/o/Documents/Library/DoDCIO-ContinuousAuthorizationImplementationGuide.pdf) — <https://dodcio.defense.gov/Portals/o/Documents/Library/DoDCIO-ContinuousAuthorizationImplementationGuide.pdf>
8. [Federal Register 90 FR 43560, DFARS CMMC final rule](https://www.govinfo.gov/app/details/FR-2025-09-10/2025-17359) — <https://www.govinfo.gov/app/details/FR-2025-09-10/2025-17359>