Politecnico di Torino

Politecnico di Torino

MSc in Computer Science Engineering

Automated Vulnerability Assessment and Remediation in Cloud-Native Environments

Candidate: Carlo Bottaro

Supervisors: Prof. Fulvio Risso
Ing. Francesco Pizzato

Academic Year 2024/25

This thesis presents a practical framework for automating vulnerability assessment and remediation in cloud-native environments, with a strong focus on developer-centric workflows and integration within CI/CD pipelines. It investigates the challenges posed by fragmented vulnerability data, inconsistent tooling, and the lack of actionable remediation strategies in modern software supply chains.

At the core of this research is **Vulnbot**, a modular and CI-integrated automation agent that orchestrates vulnerability detection, prioritization, and remediation. Vulnbot supports multiple ecosystems, interfaces with scanners like OSV-Scanner and Trivy, and automates dependency patching and pull request generation, streamlining remediation and reducing mean-time-to-remediation (MTTR).

First, it establishes a foundation in vulnerability databases and their relevance in cloud-native security. Second, it explores how security can be embedded into CI/CD processes using SBOMs, IaC validation, and policy-as-code. Third, it presents automated remediation strategies and best practices. Finally, this thesis contributes with the design of a novel approach, i.e., Vulnbot, for vulnerability remediation automation integrated with development workflows. Its implementation demonstrates how Vulnbot integrates with GitHub Actions, processes vulnerability advisories, and generates remediation pull requests with minimal developer intervention. The presented proof of concept offers insights into the future of automated, policy-driven DevSecOps pipelines.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cloud-Native Security Context

The widespread adoption of cloud-native technologies has fundamentally transformed how organizations design, deploy, and manage their applications. While offering unprecedented flexibility and scalability, this paradigm shift has introduced new security challenges that traditional approaches struggle to address effectively. The dynamic nature of cloud-native environments, characterized by ephemeral containers, microservices architectures, and serverless functions, demands a revolutionary approach to security assessment and vulnerability management [1], [2].

The cloud-native landscape presents unique security challenges that differ significantly from traditional infrastructure environments. Organizations are increasingly adopting containerization technologies, orchestration platforms like Kubernetes, and serverless computing models, creating complex, distributed systems that expand the potential attack surface. This evolution has led to:

- Rapid deployment cycles that can introduce security vulnerabilities at an unprecedented pace.
- Complex dependency chains in containerized applications that increase the potential vulnerability footprint.
- Dynamic scaling and ephemeral workloads that complicate traditional security monitoring approaches.
- Multi-cloud deployments that require consistent security practices across different environments.
- Infrastructure-as-Code (IaC) practices that can propagate misconfigurations across multiple deployments.

Traditional security models, based on periodic assessments and manual interventions, have become inadequate in this new context. Cloud-native environments require security measures that are as dynamic and automated as the infrastructure they protect. Several studies have explored different approaches to vulnerability assessment in cloud environments, including runtime security monitoring, static analysis of container images, and policy-driven enforcement mechanisms. However, many existing solutions lack seamless integration, fail to address remediation challenges, or introduce significant overhead in cloud workloads.

A particular challenge in cloud-native security is the effective integration and management of vulnerability databases. Organizations must continuously monitor and correlate information from multiple sources, each with its own update frequency, data format, and severity scoring system. This complexity is further amplified by the need to maintain real-time awareness of vulnerabilities across rapidly changing cloud environments.

1.2 Importance of Vulnerability Assessment Automation

The automation of vulnerability assessment has become crucial in cloud-native environments for several compelling reasons. First, the speed and scale of modern deployments make manual security assessments impractical and ineffective [3]. Organizations deploy hundreds or thousands of containers daily, each potentially introducing new vulnerabilities through their base images or dependencies.

Automated vulnerability assessment provides:

- Continuous security validation throughout the development and deployment lifecycle.
- Real-time identification of vulnerabilities in both application code and infrastructure components.
- Systematic tracking of security issues across different environments and deployment stages.
- Integration with CI/CD pipelines for early detection and prevention of security issues.
- Standardized assessment processes that reduce human error and ensure consistent security practices.

Beyond scanning for vulnerabilities, automation also plays a crucial role in remediation. Effective security automation should not only detect vulnerabilities but also provide actionable insights and mechanisms for mitigating risks. However, remediation automation presents its own set of challenges, such as:

- Managing **false positives** to prevent unnecessary remediation actions.
- Ensuring that automated patching does not introduce breaking changes.
- Aligning remediation strategies with compliance and governance policies.
- Maintaining system availability while applying security fixes dynamically.

These challenges highlight the need for a structured, well-integrated approach to automated vulnerability assessment and remediation in cloud-native environments.

The effectiveness of remediation efforts heavily depends on the quality and timeliness of vulnerability database information. Organizations must balance multiple factors:

- Accuracy of vulnerability information across different databases.

- Correlation of vulnerability data with actual system components.
- Prioritization of remediation efforts based on severity scores.
- Validation of patches against specific system configurations.

