

Vulnerability Assessment & Patch Management

Blue Team Context: Vulnerability management is the continuous process of identifying, evaluating, treating, and reporting on security weaknesses in systems and software. Combined with disciplined patch management, it forms the single most effective defense against the majority of real-world attacks. According to multiple industry reports, unpatched known vulnerabilities remain the top initial access vector year after year.

The Vulnerability Management Lifecycle

Effective vulnerability management is not a one-time scan — it is a continuous cycle with four core phases:

1. **Discover** — Identify assets and scan them for known vulnerabilities. You cannot protect what you do not know about. This phase includes asset inventory, network scanning, agent-based detection, and container image analysis. Discovery must cover infrastructure, applications, dependencies, and configurations.
2. **Prioritize** — Not all vulnerabilities are equal. A critical-severity CVE on an internet-facing production server demands immediate action. The same CVE on an isolated development VM behind a firewall may be acceptable risk for weeks. Prioritization combines CVSS scores with business context, exploitability data, and asset criticality.
3. **Remediate** — Apply the fix. This could mean installing a vendor patch, upgrading a dependency, applying a configuration change, or deploying a compensating control (WAF rule, network segmentation) as a temporary mitigation while a full fix is prepared.
4. **Verify** — Confirm that the remediation worked. Rescan the asset, run regression tests, and validate that the vulnerability no longer exists without introducing new issues. Close the ticket and document the outcome.

This cycle repeats continuously. Most mature organizations run discovery scans weekly or daily, with critical findings triggering immediate remediation workflows.

Vulnerability Scanning Tools

Several tools dominate the vulnerability scanning landscape, each with different strengths:

Tool	Type	Best For	License
Nessus	Network / Host scanner	Enterprise infrastructure, compliance audits	Commercial (Essentials free for 16 IPs)
OpenVAS (Greenbone)	Network / Host scanner	Open-source alternative to Nessus	GPL
Qualys VMDR	Cloud-based scanner	Large-scale enterprise, continuous monitoring	Commercial
Trivy	Container / IaC scanner	Container images, filesystems, Kubernetes, IaC	Apache 2.0

OpenVAS CLI Example

OpenVAS can be driven from the command line via `gvm-cli`. Here is a typical workflow for launching a scan:

```
# Authenticate and get a session token
gvm-cli --gmp-username admin --gmp-password secret socket \
  --socketpath /run/gvmd/gvmd.sock \
  --xml ''

# Create a target
gvm-cli socket --socketpath /run/gvmd/gvmd.sock --xml \
  ,
  Web Server Scan
  192.168.1.100
```

```

,

# Create and start a scan task (using Full and Fast scan config)
gvm-cli socket --socketpath /run/gvmd/gvmd.sock --xml \
,
    Weekly Web Server Scan

,

# Start the task
gvm-cli socket --socketpath /run/gvmd/gvmd.sock --xml \
''

```

Trivy for Container Scanning

Trivy has become the de facto standard for scanning container images, filesystems, and Infrastructure as Code templates:

```

# Scan a container image for vulnerabilities
trivy image nginx:1.25

# Scan only for CRITICAL and HIGH severity
trivy image --severity CRITICAL,HIGH python:3.11-slim

# Scan a local project directory for dependency vulnerabilities
trivy fs --scanners vuln /path/to/project

# Scan a Kubernetes cluster
trivy k8s --report summary cluster

# Output results as JSON for pipeline integration
trivy image --format json --output results.json myapp:latest

# Scan IaC files (Terraform, CloudFormation, Dockerfiles)
trivy config ./terraform/

```

CVSS Scoring

The **Common Vulnerability Scoring System (CVSS)** provides a standardized way to assess vulnerability severity. CVSS v3.1 (and its successor v4.0) scores range from 0.0

to 10.0 and consist of three metric groups:

Metric Group	What It Measures	Who Defines It
Base	Intrinsic characteristics: attack vector, complexity, privileges required, user interaction, scope, impact on confidentiality/integrity/availability	Vulnerability publisher (NVD, vendor)
Temporal	Factors that change over time: exploit code maturity, remediation level, report confidence	Security community, updated as exploits appear
Environmental	Organization-specific: modified base metrics reflecting your actual deployment, asset value, and existing mitigations	Your organization

A CVSS base score translates to a qualitative rating:

Score Range	Rating
0.0	None
0.1 – 3.9	Low
4.0 – 6.9	Medium
7.0 – 8.9	High
9.0 – 10.0	Critical

Calculating CVSS in Python

```
from cvss import CVSS3

# CVE-2024-3094 (XZ Utils backdoor) - CVSS vector string
vector = "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H"
score = CVSS3(vector)

print(f"Base Score:  {score.base_score}")    # 10.0
print(f"Severity:    {score.severities()[0]}") # Critical
```

```

print(f"Impact:      {score.isc}")
print(f"Exploitability: {score.esc}")

# Compare multiple CVEs for prioritization
cves = {
    "CVE-2024-3094": "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H",
    "CVE-2023-44487": "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H",
    "CVE-2023-4863": "CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H",
}

print("\n--- CVE Comparison ---")
for cve_id, vec in sorted(cves.items(),
                          key=lambda x: CVSS3(x[1]).base_score,
                          reverse=True):
    s = CVSS3(vec)
    print(f"{cve_id}: {s.base_score:4.1f} ({s.severities()[0]})")

```

CVE Database and NVD

Every publicly disclosed vulnerability receives a **CVE (Common Vulnerabilities and Exposures)** identifier, such as `CVE-2024-3094`. The MITRE Corporation maintains the CVE list, while the **National Vulnerability Database (NVD)**, operated by NIST, enriches each CVE with CVSS scores, affected product lists (CPE), and references.

