Managing “Unfixable” CVEs Without Turning Exceptions Into Permanent Blind Spots

Evidence-led operating model • Illustrated edition

Executive conclusion

The proposed three-part pattern—formal risk disposition, contextual exploitability analysis, and structural attack-surface reduction—is directionally sound, but it needs two corrections before a security team can safely present it as a compliance operating model.

First, there is no generally applicable rule that a vulnerability exception should expire after 90 days. A 90-day review may be a sensible internal default for some risks, but the controlling interval comes from the applicable contract, law, authorization boundary, framework, assessor interpretation, and the organization’s own risk policy. Some obligations are much shorter. PCI DSS v4.0.1, for example, requires critical security patches and updates to be installed within one month of release, while other applicable patches follow timeframes determined by the entity’s risk assessment; PCI SSC also distinguishes a vulnerability that is *resolved* from one that is *addressed* through another measure such as disabling a vulnerable service or applying a compensating control (PCI SSC FAQ 1597). For US federal civilian agencies, entries in CISA’s Known Exploited Vulnerabilities (KEV) Catalog carry specific remediation due dates under Binding Operational Directive 22-01; a generic 90-day acceptance cannot supersede those dates (CISA KEV Catalog). Risk acceptance is therefore a governed decision, not an automatic extension of a scanner SLA.

Second, “reachability” is not one thing, and absence of a discovered call path is not proof of non-exploitability. Source call-graph analysis, build-time component resolution, runtime observation, configuration analysis, network exposure, and a supplier’s VEX statement answer different questions. Snyk explicitly labels a negative result **NO PATH FOUND**, warns that this does not mean the vulnerability is completely unreachable or unexploitable, and identifies incomplete information, control-flow complexity, reflection, and other dynamic behavior as limitations (Snyk reachability documentation). Grype, by contrast, primarily inventories packages (directly or through an SBOM), matches their names and versions to vulnerability records, and then applies ignore rules and VEX statements; its documented architecture does not perform application call-graph reachability analysis (Grype architecture, Grype result filtering).

The defensible operating model is consequently broader than three sequential steps:

1. **Verify the finding and identify the governing requirement.**
2. **Classify the disposition correctly:** false positive, not affected, affected and mitigated, affected and accepted, or fixed.
3. **Build a reproducible exploitability and impact case** from component identity, vulnerable conditions, technical reachability, exposure, privilege, asset criticality, threat intelligence, and compensating-control tests.
4. **Prefer removal, replacement, isolation, feature disablement, or supported upgrade over acceptance.**
5. **Have the accountable risk owner approve any residual risk**, with an explicit end condition, review triggers, and evidence-preserving workflow.
6. **Continuously re-evaluate** when code, configuration, deployment, threat intelligence, vendor status, control effectiveness, or the applicable requirement changes.

This is not a way to “make the scanner green.” It is a way to keep the scanner finding visible while representing its real disposition accurately and showing that the organization made, authorized, tested, and continues to monitor a risk decision.

The core issue: a CVE finding is evidence of a possible condition, not the final risk decision

A conventional SCA or container scan usually combines an observed component identity and version with an advisory asserting that a version range is affected. That result is useful, but several questions remain:

- Did the scanner identify the component and version correctly?
- Does the advisory apply to the distributor’s build, including backported fixes?
- Is the affected component present in the deployed artifact rather than only in a build stage, test scope, cache, or package-manager metadata?
- Are the vulnerable functions, modules, features, protocols, or configurations present and enabled?
- Can an attacker cross the necessary network, identity, tenant, privilege, and input-validation boundaries?

- What would exploitation affect in this asset and business process?
- Is exploitation occurring in the wild?
- Do existing or additional controls reduce likelihood or impact enough to fall within the authorized risk tolerance?
- Does the governing requirement permit a compensating control or risk acceptance at all?

NIST frames patching as one of several vulnerability-risk responses. Its enterprise patch-management guidance lists acceptance, mitigation, transfer, and avoidance. Mitigation may include patching, disabling a vulnerable feature, upgrading, firewalls, or segmentation; avoidance may include uninstalling the software, decommissioning the asset, or disabling unnecessary computing capability. NIST also makes the limit clear: patching/updating/upgrading are the only responses that completely remove the vulnerability without removing functionality (NIST SP 800-40 Rev. 4, §2.1).

This distinction explains why auditors resist silent suppressions. A suppression changes what a tool displays; it does not change whether the vulnerable condition exists, whether exploitation is possible, whether the risk was authorized, or whether a compliance objective is met. A defensible record must connect the raw technical observation to a controlled disposition and preserve both.