1.3 Thesis Objectives and Contributions

This thesis aims to address the gap between **vulnerability detection and actionable remediation** in modern cloud-native development workflows. As software supply chains grow more complex and fast-paced, traditional vulnerability management practices struggle to scale and keep up. This work proposes a developer-centric, automated approach that integrates vulnerability intelligence, prioritization, and remediation directly into CI/CD pipelines.

The main objectives of this thesis are:

- To analyze the current landscape of vulnerability databases and assess their role in automated remediation.
- To investigate how security practices such as SBOM generation, IaC validation, and policy-as-code can be embedded in CI/CD pipelines.
- To explore the technological and organizational challenges in implementing automated vulnerability remediation at scale.
- To design and implement a practical, extensible tool (**Vulnbot**) that bridges vulnerability scanning tools and automated remediation strategies.
- To evaluate Vulnbot's effectiveness in real-world CI/CD scenarios and identify its current limitations and future directions.

The contributions of this thesis include:

- A comprehensive review of vulnerability intelligence sources (OSV, NVD, GHSA, etc.) and their integration into cloud-native workflows.
- A systematic breakdown of the vulnerability remediation lifecycle, from detection to verification, mapped to automation opportunities.
- A practical implementation of Vulnbot, a modular CI-integrated tool that supports vulnerability scanning, prioritization, patch generation, and pull request creation.
- Guidelines and lessons learned for embedding automated remediation into secure software supply chains.

Chapter 2

Background on Vulnerability Databases and Cloud-Native Security

2.1 Overview of Vulnerability Databases

Vulnerability databases are authoritative repositories that systematically collect, organize, and disseminate information about known security weaknesses across software, operating systems, firmware, and hardware components. They play a foundational role in modern cybersecurity practices by enabling organizations to identify, assess, and respond to threats in a timely and informed manner.

These databases provide standardized metadata about each vulnerability such as unique identifiers, severity scores, affected components, exploit availability, and remediation guidance allowing for consistent integration with security tools, vulnerability scanners, and automation workflows. Examples include the National Vulnerability Database (NVD) [4], maintained by NIST [5], and vendor-specific feeds like Red Hat's OVAL [6] or GitHub Security Advisories [7].

By aggregating data from coordinated disclosures, security researchers, software vendors, and bug bounty programs, vulnerability databases serve as a single source of truth for risk assessment. They facilitate the tracking of Common Vulnerabilities and Exposures (CVEs), which are standardized identifiers for publicly known security flaws [8]. Each CVE provides a unique reference for a specific vulnerability (e.g. CVE-2025-12345). The Common Vulnerability Scoring System (CVSS) offers a numerical severity score (from 0 to 10) that helps prioritize remediation based on exploitability and impact. Additionally, Common Platform Enumeration (CPE) provides a structured naming scheme to identify affected software, hardware, or firmware, enabling precise matching between vulnerabilities and system components.

In cloud-native environments characterized by rapid software iteration, containerized applications, and ephemeral infrastructure, the ability to consume and act upon accurate vulnerability intelligence in real time becomes even more critical. Integrating these databases into automated security pipelines enables continuous monitoring, contextual prioritization, and effective remediation of security flaws across dynamic environments.

Ultimately, the strategic use of vulnerability databases helps organizations shift from reactive to proactive security postures, aligning technical risk management with business continuity and compliance requirements.

2.1.1 Purpose and Functionality

Vulnerability databases perform several key functions that contribute to modern security operations:

- **Identification of Vulnerabilities**: They assign unique identifiers and maintain structured records of vulnerabilities, often including severity ratings, affected systems, and potential mitigations.
- **Standardization and Classification**: Vulnerabilities are categorized using standard frameworks such as the Common Vulnerabilities and Exposures (CVE) [8] system and the Common Vulnerability Scoring System (CVSS) [9].
- **Automated Security Tools Integration**: Security scanners, compliance tools, and threat intelligence platforms integrate with these databases to automate vulnerability detection and assessment.
- **Timely Updates and Feeds**: Security researchers and organizations rely on vulnerability databases for up-to-date information, ensuring rapid response to emerging threats.
- **Risk Assessment and Prioritization**: By providing impact analysis and contextual data, these repositories help organizations prioritize remediation efforts based on risk severity.