The NVD API allows programmatic access to vulnerability data:

```

import requests

def query_nvd(cve_id: str) -> dict:
    """Fetch CVE details from the NVD API."""
    url = f"https://services.nvd.nist.gov/rest/json/cves/2.0?cveId={cve_id}"
    resp = requests.get(url, timeout=15)
    resp.raise_for_status()
    data = resp.json()
    vuln = data["vulnerabilities"][0]["cve"]

    description = vuln["descriptions"][0]["value"]
    metrics = vuln.get("metrics", {})
    cvss_data = None
    if "cvssMetricV31" in metrics:
        cvss_data = metrics["cvssMetricV31"][0]["cvssData"]

```

```
return {
    "id": cve_id,
    "description": description,
    "base_score": cvss_data["baseScore"] if cvss_data else "N/A",
    "vector": cvss_data["vectorString"] if cvss_data else "N/A",
    "severity": cvss_data["baseSeverity"] if cvss_data else "N/A",
}

info = query_nvd("CVE-2024-3094")
for k, v in info.items():
```

Prioritization: Beyond CVSS

CVSS base scores are necessary but not sufficient for prioritization. A vulnerability with a base score of 9.8 in a library your application does not actually call is less urgent than a 7.5 that is being actively exploited in the wild against your exposed service. Effective prioritization combines multiple signals:

- **Exploitability:** Is there a public exploit? Is it in exploit kits? Check CISA's Known Exploited Vulnerabilities (KEV) catalog and Exploit-DB.
- **Asset Criticality:** Is the affected system internet-facing? Does it handle sensitive data? Is it a single point of failure?
- **Exposure:** Is the vulnerable service reachable from the internet, or is it behind multiple layers of network controls?
- **Business Impact:** What happens if this system is compromised? Revenue loss, data breach, regulatory penalties?
- **Compensating Controls:** Are there WAF rules, network segmentation, or other mitigations already reducing the effective risk?

Common Mistake: Many teams fall into the trap of treating every Critical-rated CVE with equal urgency. This leads to "alert fatigue" and ironically causes genuinely dangerous vulnerabilities to get lost in the noise. Prioritize by real-world risk, not raw score alone.

Patch Management Strategies

Patching is the primary remediation mechanism, but applying patches to production systems carries its own risks. A structured patch management process minimizes both vulnerability exposure and patch-induced outages:

1. **Testing:** Apply the patch in a dedicated test environment that mirrors production. Run automated tests (unit, integration, smoke) to detect regressions. For critical infrastructure, include performance and compatibility testing.
2. **Staging:** Deploy to a staging environment that receives real-like traffic. Monitor for errors, performance degradation, and unexpected behavior for an appropriate soak period (hours to days depending on criticality).
3. **Rollout:** Use progressive deployment: canary releases (patch 1–5% of fleet first), then rolling updates across availability zones. Monitor error rates and latency at each stage.
4. **Rollback:** Always have a tested rollback plan. For operating system patches, ensure you have snapshots or the ability to uninstall. For application dependencies, pin versions and keep the previous known-good version deployable. Rollback should be a single command or pipeline trigger, not a manual process.

For emergency patches (actively exploited zero-days), organizations may compress this process but should never skip testing entirely. Even a 30-minute smoke test can prevent a patch from causing a worse outage than the vulnerability it fixes.

AI-Assisted Vulnerability Management

LLMs like Claude can accelerate several parts of the vulnerability management lifecycle:

```
import anthropic
import json

client = anthropic.Anthropic()

# Sample scan results (simplified)
scan_results = [
    {"cve": "CVE-2024-3094", "cvss": 10.0, "package": "xz-utils 5.6.0",
     "asset": "build-server-01", "internet_facing": False},
    {"cve": "CVE-2023-44487", "cvss": 7.5, "package": "nginx 1.24",
     "asset": "api-gateway-prod", "internet_facing": True},
```

```

        {"cve": "CVE-2023-4863", "cvss": 8.8, "package": "libwebp 0.5.0",
         "asset": "image-processor-prod", "internet_facing": True},
        {"cve": "CVE-2022-0778", "cvss": 7.5, "package": "openssl 1.1.1n",
         "asset": "internal-wiki", "internet_facing": False},
    ]

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=2048,
    messages=[
        {
            "role": "user",
            "content": (
                "You are a vulnerability management analyst. Given the followi

```

Claude can also generate patch plans, write change management tickets, summarize CVE advisories for non-technical stakeholders, and draft compensating control configurations while waiting for a full patch.