"Unfixable" should be a temporary technical state, not a governance category

Security teams use *unfixable* to describe several materially different states:

State	What it actually means	Appropriate next action
No upstream fix exists	The maintainer has not published a corrected version or patch	Remove/disable/isolate if possible; monitor upstream and threat changes; consider temporary mitigation or acceptance
Product is end-of-life	The supplier will not provide a fix	Migrate, replace, isolate, or decommission; acceptance should have a funded exit plan
Fix exists but is incompatible	Upgrade causes a documented functional, safety, or availability failure	Engineer around the incompatibility, isolate, or accept only the residual risk for a bounded period
Distribution backport is not recognized	The scanner compares upstream versions and misses a vendor patch	Verify the distributor advisory and package build; correct scanner matching or record a false positive/not-affected disposition
Vulnerable code is absent	Package identity is present but the affected code was removed or not built	Produce a `not_affected` statement with evidence
Vulnerable code cannot execute in the product configuration	Required feature, platform, or inline mitigation prevents execution	Produce a carefully scoped `not_affected` or mitigated disposition, depending on the facts and VEX vocabulary
Code is currently not reached	Static or dynamic analysis found no path in the analyzed scope	Treat as prioritization evidence, not conclusive proof; define assumptions and re-test triggers
A fix is operationally deferred	The patch awaits testing, outage, certification, or coordinated release	Track a time-bounded remediation plan and interim controls; do not describe this as permanently unfixable

NIST expressly recognizes that a patch may not yet exist, that a vendor may have ended support, and that an organization may need testing, an outage window, dependent updates, or training before deployment (NIST SP 800-40 Rev. 4, §2.1). Those are reasons to choose and document an interim response; they are not evidence that the risk is negligible.

What compliance and assurance regimes actually support

NIST-based programs: explicit, accountable, continuously monitored risk decisions

NIST's Risk Management Framework is built around control selection, assessment, authorization, and continuous monitoring, with senior leaders receiving information needed to make risk decisions (NIST SP 800-37 Rev. 2). NIST's OSCAL POA&M; model makes the expected evidence structure concrete: risks include weakness descriptions and risk characteristics; deviations can represent false positives, risk adjustments, and risk acceptance; POA&M; items include remediation plans and status (NIST OSCAL POA&M; model). NIST defines a compensating control as a management, operational, or technical control used in place of a recommended baseline control and providing equivalent or comparable protection (NIST CSRC glossary).

The practical lesson is that a line manager or scanner administrator should not unilaterally "accept" material enterprise risk unless the organization has expressly delegated that authority at that level. The accountable role should be defined by policy, calibrated to risk magnitude, and able to own the business consequence. The security team supplies the analysis; the designated risk owner makes the decision.

A POA&M; is also not synonymous with acceptance. It normally tracks a weakness, planned corrective actions, owners, resources, milestones, dates, and status. NIST OSCAL separately represents response types such as `avoid`, `mitigate`, `transfer`, `accept`, and `none` for a false positive (OSCAL POA&M; response model). Keeping those concepts separate prevents the common contradiction of marking a risk "accepted" while also claiming it has a committed remediation date, without saying which condition ends the acceptance.

FedRAMP's current cloud-service guidance reinforces the distinction. Provider-maintained vulnerability lists, product roadmaps, accepted-vulnerability records, and internal risk tracking do not automatically become an agency POA&M.; An agency POA&M; is appropriate when the agency itself has an action, compensating control, monitoring activity, or risk decision to own (FedRAMP agency POA&M; guidance). That approach reduces duplicate paperwork while preserving ownership.

PCI DSS: risk analysis does not waive applicable requirements

PCI DSS is a useful counterexample to the idea that a generic risk memo always “satisfies compliance.” PCI SSC states that organizations may not use risk assessment to avoid or bypass an applicable requirement. A conclusion that a requirement is not applicable to a system must be verified and supported by documented evidence; controls used to reduce applicability must also be verified as correctly implemented and effective (PCI SSC applicability FAQ).

PCI DSS v4.x offers several distinct paths:

- The **defined approach** implements the requirement as written.
- Within the defined approach, a **compensating control** may be used when a legitimate, documented technical or business constraint prevents compliance as written. The control must meet the intent and rigor of the original requirement, provide a similar level of defense, address the additional risk, be above and beyond other PCI DSS requirements, and be documented and assessed.
- The **customized approach** is an intentionally different implementation that meets the requirement’s customized-approach objective and entails targeted risk analysis, control design, testing, and maintenance.

PCI SSC emphasizes that compensating controls are for current and future operation and cannot retroactively repair a requirement that was simply missed (PCI SSC, “Compensating Controls vs Customized Approach”). Its compensating-control worksheet asks for the constraint, the control, the original objective and objective met, additional risk, validation/testing, and maintenance process (PCI DSS v4.0 SAQ D Merchant, Appendix B). Thus a firewall rule named in an exception is not enough; evidence must show that the rule is correctly scoped, enforced, tested, monitored, and maintained.

CISA KEV: known exploitation can override ordinary queueing assumptions

CISA describes the KEV Catalog as the authoritative source of vulnerabilities known to have been exploited in the wild and recommends that organizations use it as an input to vulnerability-management prioritization (CISA KEV Catalog). Under BOD 22-01, US federal civilian executive branch agencies must remediate catalog vulnerabilities by the listed due dates. Catalog instructions commonly require applying vendor mitigations, following applicable cloud guidance, or discontinuing use when mitigations are unavailable.

Consequently:

- A low static reachability score should not automatically outrank confirmed exploitation evidence.
- An expired vendor, unavailable patch, or breaking upgrade may make the operational problem harder, but does not erase a mandatory due date.
- Where the applicable authority permits mitigation, the record must cite the vendor/CISA mitigation and verify that it is effective in the actual deployment.
- Where the authority requires removal or discontinuation in the absence of mitigation, ordinary risk acceptance may not be an available compliance path.