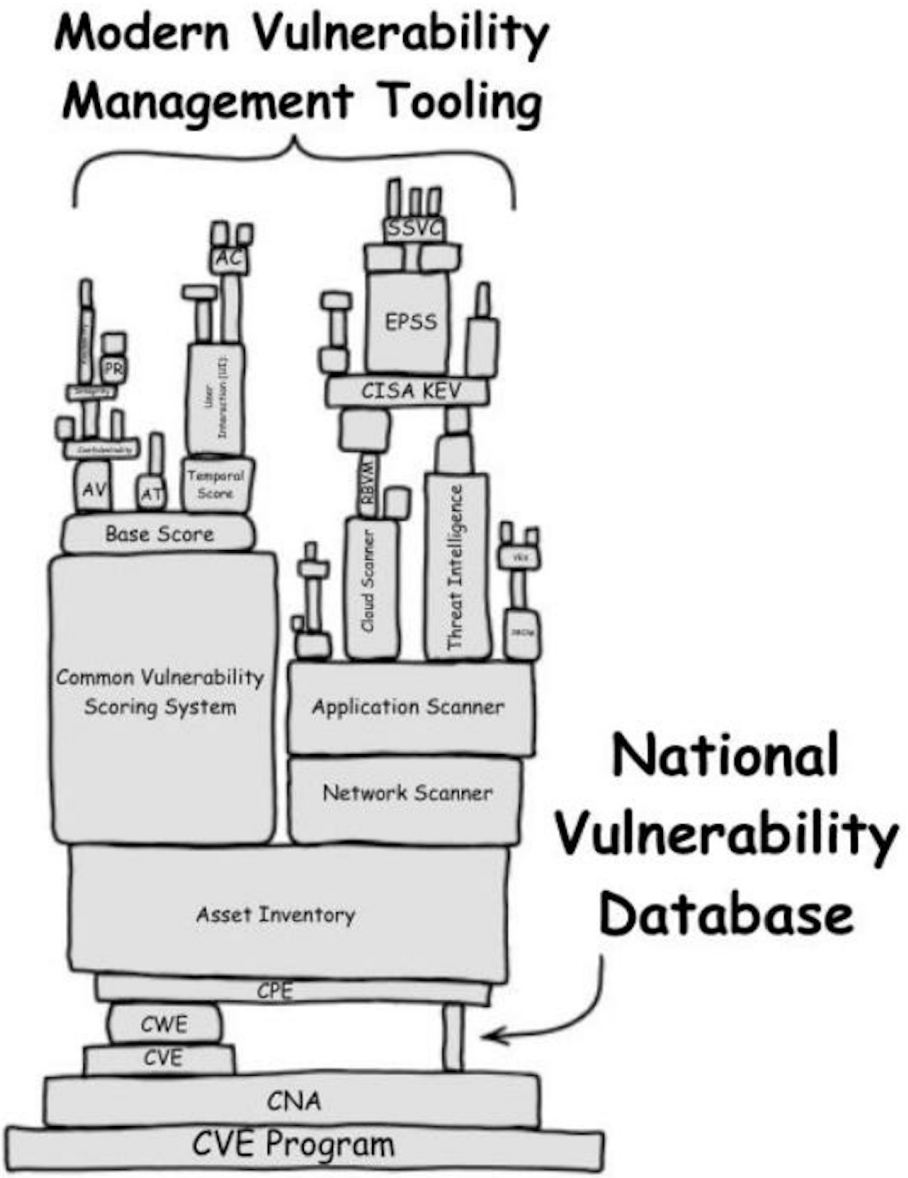
2.1.2 National Vulnerability Database (NVD)

The National Vulnerability Database (NVD) [4] is the U.S. government's repository for managing publicly disclosed cybersecurity vulnerabilities. It is maintained by the National Institute of Standards and Technology (NIST) [5] and plays a critical role in vulnerability standardization and dissemination. Key features include:

- **CVSS Scoring**: Vulnerabilities are assessed using the Common Vulnerability Scoring System (CVSS) to provide a standardized severity rating.
- **CPE Matching**: The Common Platform Enumeration (CPE) [10] system is used to identify affected software and hardware components.
- **Detailed Metadata**: NVD entries contain information on exploitability, impact, and potential mitigation strategies.
- **Machine-Readable Feeds**: JSON and XML feeds allow security tools to ingest and process vulnerability data automatically.
- **Patch and Update Tracking**: Information on remediation measures and vendor advisories is included.

2.1.3 Common Vulnerabilities and Exposures (CVE)

The Common Vulnerabilities and Exposures (CVE) system is a globally recognized framework for identifying and naming publicly disclosed cybersecurity vulnerabilities [8]. Managed by the MITRE Corporation [11], CVE provides a standardized method for referencing vulnerabilities, ensuring consistency across vulnerability databases, security tools, and organizational workflows.



- **Unique Identifiers**: Each vulnerability is assigned a unique CVE ID (e.g., `CVE-2023-12345`), which allows consistent referencing in patch notes, threat reports, and vulnerability scanners.
- **Decentralized Reporting via CNAs**: The CVE system relies on a federated model with over 200 CVE Numbering Authorities (CNAs), including software vendors, open-source projects, and government agencies, who are authorized to assign CVE identifiers.
- **Structured Entries**: CVE records include a concise vulnerability summary, affected products, associated attack vectors, and references to advisories or exploit databases.
- **Ecosystem Integration**: CVE IDs are used by numerous security tools and databases (e.g., NVD, GitHub Security Advisories, Red Hat Security Data) to aggregate further analysis, such as severity scoring (CVSS), exploit availability, and remediation instructions.

A simplified CVE lifecycle typically involves vulnerability discovery, CVE ID assignment by a CNA, publication on the CVE List, and further enrichment by external

databases such as the NVD. This structured process ensures global coordination and timely dissemination of vulnerability information, empowering organizations to take informed action.