Dependency Scanning

Modern applications have deep dependency trees. A single Python project might pull in hundreds of transitive dependencies, each a potential vulnerability vector. Dependency scanning tools integrate into development workflows to catch vulnerable packages early:

```

# Python - pip-audit (uses OSV and PyPI advisory database)
pip install pip-audit
pip-audit
pip-audit --format json --output audit-results.json

# Python - safety (uses Safety DB)
pip install safety
safety check

# Node.js - built-in npm audit
npm audit
npm audit --json > audit-results.json
npm audit fix # auto-fix compatible updates

# Multi-language - Snyk CLI

```

```
snyk test
snyk monitor # continuous monitoring

# Multi-language - Trivy (also handles OS packages)
trivy fs --scanners vuln .
```

pip-audit in a CI Pipeline

```
"""
Script to run pip-audit and format results for team review.
Integrates with CI pipelines as a quality gate.
"""

import subprocess
import json
import sys

def run_audit() -> list[dict]:
    """Run pip-audit and return parsed results."""
    result = subprocess.run(
        ["pip-audit", "--format", "json", "--desc"],
        capture_output=True, text=True
    )
    if result.returncode == 0:
        print("No vulnerabilities found.")
        return []

    findings = json.loads(result.stdout)
    return findings

def format_report(findings: list[dict]) -> str:
    """Format audit findings into a readable report."""
    lines = [f"{'Package':<25} {'Version':<12} {'CVE':<20} {'Fix':<12}"]
    lines.append("-" * 70)
    for f in findings:
```

Software Bill of Materials (SBOM)

A **Software Bill of Materials (SBOM)** is a complete, machine-readable inventory of every component in a software artifact — libraries, frameworks, OS packages, and their versions. SBOMs are becoming a regulatory requirement (US Executive Order 14028) and are essential for supply chain security.

Two dominant SBOM formats exist: **SPDX** (Linux Foundation) and **CycloneDX** (OWASP). Tools like Syft, Trivy, and the CycloneDX generators can produce SBOMs automatically:

```
# Generate an SBOM for a container image using Syft
syft nginx:1.25 -o cyclonedx-json > sbom.json

# Generate SBOM using Trivy
trivy image --format cyclonedx --output sbom.cdx.json myapp:latest

# Scan an existing SBOM for vulnerabilities using Gype
gype sbom:./sbom.json

# Generate SBOM for a Python project
pip install cyclonedx-bom
cyclonedx-py environment -o sbom.xml
```

Once you have an SBOM, you can continuously monitor it against vulnerability databases without rescanning the entire system. When a new CVE is published, you check it against your SBOM inventory to determine impact within minutes rather than hours.

Sample Vulnerability Report

A structured vulnerability report ties together scanning results, prioritization, and remediation plans:

```
"""Generate a structured vulnerability assessment report."""
import json
from datetime import datetime

report = {
    "report_metadata": {
        "title": "Weekly Vulnerability Assessment Report",
        "date": datetime.now().isoformat(),
        "scope": "Production environment - 142 assets",
        "scanner": "Trivy v0.50 + pip-audit v2.7",
    },
    "summary": {
        "total_vulnerabilities": 47,
        "critical": 3,
        "high": 8,
```

```
    "medium": 21,  
    "low": 15,  
    "assets_scanned": 142,  
    "assets_with_findings": 34,  
  },  
  "top_findings": [  
    {  
      "cve": "CVE-2024-3094",  
      "cvss": 10.0,  
      "package": "xz-utils 5.6.0",  
    }  
  ]  
}
```

Key Takeaways

Vulnerability management is a continuous process, not a periodic event. The organizations that get breached are not the ones that find vulnerabilities — they are the ones that find them and fail to act in time.

- Follow the lifecycle: discover, prioritize, remediate, verify — then repeat.
- Use the right scanner for the job: Nessus/OpenVAS for infrastructure, Trivy for containers and IaC, pip-audit/npm audit for dependencies.
- CVSS scores are a starting point, not the whole picture. Factor in exploitability, asset criticality, and exposure.
- Patch management requires discipline: test, stage, roll out progressively, and always have a rollback plan.
- Generate and maintain SBOMs for all software you build and deploy — they are essential for rapid response when new CVEs drop.
- AI tools like Claude can accelerate prioritization, generate patch plans, and translate technical findings into business risk language.
- Dependency scanning must be integrated into CI/CD pipelines, not run as an afterthought.

Self-check: Can you explain the four phases of the vulnerability management lifecycle? Can you describe why CVSS alone is insufficient for prioritization? Can you run Trivy against a container image and interpret the output? Can you explain what an SBOM is and why it matters for supply chain security?

Download Files

These files contain synthetic vulnerability and patching data for the challenges. All files are plain text (.txt) and safe to open.

↓ [vuln-scan-results.txt](#) ~16 KB

↓ [sbom-sample.txt](#) ~13 KB

↓ [dependency-audit.txt](#) ~10 KB

↓ [patch-inventory.txt](#) ~2 KB