The 90-day misconception

Ninety days is best treated as an **illustrative internal cadence**, never as a universal auditor expectation. Review dates should be derived from:

1. the explicit external remediation deadline;
2. exploitation status and speed;
3. internet or adversary accessibility;
4. asset and mission impact;
5. stability and coverage of interim controls;
6. expected upstream release or migration milestone;
7. the organization’s formally approved risk appetite and delegation matrix; and
8. event-driven triggers.

A critical internet-facing KEV may need daily monitoring and remediation in days. A genuinely not-affected library instance supported by reproducible build evidence may be reassessed on every build or dependency change rather than every calendar quarter. A low-impact isolated legacy component may receive a quarterly review, but the choice comes from policy and facts—not folklore.

A rigorous workflow for an unremediable finding

1. Preserve and normalize the raw finding

Do not begin by suppressing it. Create a durable record containing:

- scanner product and version;
- vulnerability database version and scan timestamp;

- CVE or other advisory identifiers and aliases;
- package URL (purl), CPE where applicable, supplier, package name, version, architecture, and cryptographic digest;
- dependency path and whether the component is direct, transitive, development-only, or runtime;
- artifact/image digest and deployment identifiers;
- detection location and evidence;
- advisory source and affected/fixed version ranges;
- scanner severity, CVSS vector/version, and fix state;
- first-seen and last-seen timestamps;
- affected assets, owners, data classification, network zone, and business service;
- prior dispositions and their authors.

This is necessary because vulnerability data changes. Advisories are enriched, version ranges are corrected, aliases are merged, packages are rebuilt, images move under mutable tags, and scanner databases update. A decision that cannot be reproduced against the original evidence will be difficult to defend later.

Container findings should be bound to an immutable image digest, not only a tag. Docker notes that tags are mutable and recommends digest pinning for reproducibility, while also warning that a pinned digest opts out of automatic base-image updates unless an update workflow detects and proposes the new digest (Docker build best practices). The correct pattern is both reproducibility and an automated update signal.

2. Identify the exact obligation before choosing the disposition

Record:

- framework/control/contract citation;
- assessment scope and system boundary;
- whether the control is prescriptive, objective-based, or risk-based;
- required remediation interval;
- whether compensating controls, a customized approach, POA&M;, deviation, or acceptance are permitted;
- approver and assessor roles;
- required evidence and retention;
- any special treatment for KEV, internet-facing assets, safety systems, payment systems, or regulated data.

This step prevents a technically elegant reachability report from being used where the requirement simply demands patching, removal, or an assessor-approved compensating control.

3. Validate component identity and advisory applicability

Package scanners can be wrong because ecosystem, namespace, distribution, version syntax, or CPE matching is wrong. Validate:

- Is the component actually in the deployable artifact?
- Is it from the ecosystem assumed by the advisory?
- Is the version comparison appropriate for that ecosystem?
- Has the Linux distribution backported the security fix without adopting the upstream version?
- Is the component a source archive, metadata, cache, test fixture, or executable runtime dependency?
- Does the advisory require a particular platform, architecture, compiler option, feature, or configuration?
- Does the supplier publish an advisory or VEX statement for the exact product version?

Grype's documented matching pipeline illustrates why this matters: it prepares package candidates, selects ecosystem-specific matchers, queries vulnerability records, compares versions, rejects explicit unaffected records, then applies ignore and VEX processing (Grype architecture). This is valuable correlation, but it is not equivalent to demonstrating executable control flow.

If the match is wrong, use a **false-positive** disposition with the evidence that disproves it. If the component match is right but the product is not affected because vulnerable code is absent or required conditions cannot occur, use a **not-affected** disposition. Do not call either one "accepted risk": acceptance concedes that residual risk remains.

4. Analyze vulnerable conditions, not merely package presence

Start from the vulnerability's mechanics:

- affected function, class, module, driver, parser, protocol, endpoint, or feature;
- attacker prerequisites;
- input source and required data shape;
- authentication and authorization state;
- local versus remote access;
- user interaction;
- required privileges and resulting privileges;
- operating system, CPU, library, compiler, and configuration conditions;
- confidentiality, integrity, availability, safety, and cross-tenant consequences;
- known exploit and exploitation evidence.

Map those conditions onto the deployment. A good attack-path narrative is testable:

> *Untrusted input from boundary A can/cannot reach interface B; identity control C permits/denies the required principal; configuration D enables/disables feature E; application call path F reaches/does not establish a path to vulnerable function G; sandbox H and network policy I limit the post-exploitation consequence to J.*

Each link should cite a configuration export, code location, test, trace, policy decision, or authoritative advisory. Avoid conclusory sentences such as “not internet-facing, therefore not exploitable.” An attacker may reach the service through a compromised workload, SSRF, message queue, uploaded file, management plane, VPN, partner connection, or malicious insider.

5. Use reachability as layered evidence

Source and call-graph reachability

Static analysis can identify a call path from first-party code through dependencies to a vulnerable symbol. A positive path is strong prioritization evidence. A negative result is weaker because soundness and completeness are difficult in the presence of:

- reflection and dynamic dispatch;
- dependency injection and plugin loading;
- native interfaces;
- generated code;
- callbacks and framework inversion of control;
- deserialization;
- runtime code generation;
- environment-specific modules;
- scripting/evaluation;
- incomplete source or build configuration;
- optional code activated only by rare input or feature flags.

Snyk says its analysis uses fix commits to identify related/root-cause elements and analyzes application and dependency call graphs. It also expressly warns that static analysis can establish at least one reachable path, while no discovered path does not establish impossibility; its UI therefore says `NO PATH FOUND`, not `UNREACHABLE` (Snyk reachability analysis).