2.1.4 Open Source Vulnerabilities (OSV)

The Open Source Vulnerabilities (OSV) database https://osv.dev/, developed by Google, is a specialized resource designed to track security vulnerabilities in open-source software packages [12]. It offers several advantages over traditional vulnerability databases:

- **Comprehensive Package Ecosystem Coverage**: OSV includes major language-specific package managers such as npm (JavaScript), PyPI (Python), Go Modules, and RubyGems.
- **Machine-Readable JSON Format**: OSV stores vulnerability data in a structured JSON schema, facilitating seamless integration with automated security tooling and CI/CD pipelines.
- **Precise Versioning with Semantic Version Ranges**: Instead of broad vulnerability listings, OSV provides detailed version ranges using semantic versioning, improving accuracy in vulnerability detection.
- **Direct Contributions from Maintainers**: Many vulnerability reports are submitted and verified by the maintainers of the affected open-source projects, enhancing reliability and timeliness.

This focus on open-source software security makes OSV a valuable tool for developers and organizations aiming to improve their supply chain security and reduce risks from third-party dependencies.

Open Source Vulnerability format

The OSV database leverages the OSV Schema, an open standard developed by the Open Source Security Foundation (OSSF) to represent vulnerability data in a structured and machine-readable format [13]. The schema is designed specifically for open-source software vulnerabilities and captures essential information such as affected packages, version ranges, severity ratings, and detailed references to commits, advisories, and patches. By adopting a JSON-based schema tailored for package ecosystems, OSV Schema improves interoperability across tools and enhances automated vulnerability detection workflows. This focused structure allows precise vulnerability descriptions aligned with semantic versioning, which reduces false positives and improves security tooling integration in modern software supply chains.

An example of the OSV schema structure is illustrated below 2.1, more info https://github.com/ossf/osv-schema/blob/main/validation/schema.json:

2.1.5 OS-Specific Vulnerability Feeds

In addition to global vulnerability databases, many operating systems maintain their own internal vulnerability tracking and advisory feeds. These feeds provide timely security updates and patches tailored to specific distributions and environments:

```
{
        "schema_version": string,
        "id": string,
        "modified": string,
        "published": string,
        "withdrawn": string,
        "aliases": [ string ],
        "upstream": [ string ],
        "related": [ string ],
        "summary": string,
        "details": string,
        "severity": [ {
                "type": string,
                "score": string
        } ],
        "affected": [ {
                "package": {
                        "ecosystem": string,
                        "name": string,
                        "purl": string
                },
                "severity": [ {
                        "type": string,
                        "score": string
                } ],
                "ranges": [ {
                        "type": string,
                        "repo": string,
                        "events": [ {
                                "introduced": string,
                                "fixed": string,
                                "last_affected": string,
                                "limit": string
                        } ],
                        "database_specific": { see description }
                } ],
                "versions": [ string ],
                "ecosystem_specific": { see description },
                "database_specific": { see description }
        } ],
        "references": [ {
                "type": string,
                "url": string
        } ],
        "credits": [ {
                "name": string,
                "contact": [ string ],
                "type": string
        } ],
        "database_specific": { see description }
}
```


Figure 2.1: OSV schema structure

- **Debian Security Tracker**: Monitors security issues affecting Debian packages and integrates with the Debian package management system [14].
- **Ubuntu CVE Tracker**: Maintained by Canonical, it provides vulnerability assessments and patch details specific to Ubuntu distributions [15].
- **Red Hat Security Data API**: Offers vulnerability data and security advisories for Red Hat Enterprise Linux (RHEL), CentOS, and Fedora [16].
- **Microsoft Security Response Center (MSRC)**: Microsoft issues security bulletins and patches through its monthly "Patch Tuesday" releases, ensuring Windows and related software remain protected [17].

Database	Update Frequency	Integration Method
Debian Security	Real-time	APT hooks, JSON API
Red Hat Security	Daily	REST API, OVAL definitions
Ubuntu CVE Tracker	Daily	Launchpad API
MSRC	Monthly (Patch Tuesday)	RSS feeds, MS Graph API

Table 2.1: Comparison of vulnerability databases

2.1.6 Vulnerability Exploitability eXchange (VEX) and Open Vulnerability and Assessment Language (OVAL)

In the context of vulnerability management, standardized data formats and communication protocols play a critical role in automating and streamlining security assessments. Two important standards that facilitate this are the Vulnerability Exploitability eXchange (VEX) and the Open Vulnerability and Assessment Language (OVAL).

Vulnerability Exploitability eXchange (VEX) VEX is a recently developed specification designed to communicate the exploitability status of vulnerabilities within specific products or environments [18]. Unlike traditional vulnerability databases that list all known vulnerabilities, VEX focuses on indicating whether a particular vulnerability is exploitable in a given context. This enables organizations to prioritize remediation efforts and reduce noise from vulnerabilities that do not affect their systems.

Key features of VEX include:

- **Contextualized Vulnerability Information**: VEX documents specify whether a vulnerability affects a particular product version or configuration.
- **Machine-Readable Format**: Typically expressed in JSON or XML, allowing integration with automated security tools and pipelines.
- **Reduction of False Positives**: By explicitly stating exploitability status, VEX helps security teams focus on relevant issues.
- **Industry Adoption**: Supported by vendors and open standards groups, VEX is gaining traction in supply chain security and vulnerability management workflows.

Open Vulnerability and Assessment Language (OVAL) OVAL is a community-driven, standardized language developed to encode vulnerability, configuration, and patch assessment information [15], [16]. It allows security tools to perform automated checks on systems and software to detect vulnerabilities and misconfigurations.

Important aspects of OVAL include:

- **Structured Definitions**: OVAL uses XML schemas to define how to check for specific vulnerabilities, system configurations, and patches.
- **Interoperability**: Many operating system vendors, including Red Hat and Ubuntu, publish vulnerability and compliance data using OVAL definitions.
- **Assessment Automation**: Security scanners utilize OVAL to programmatically query system states and determine vulnerability presence.
- **Extensibility**: OVAL can describe a wide range of security-related checks beyond vulnerabilities, such as compliance with security policies.

Together, VEX and OVAL contribute to more accurate, actionable, and automated vulnerability management by enabling richer contextual information and machine-readable assessments, especially important in complex and dynamic environments like cloud-native infrastructures.

2.2 Related Work in Vulnerability Management

The field of automated vulnerability management has seen significant academic and industrial research over the past decade. This section examines key contributions and positions our work within the broader research landscape.

2.2.1 Academic Research in Vulnerability Assessment

Several foundational studies have explored different aspects of vulnerability management in cloud-native environments. Scandariato et al. [19] conducted a systematic literature review of vulnerability assessment techniques in cloud computing, identifying key gaps in automated remediation capabilities. Their work highlighted the need for better integration between vulnerability detection and response mechanisms.

Research by Chen and Williams [20] specifically focused on dependency vulnerability management in modern software projects. They proposed a graph-based approach to analyze transitive dependencies and predict vulnerability propagation. While their work provided valuable insights into dependency relationships, it lacked practical implementation for automated remediation.

The MITRE Corporation's work on the Common Weakness Enumeration (CWE) system has provided foundational categorization frameworks for vulnerabilities [21]. However, these classification systems primarily focus on vulnerability description rather than remediation automation.

2.2.2 Industry Solutions and Approaches

Commercial vulnerability management platforms have evolved significantly in recent years. Snyk's approach to developer-first security [22] emphasizes integration with developer workflows, similar to our approach with Vulnbot. However, their solution is primarily SaaS-based and lacks the scanner-agnostic architecture we propose.

WhiteSource (now Mend) has pioneered automated dependency updates [23], but their approach is tightly coupled to their proprietary scanning engine. Our work extends this concept by providing a modular, scanner-independent solution.

GitHub's Dependabot [24] represents one of the most widely adopted automated dependency update systems. While effective for basic scenarios, it lacks support for multi-ecosystem vulnerability scanner and it only support few programming languages. Our research aims to fill these gaps by providing a more extensible and comprehensive framework.

2.2.3 Gap Analysis and Our Contribution

While existing research has made significant contributions to individual aspects of vulnerability management, a comparative analysis reveals distinct gaps across academic research, industry solutions, and our proposed approach. Table 2.2 provides a structured comparison of these different approaches.

Table 2.2: Gap Analysis: Academic vs Industry vs Vulnbot Approach

Aspect	Academic Research	Industry Solutions	Vulnbot Approach
Solution Scope	Focused on specific aspects (detection, prioritization, or remediation)	End-to-end but vendor-locked	Comprehensive, scanner-agnostic end-to-end automation
Scanner Integration	Assumes specific scanners or synthetic datasets	Proprietary, single-vendor solutions	Modular, multi-scanner architecture with pluggable adapters
Developer Experience	Limited consideration of human factors and workflow integration	Tool-centric rather than developer-centric	Developer-first design with seamless CI/CD integration
Deployment Model	Research prototypes not production-ready	SaaS-dependent or enterprise-only	Flexible: cloud, on-premises, or hybrid deployment
Customization	Highly customizable but complex	Limited to vendor roadmap	Open-source with modular extensibility

Our work addresses these identified gaps through Vulnbot's comprehensive approach that combines scanner-agnostic architecture, vulnerability prioritization, and seamless CI/CD integration. This positions our contribution as a significant advancement in bridging the gap between academic research and practical application of automated vulnerability remediation.

2.2.4 Data Synchronization

Modern vulnerability management systems must address several challenges in integrating with diverse vulnerability databases. One of the primary concerns is the variation in update frequency among different data sources. For instance, the National Vulnerability Database (NVD) typically provides daily updates, whereas other databases may refresh their data less frequently. Ensuring that security systems ingest the most recent vulnerability information in a timely manner is essential for maintaining an accurate security posture.

Another key challenge is ensuring data consistency across sources. Vulnerabilities might be reported in one database but absent in another, leading to discrepancies that could hinder reliable vulnerability assessment. Organizations must therefore adopt mechanisms to cross-check and validate data from multiple sources to improve the trustworthiness of the collected information.

Furthermore, vulnerability databases often use different schemas and formats such as JSON, XML, or CSV. To enable seamless integration, it becomes necessary to normalize these diverse formats into a unified structure. Standardized formats, like those discussed in Section 2.1.6, can facilitate this process and improve interoperability.