The audit record should preserve tool version, supported language/ecosystem, analyzed commit, source scope, build files, configuration, result, call path when positive, and documented limitations. Re-run on relevant code or dependency changes.

Build and artifact reachability

Ask whether the vulnerable material is:

- resolved by the package manager;
- downloaded in a build stage;
- compiled or linked;
- copied into the final artifact;
- loadable for the target platform;
- included only in tests, docs, examples, or caches;
- stripped by tree shaking, dead-code elimination, or linker settings.

Multi-stage container builds provide a strong structural boundary when only the required runtime artifacts are copied into the final stage. Docker explains that each `FROM` starts a stage and selective `COPY --from` leaves unwanted build material behind (Docker multi-stage build documentation). Verify the final image or binary; do not infer it solely from the Dockerfile.

Runtime reachability

Runtime traces, coverage, eBPF telemetry, loaded-module inventories, audit logs, or integration tests can show that code executed under observed workloads. They cannot prove that an unobserved adversarial input will never execute it. Record workload duration, traffic representativeness, enabled features, environment, test cases, and telemetry blind spots.

Dynamic evidence is particularly useful for verifying that a compensating control blocks a concrete exploit precondition, but tests should include negative and bypass cases rather than only confirming the expected happy path.

Network and identity reachability

Network reachability asks whether an attacker-controlled source can reach the relevant listener or data path. Identity reachability asks whether the required principal, token, role, or tenant relationship can be obtained. Evidence may include:

- cloud security-group and firewall evaluation;
- Kubernetes NetworkPolicy and service exposure;
- ingress, API gateway, service mesh, and load-balancer routes;
- IAM policy simulation;
- authentication mode;
- egress controls;
- segmentation test results;
- asset inventory and external attack-surface observations.

A diagram alone is insufficient. Auditors need evidence that the deployed rules match the diagram and are tested continuously.

Configuration reachability

Many advisories apply only when a feature, protocol, codec, endpoint, or option is enabled. Capture the effective runtime configuration, not only a repository default. Include inheritance, environment variables, startup arguments, overlays, admission mutations, and tenant-specific settings.

Reachability is not impact

Even a reachable vulnerable function may require privileges an attacker cannot obtain or may execute within a sandbox that sharply limits impact. Conversely, a “low” package finding in a highly privileged internet-facing control plane can deserve urgent action. Threat, exploitability, exposure, asset value, and impact should remain separate inputs so reviewers can see the reasoning.

FIRST’s EPSS estimates the probability of observing exploitation activity for a published CVE in the next 30 days and is updated daily (FIRST EPSS model). FIRST cautions that EPSS does not include a specific environment, compensating controls, or impact and should not be treated as a complete risk score; direct evidence of active exploitation should supersede the estimate (FIRST EPSS FAQ). EPSS can therefore prioritize investigation but cannot prove that a particular deployment is safe.

6. Express the conclusion with VEX where appropriate

Vulnerability Exploitability eXchange (VEX) is designed to communicate a product/component’s status with respect to a vulnerability in machine-readable form. CISA’s minimum-elements document describes statuses including affected, not affected, fixed, and under investigation, along with product, vulnerability, timestamp, and status details (CISA Minimum Requirements for VEX). CISA also notes that VEX can be authored by a supplier or a third party and that its guidance does not itself impose compliance.

For `not_affected`, use a standardized justification only when the evidence supports it. Common concepts include:

- vulnerable code is not present;
- vulnerable code is not in the execute path;
- vulnerable code cannot be controlled by an attacker;
- inline mitigations already prevent exploitation.

The assertion must identify the exact product/version and vulnerability. A VEX statement for one immutable artifact should not silently apply to another build, configuration, or deployment. Include an impact statement or action statement explaining the reasoning and any conditions.

Grype can consume VEX documents and, by default, move `not_affected` and `fixed` matches into its ignored list (Grype VEX filtering). That is an implementation mechanism for propagating a disposition; Grype does not independently prove the VEX assertion. The organization still needs provenance, approval, scope, and supporting evidence.

VEX is especially useful for reducing endless tool-specific exceptions:

1. Preserve one signed or provenance-bound disposition.
2. Associate it with an immutable product identifier.
3. Feed it to scanners, dashboards, ticketing, and customer assurance.
4. Expire or supersede it when its assumptions change.

Do not issue `not_affected` when the truth is “affected, but we believe current controls reduce the risk.” That state should remain visible as affected/mitigated or accepted according to the chosen schema and governance process.

7. Prefer structural reduction and supported change

Remove what is not needed

NIST’s avoidance response includes uninstalling vulnerable software, decommissioning affected assets, and disabling capabilities that are not required (NIST SP 800-40 Rev. 4, §2.1). This is often stronger than repeatedly accepting every CVE associated with an unused package.

For containers:

- separate build and runtime stages;
- copy only required runtime artifacts;
- avoid package-manager caches, compilers, shells, debuggers, and editors in production unless operationally required;
- use `--no-install-recommends` or ecosystem equivalents;
- remove unused plugins, locales, protocols, and helper binaries;
- scan the final image by immutable digest;
- rebuild frequently from supported and patched inputs.