The dynamic nature of vulnerability data also requires attention to historical changes. Vulnerabilities may be reclassified or updated over time, and maintaining a record of these changes is crucial for long-term risk analysis and compliance reporting.

Lastly, real-time awareness of newly disclosed vulnerabilities is increasingly important. Organizations benefit from event-driven update mechanisms or scheduled synchronization strategies that ensure newly published vulnerabilities are quickly reflected in security tools. This reduces the time between discovery and remediation, enhancing the effectiveness of security operations.

2.2.5 Data Quality Considerations

High-quality vulnerability data is fundamental for reducing false positives, improving the accuracy of detection, and providing actionable insights to security teams. One of the main issues arises from false positives, where vulnerabilities are flagged as relevant to a system even though they are not exploitable in its specific context. This can be mitigated through the use of refined matching algorithms and filtering mechanisms.

Equally important is assessing the contextual relevance of each vulnerability. Not all vulnerabilities pose the same risk in every environment; factors such as the specific software configuration, the runtime environment, and dependency relationships all influence whether a particular security issue is truly exploitable.

Version accuracy plays a vital role in this evaluation. Inaccuracies in identifying affected software versions can either cause false negatives, where real threats are overlooked, or excessive reporting that overwhelms analysts with irrelevant alerts. Proper version tracking mechanisms are therefore essential.

An accurate assessment of the impact of each vulnerability, including severity ratings like CVSS scores and exploitability metrics, helps prioritize remediation efforts. Without this, organizations may misallocate resources to lower-risk issues while neglecting critical threats.

Lastly, high-quality remediation guidance is indispensable. Effective vulnerability management requires not only detection but also actionable information about how to remediate the issue. This includes the availability of patches, suggested workarounds, and vendor-specific recommendations, which collectively support faster and more effective resolution.

2.2.6 Integration Patterns

Integrating vulnerability data into security systems can follow various architectural patterns, each with its own strengths and limitations. One common approach is API-based integration, where security tools query external vulnerability databases in real time. This ensures access to the most up-to-date information but depends on reliable internet connectivity and efficient querying strategies, especially when dealing with high volumes of data.

Alternatively, some organizations opt to maintain local mirrors of vulnerability databases. This approach reduces reliance on external sources and can improve performance, particularly in restricted or air-gapped environments. However, it requires a robust synchronization process to ensure the mirrored data remains current.

Event-driven integration represents a more proactive model. Some databases offer webhook notifications or support streaming updates, allowing security systems to receive immediate alerts whenever new vulnerabilities are published. This minimizes the delay between publication and detection, enabling faster response times.

In practice, many organizations adopt hybrid strategies that combine these approaches. For example, real-time API queries may be used for critical services, while a local mirror supports broader historical analysis. This hybrid model balances the need for immediacy, accuracy, and scalability in vulnerability data integration.

2.3 Vulnerability Data in Cloud-Native Contexts

While CVE provides global identifiers and OSV offers ecosystem-specific intelligence, cloud-native environments present fundamentally different challenges that traditional vulnerability databases were not designed to address. The ephemeral nature of containers, the complexity of layered filesystems, and the rapid drift of Infrastructure as Code (IaC) configurations create a dynamic threat landscape that requires specialized approaches to vulnerability tracking and remediation.

Cloud-native environments pose unique challenges to vulnerability management due to their highly dynamic nature, reliance on containerized workloads, and widespread use of Infrastructure as Code (IaC). Unlike traditional monolithic applications that remain relatively static once deployed, cloud-native applications are characterized by:

- **Ephemeral infrastructure**: Containers and serverless functions that exist for minutes or hours rather than months or years, making persistent vulnerability tracking challenging
- **Immutable deployments**: Infrastructure is replaced rather than updated, requiring vulnerability assessment to occur before deployment rather than post-deployment patching
- **Complex dependency graphs**: Microservices architectures create intricate webs of service dependencies that can amplify the impact of individual vulnerabilities
- **Configuration drift**: IaC templates may diverge from actual deployed infrastructure, creating gaps in vulnerability coverage

To effectively detect and mitigate vulnerabilities in such settings, security strategies must evolve to accommodate these new paradigms, moving from reactive patching to proactive prevention integrated into the deployment pipeline.

2.3.1 Container-Specific Considerations

Containerized applications package software and dependencies together, introducing new layers of complexity in security analysis. One critical aspect is the **layered structure** of container images. Each instruction in a Dockerfile produces a new filesystem layer, which is combined with others using a union filesystem such as OverlayFS to form the final container image. Vulnerabilities can be introduced in any of these layers, and determining whether a vulnerability remains in the final image often requires deep inspection2.3.2.

For instance, if a vulnerable package is installed in one layer and removed or updated in a subsequent layer, a simplistic scanner might incorrectly flag the image as vulnerable. Accurate scanning tools must therefore reconstruct the entire layered filesystem, track the creation and deletion of files, and correlate package

installation and removal events with the appropriate layers. This prevents false positives, particularly in complex images with multiple intermediate changes.