Docker recommends minimal trusted base images, separate build/test and runtime images, and avoiding unnecessary packages; it states that smaller images reduce dependencies and vulnerabilities introduced by those dependencies (Docker build best practices). NIST likewise warns that container images can become vulnerable as components age and that unnecessary services such as an SSH daemon expose a container to avoidable network risk (NIST SP 800-190, §3.1).

Minimal and distroless images: real benefits, real trade-offs

Minimal or distroless images can remove entire classes of findings by omitting software rather than suppressing it. They do not make remaining code invulnerable, and they can create operational costs:

- fewer diagnostic tools during incidents;
- incompatibility with applications that assume a shell, package database, user records, CA bundle, timezone data, or dynamic libraries;
- harder forensic acquisition;
- different libc behavior;
- pressure to copy ad hoc utilities into a running container;
- risk of stale application dependencies even when the OS layer is small.

Docker’s distroless guidance recommends multi-stage builds, explicit runtime dependencies, CI validation, and continued CVE monitoring even for minimal images (Docker minimal/distroless guidance). Teams should test startup, TLS trust, DNS, locale/timezone behavior, health checks, signal handling, observability, crash dumps, and incident-response procedures before migration.

Rebuilding is part of remediation

Container images are immutable snapshots. Updating the base-image tag in source does not change deployed images until they are rebuilt and redeployed. Docker recommends regular rebuilds and using `--pull` to retrieve a fresh base image (Docker build best practices). Evidence of closure should therefore include the new image digest, scan result, deployment rollout, and verification that old vulnerable replicas and registry references are no longer in use.

8. Design and test compensating controls against the actual exploit chain

A compensating control is not any control that sounds relevant. It must reduce the missing control’s risk to the level required by the governing framework. Map each exploit prerequisite to a preventive, detective, or impact-limiting control:

Exploit prerequisite or consequence	Candidate control	Evidence to retain
Internet request reaches vulnerable endpoint	Route removal, WAF/API rule, gateway deny, service unpublishing	Effective configuration, external test, bypass test, change monitor

Exploit prerequisite or consequence	Candidate control	Evidence to retain
Untrusted file reaches parser	File-type allowlist, content disarm, sandbox, queue isolation	Test corpus, rejected samples, sandbox policy, alert evidence
Feature must be enabled	Feature disabled and configuration locked	Effective config, policy-as-code test, drift alert
Attacker needs a privileged identity	Strong authentication, least privilege, conditional access	Policy export, entitlement review, simulation/test
Exploit requires lateral movement	Segmentation, host firewall, workload identity	Reachability test matrix, denied-flow logs
Exploit yields code execution	Non-root runtime, read-only filesystem, seccomp/AppArmor/SELinux, capability removal	Runtime spec, admission policy, enforcement test
Exploit contacts command-and-control	Default-deny egress, DNS/proxy policy	Egress policy, denied-domain test, alert telemetry
Exploit affects sensitive data	Data minimization, encryption/key isolation, tenant boundary	Data-flow evidence, access test, key policy

Controls should be evaluated as a chain. A WAF signature may reduce one payload pattern while leaving alternate encodings, protocols, internal routes, or non-HTTP entry points. A network policy may not apply to host networking or an unmanaged namespace. Running as non-root limits some consequences but does not prevent data theft available to the application identity.

Validation should include:

- a stated security objective;
- exact assets and paths covered;
- configuration and enforcement point;
- failure and bypass modes;
- positive and negative test cases;
- alert delivery and response;
- ownership and on-call process;
- drift detection;
- dependencies on other controls;
- residual likelihood and impact after the control;
- independent review where required.

9. Write the risk record so another reviewer can reproduce the decision

A professional risk disposition should contain at least:

Identity and scope

- unique exception/risk ID;
- CVE/advisory aliases;
- component identity and version;
- artifact digest, commit, release, environment, assets, and tenants;
- governing controls and scanner-policy references.

Finding validation

- raw finding;
- authoritative vendor/distributor advisory;
- component and version verification;
- affected conditions;
- correction of any false-positive matching.

Threat and attack-path analysis

- adversary and access prerequisites;
- exposure and trust boundaries;
- call/build/runtime/configuration reachability evidence;
- privileges and user interaction;
- KEV status, exploit availability, and EPSS as a supporting—not dispositive—signal;
- business and technical impact.

Response and residual risk

- selected response: avoid, mitigate, transfer/share, accept, or no action because false positive;
- alternatives considered;
- why patch/upgrade is unavailable or currently infeasible;
- compensating controls and validation;
- inherent and residual risk using the organization's approved method;
- uncertainty and evidence gaps.

Accountability and time

- remediation/exit plan, owner, funding, milestones, and target;
- designated risk owner's approval;
- assessor approval where the framework requires it;
- effective date, review date, and hard expiration if policy uses one;
- event-driven revocation triggers;
- escalation path for overdue items.

Machine and audit lifecycle

- VEX statement or tool suppression bound to the approved record;
- creation and expiry timestamps;
- ticket and evidence links;
- immutable approval/audit log;
- re-scan and control-test cadence;
- closure evidence.

The scanner exception should be the *last* artifact generated from the risk record, not the record itself. Require a reason, owner, and expiry in the scanner where supported, and reconcile scanner ignores against the authoritative risk register.

10. Use both time-driven and event-driven review

A calendar expiration is useful because it prevents forgotten decisions, but event triggers are more responsive. Reopen the decision when:

- a patch, supported release, or vendor mitigation appears;
- the component or base image changes;
- code or configuration changes;
- the service becomes internet-accessible or moves zones;
- the asset begins processing more sensitive data;
- the vulnerability enters KEV or exploitation intelligence changes;
- a working exploit becomes public;
- EPSS changes materially;
- the compensating control fails a test or drifts;
- the business owner, risk tolerance, framework, contract, or authorization changes;
- the exception reaches its expiration.

The result of review should be a new versioned decision, not an overwritten date. Repeated renewals are a governance signal. Track exception age, number of renewals, overdue exit milestones, and concentration by product owner or unsupported platform.

Evidence hierarchy for auditors

Not all evidence has equal strength. A practical hierarchy is:

1. **Direct remediation or removal evidence:** corrected version in the deployed artifact, package absent, feature removed, asset decommissioned.
2. **Authoritative product-specific evidence:** supplier/distributor advisory or signed VEX for the exact version.

3. **Reproducible technical evidence:** artifact inspection, call path, build provenance, configuration export, network/IAM simulation, exploit-condition test.
4. **Control-effectiveness evidence:** repeated tests, telemetry, alerts, drift monitoring, independent assessment.
5. **Reasoned analysis:** attack-path model tied to documented assumptions.
6. **Assertion without evidence:** “not used,” “internal only,” or “WAF protects it.”

The fifth category can be valid when direct proof is impossible, but uncertainty should increase residual risk and shorten review cadence. The sixth category should not support suppression.

What the three proposed steps get right—and where each can fail

1. Formal risk acceptance with expiration

What is right: silent suppression is not risk management. An authorized, reviewable record with a clear owner, rationale, residual risk, compensating controls, and end condition is substantially more defensible. NIST’s models explicitly support risk acceptance and risk deviations, and PCI requires detailed, tested documentation for compensating controls.

Necessary correction: not every unresolved CVE should be “accepted.” False positives, not-affected conditions, mitigated vulnerabilities, and planned remediation are different. Approval authority and review interval must come from policy and the applicable regime. A 90-day expiration is a possible internal control, not an auditor-wide norm.

Failure modes:

- security approves risk on behalf of a business owner without delegated authority;
- the record documents gross CVSS but not deployment-specific likelihood/impact;
- the “compensating control” is an untested existing baseline control;
- acceptance auto-renews;
- the exception hides the finding from dashboards and downstream consumers;
- the exit plan has no funded owner;
- a mandatory deadline or KEV instruction is ignored.

2. Contextual reachability analysis

What is right: package presence alone overstates actionable exposure in many contexts. Call paths, build inclusion, runtime loading, configuration, network access, identity, and attacker control can sharply improve prioritization and can support a `not_affected` VEX statement when the evidence truly establishes it.

Necessary correction: Snyk and Grype are not interchangeable examples. Snyk documents source/call-graph reachability for supported ecosystems. Grype performs package/vulnerability matching and can consume VEX; it does not itself prove source-code reachability. Moreover, “no path found” is not equivalent to mathematical proof of unreachability.

Failure modes:

- analyzing a repository revision different from the deployed artifact;
- unsupported language/framework treated as a negative result;
- reflection or plugin loading omitted;
- runtime observation window misses rare adversarial inputs;
- network reachability ignores internal compromise paths;
- `not_affected` is copied across versions or products;
- threat/impact evidence is ignored because code reachability appears low.

3. Structural reduction

What is right: removing the vulnerable package, feature, service, or asset is stronger than accepting its risk. Minimal runtime images and multi-stage builds reduce components and attack surface, and authoritative NIST and Docker guidance support those practices.

Necessary correction: fewer findings are not automatically equivalent to lower total risk. A distroless image can still contain a critical vulnerable application library; a small image can be poorly configured; and loss of diagnostic tools can impair operations. Structural reduction must be verified against the final artifact and accompanied by rebuild, testing, and incident-response design.

Failure modes:

- scanning a build stage instead of—or without—the final image;
- removing package metadata so a scanner can no longer identify a library while the vulnerable code remains;

- changing libc/base family and breaking behavior;
- failing to rebuild/redeploy;
- using mutable tags without tracking the deployed digest;
- declaring victory based on CVE count rather than exploitability and impact.

An implementation blueprint that avoids endless manual exceptions

Policy layer

Define a disposition taxonomy and approval matrix:

- `false_positive`;
- `not_affected`;
- `affected_mitigated`;
- `affected_accepted`;
- `under_investigation`;
- `fixed`;
- `scheduled_for_remediation`.

For each, define required evidence, who can author, who approves, maximum duration, review triggers, and whether a scanner may filter it from a blocking view. Keep raw findings available in an unfiltered audit view.

Data layer

Use stable identifiers:

- CVE plus advisory aliases;
- purl and supplier/product identifiers;
- SBOM serial/version;
- artifact and image digests;
- source commit;
- environment/deployment identity;
- control and risk-record IDs.

Store evidence once and reference it from VEX, tickets, scanner policy, POA&M;, and the GRC system. CISA's VEX work exists specifically to make product-vulnerability status exchange machine-readable (CISA SBOM Resources Library).