Moreover, the choice of base images significantly influences container security. Many container images inherit from public base images like Alpine, Debian, or Ubuntu, making it essential to track vulnerabilities in these upstream sources as well. The inclusion of system dependencies through package managers such as `apt`, `yum`, or `apk` adds another dimension to this analysis.

The runtime context of containers also matters. Even if a vulnerability exists in the image, its exploitability might depend on runtime factors such as container privileges, network settings, or security mechanisms like AppArmor, SELinux, or seccomp profiles. Similarly, orchestrators like Kubernetes introduce their own security controls such as role-based access control (RBAC), network policies, and admission controllers which must be considered when evaluating vulnerability exposure.

2.3.2 Container Layers and Layer Analysis

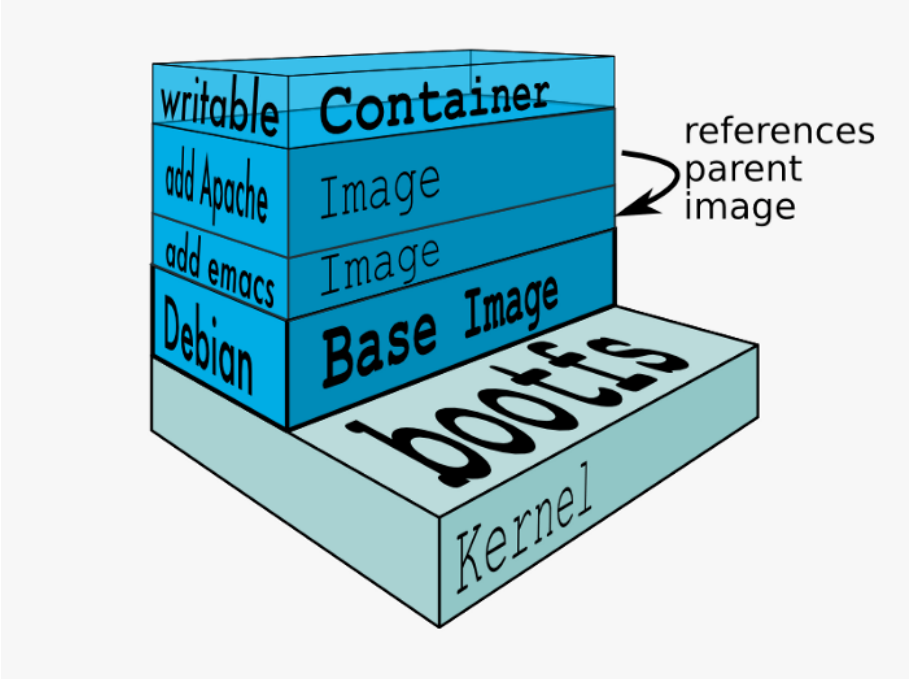


Figure 2.2: Union Filesystem layering example

Container images are built incrementally through layers, each representing a snapshot of filesystem changes. For example, a simple Dockerfile might begin with a Debian base image, install a package like `curl`, copy application files, and set permissions. Each of these steps creates a new layer. While these layers improve build efficiency and reusability, they complicate vulnerability detection because changes made in one layer can be altered or reversed in a subsequent one.

A vulnerability introduced when installing a package might be remediated later in the build process. However, without full image analysis, a scanner might still report the vulnerability, resulting in a false positive. A precise vulnerability assessment must reconstruct the full layered filesystem, track file changes across layers, and analyze the sequence of package operations to determine the actual state of the final image.

Consider the case where a binary containing a known vulnerability is copied into the image in one layer and then removed in the next. A scanner that only inspects individual layers might report the binary's presence, even though it no longer exists in the final image. This underlines the need for comprehensive and context-aware analysis.

2.3.3 Infrastructure as Code (IaC) Security

Infrastructure as Code (IaC) enables automated and consistent infrastructure provisioning through configuration files. Tools such as Terraform, CloudFormation, and Ansible allow teams to define their infrastructure declaratively. However, misconfigurations in these templates can introduce significant security risks if left unchecked.

Analyzing IaC templates for insecure defaults or risky configurations such as publicly accessible cloud storage, missing encryption, or overly permissive access controls is essential. Automated scanners can detect such issues early, ideally before deployment occurs. To enforce best practices, organizations can leverage policy-as-code frameworks like Open Policy Agent (OPA) or HashiCorp Sentinel, which evaluate IaC configurations against predefined security policies during the CI/CD process.

Security integration with version control systems such as GitHub and GitLab further strengthens IaC practices. By embedding security checks directly into CI/CD pipelines, teams can catch vulnerabilities and misconfigurations at the earliest stages of the development lifecycle.

Moreover, managing secrets within IaC files is a critical concern. Exposing hardcoded credentials in configuration files poses a major threat. Tools like HashiCorp Vault and AWS Secrets Manager help mitigate this risk by enabling secure handling and storage of sensitive information.

2.3.4 Future Trends

As cloud-native architectures continue to evolve, several trends are emerging to improve vulnerability management. One such trend is the integration of artificial intelligence and machine learning to enhance the prioritization of vulnerabilities. These technologies can identify patterns in historical data, predict exploitability, and support intelligent triage of alerts based on contextual risk.