Automation layer

An effective pipeline can:

1. generate an SBOM and provenance for the final artifact;
2. scan the immutable artifact;
3. enrich with vendor advisories, KEV, EPSS, asset context, and exposure;
4. run supported reachability analyses;
5. apply only current, product-matching, approved VEX statements;
6. open or update one canonical risk/remediation record;
7. route approval by residual risk;
8. generate time-bounded scanner policy from the approved disposition;
9. re-evaluate on builds and intelligence changes;
10. revoke expired statements and fail policy when required;
11. retain both raw and dispositioned results.

This changes the unit of work from "write the same exception in every scanner" to "make one evidence-backed disposition and distribute it."

Metrics that reward risk reduction rather than cosmetic cleanliness

Useful measures include:

- exploitable/affected findings by asset criticality;
- KEV exposure and time to required action;
- time from fix availability to verified deployment;
- exception age and renewal count;
- unsupported/EOL component count;
- percentage of dispositions with immutable artifact scope;
- compensating-control test pass/fail and drift;
- percentage of `not_affected` statements revalidated after relevant change;
- components removed and attack paths closed;
- reopened exceptions after threat or deployment change;
- raw-to-dispositioned finding reconciliation.

Avoid using total CVE count or “zero criticals after suppression” as the primary outcome. Those measures can improve when visibility gets worse.

A concise decision procedure

For each flagged CVE:

1. **Is the match wrong?**

If yes, record `false_positive` with package/advisory evidence and correct the matching source where possible.

1. **Is the affected code or required condition absent in this exact product?**

If yes, record `not_affected`, preferably in VEX, with immutable product scope and reproducible evidence.

1. **Can the vulnerable component, feature, endpoint, or asset be removed or disabled?**

If yes, avoid the risk and verify the deployed result.

1. **Is a supported fixed version or vendor mitigation available?**

If yes, implement it within the governing timeframe, rebuild/redeploy, and verify.

1. **Is the issue subject to KEV or another mandatory action/deadline?**

If yes, follow that authority; do not substitute an ordinary review cadence.

1. **Can additional controls meet the requirement and reduce residual risk to tolerance?**

If yes, document, test, monitor, and obtain any required assessor approval.

1. **Does residual risk remain and is acceptance legally/policy permissible?**

If yes, route it to the authorized risk owner with an exit plan, review triggers, and expiration.

1. **If none of the above is acceptable,** discontinue, isolate, replace, or deny authorization for the affected use.

Limitations of this analysis

Compliance outcomes are scope- and jurisdiction-specific. This article uses public NIST, CISA, FedRAMP, PCI SSC, FIRST, Docker, Snyk, and Anchore/Grype sources to establish the general operating model, but it cannot determine whether a particular assessor will accept a particular control without the contract, system boundary, exact requirement, evidence, and assessor procedures. ISO/IEC standards and some commercial assurance criteria are not reproduced here because their normative texts may be paywalled or licensed; an organization using them should map this workflow to its licensed copy and auditor instructions.

Tool capability also changes. The Snyk and Grype descriptions here reflect their live documentation linked above; teams should record the product edition, supported language/integration, feature status, tool version, and analysis date rather than assuming a brand name guarantees a particular reachability method.

Reachability and exploitability assessments are models with uncertainty. Static analysis may miss dynamic behavior; runtime testing observes only exercised behavior; configuration and network evidence can drift; supplier VEX may not match a downstream-modified artifact. These limitations argue for layered evidence and event-driven re-evaluation, not for abandoning contextual analysis.

Finally, structural reduction can reduce the number of installed components and potential attack paths, but no authoritative source supports treating raw image size or CVE count as a complete security metric. The remaining code, configuration, privileges, data, and exposure still require assessment.

Practical meaning for security, platform, and GRC leaders

The most mature position is neither “patch every scanner finding regardless of context” nor “suppress anything not currently reachable.” It is to maintain a trustworthy chain from observation to decision:

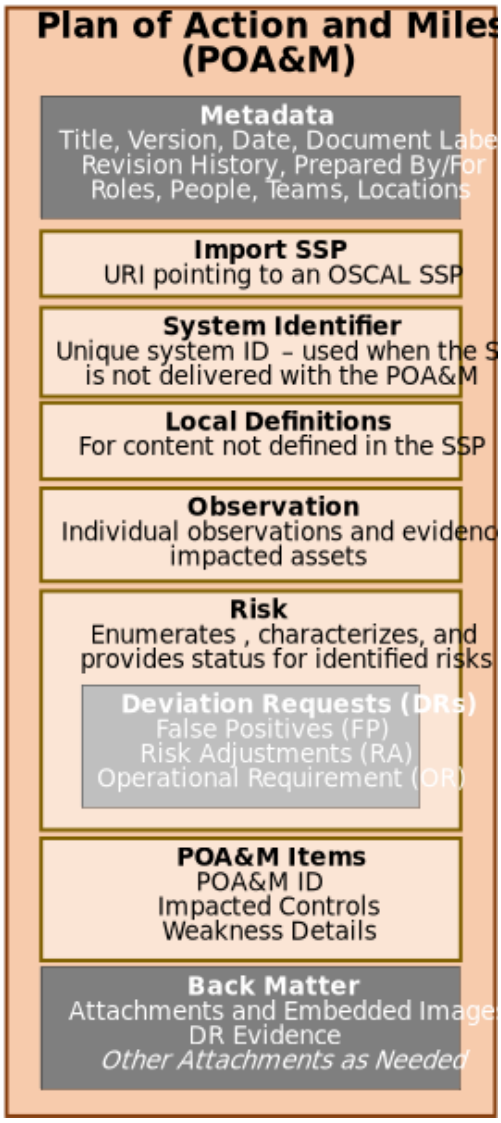
finding → validated identity → affected conditions → deployment context → exploitability/threat/impact → response → tested residual risk → accountable approval → continuous review → verified closure.

Security engineering should own evidence quality and technical analysis. Platform and product teams should own removal, upgrades, rebuilds, and operational controls. Business or mission leaders with delegated authority should own residual-risk decisions. GRC should define the allowed disposition paths, evidence rules, deadlines, and reconciliation. Internal audit should be able to select any filtered finding and reconstruct why it was filtered, who approved that status, what assumptions remain true, and when the decision will be re-evaluated.

When implemented this way, formal acceptance, reachability analysis, and minimal images are not three tricks for passing an audit. They are complementary parts of a governed vulnerability-response system. Formal acceptance provides accountability for residual risk; contextual analysis improves the factual basis and can distinguish genuinely not-affected products; structural reduction removes unnecessary exposure. Automation through SBOM, VEX, immutable artifact identity, policy-as-code, and integrated risk records eliminates repetitive transcription while preserving the evidence an auditor actually needs.

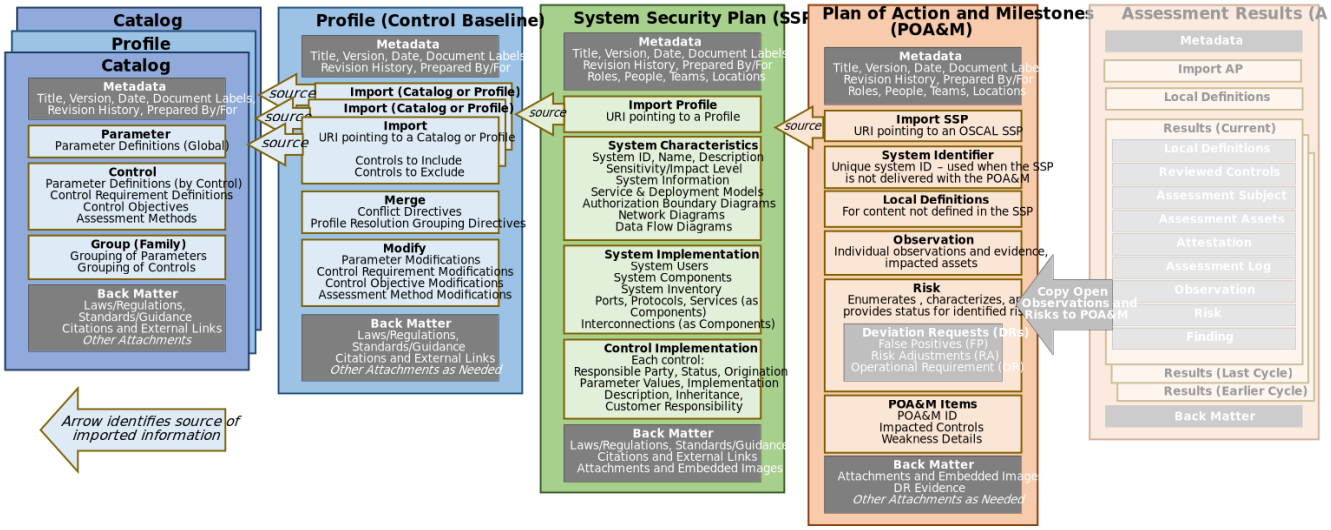
Visual assets

NIST OSCAL POA&M; information model



Credit: NIST OSCAL — Plan of Action and Milestones Model. This source-provided diagram is useful for explaining that a POA&M; is structured evidence—metadata, system association, risks, remediation items, and supporting material—not merely a scanner ignore.

NIST OSCAL stack showing the POA&M;’s relationship to assessment and authorization artifacts



Credit: [NIST OSCAL — Plan of Action and Milestones Model]. The diagram places POA&M; data in the broader machine-readable assessment and authorization flow.

Sources

	Source
	NIST SP 800-40 Rev. 4, *Guide to Enterprise Patch Management Planning*
	NIST SP 800-37 Rev. 2, *Risk Management Framework for Information Systems and Organizations*
	NIST CSRC, “compensating security control”
	NIST OSCAL, “Plan of Action and Milestones Model”
	NIST OSCAL POA&M; JSON definitions
	FedRAMP, “Agency Plans of Action and Milestones”
	PCI SSC FAQ 1597
	PCI SSC, “Do all PCI DSS requirements apply to every system component?”
	PCI SSC, “PCI DSS v4.0: Compensating Controls vs Customized Approach”
	PCI DSS v4.0 SAQ D Merchant, Appendix B
	CISA Known Exploited Vulnerabilities Catalog
	CISA, *Minimum Requirements for Vulnerability Exploitability eXchange (VEX)*
	CISA SBOM Resources Library
	Snyk, “Reachability analysis”
	Anchore Open Source, “Grype”
	Anchore Open Source, “Filter scan results”
	FIRST, “The EPSS Model”
	FIRST, “EPSS Frequently Asked Questions”
	Docker, “Building best practices”
	Docker, “Multi-stage builds”
	Docker, “Minimal or distroless images”
	NIST SP 800-190, *Application Container Security Guide*