Supply chain security is also gaining prominence, particularly in response to attacks that compromise software dependencies. Organizations are increasingly adopting practices such as dependency verification, signed container images, and the use of Software Bill of Materials (SBOMs) to improve transparency and traceability.

To manage the overwhelming volume of vulnerability alerts, automated triage systems are being developed. These systems analyze contextual signals such as whether a vulnerable component is actually used at runtime to prioritize the most critical issues and reduce alert fatigue.

Given the diversity of cloud platforms, runtimes, and deployment models, a unified view across heterogeneous environments is becoming essential. Cross-platform correlation tools aim to consolidate vulnerability tracking and enhance visibility across all systems, regardless of where or how they are deployed.

Finally, the industry is embracing a shift-left security philosophy, where vulnerability detection is integrated into the earliest phases of software development. By embedding security checks into the developer workflow, organizations can identify

and fix issues before they reach production, resulting in more secure and resilient systems.

By addressing these emerging challenges and adopting innovative approaches, organizations can enhance their ability to manage vulnerabilities across complex, cloud-native environments. The discussed standards and databases form the foundation upon which security can be embedded in CI/CD pipelines. The next chapter will explore how these vulnerability intelligence sources are operationalized in practice, examining the integration patterns, scanning strategies, and automation frameworks that transform raw vulnerability data into actionable security measures within modern development workflows.

Chapter 3

Embedding Security in CI/CD Pipelines

3.1 CI/CD Scanning Pipelines

The integration of security scanning within Continuous Integration and Continuous Deployment (CI/CD) pipelines represents a fundamental shift in how organizations approach security in cloud-native environments. Rather than treating security as a separate, post-deployment activity, modern approaches embed security checks throughout the development and deployment lifecycle. This "shift-left" approach allows for early detection and remediation of vulnerabilities, significantly reducing the cost and impact of security issues [25].

3.1.1 Pipeline Integration Points

A comprehensive CI/CD security scanning strategy involves multiple integration points, each serving distinct purposes and providing different security guarantees. These integration points correspond to different stages in the software development lifecycle and allow for progressive security validation.

Pre-commit Stage

The pre-commit stage represents the earliest opportunity to identify and address security issues. At this stage, developers run scans on their local workstations before

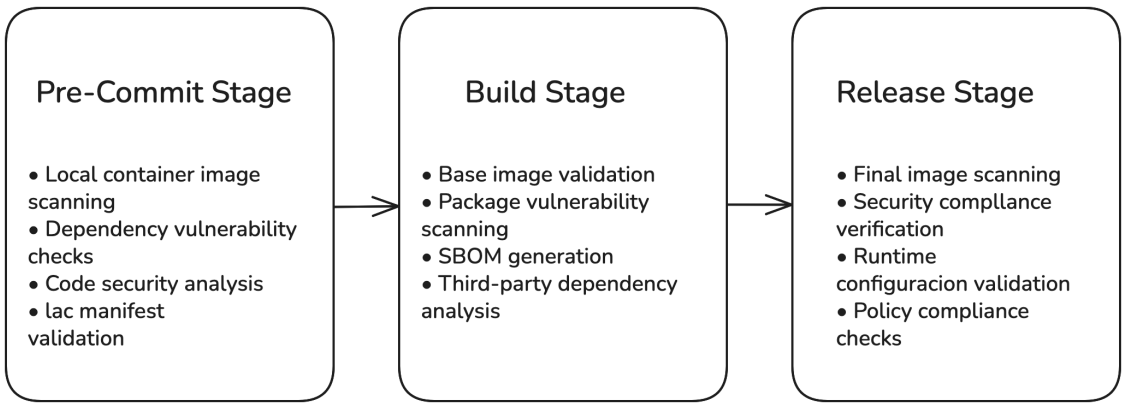


Figure 3.1: CI/CD Security Integration

committing code to the repository.

- **Local container image scanning**: Developers can use lightweight scanning tools like Trivy or Grype to perform quick assessments of container images before pushing them to registries. These tools can identify vulnerabilities in the container's base image and installed packages.
- **Dependency vulnerability checks**: Tools like npm audit, OWASP Dependency-Check, or Snyk can analyze application dependencies to identify known vulnerabilities in third-party libraries. This is particularly important for modern applications that often rely on dozens or hundreds of external dependencies.
- **Code security analysis**: Static Application Security Testing (SAST) tools like SonarQube, Checkmarx, or open-source alternatives can analyze source code to identify potential security weaknesses, such as SQL injection vulnerabilities, cross-site scripting (XSS) issues, or improper input validation.
- **IaC manifest validation**: Infrastructure as Code (IaC) scanners like Checkov, Terrascan, or Snyk IaC can verify that infrastructure manifests (e.g., Terraform configurations, Kubernetes YAML files) adhere to security best practices and do not introduce misconfigurations.

Build Stage

The build stage represents the next critical integration point, where more comprehensive scanning can be performed after code is committed to the repository. This stage typically runs in a CI/CD environment and can include more resource-intensive scans.

- **Base image validation**: Automated processes can verify that container base images are from trusted sources and do not contain known vulnerabilities. This can include checks against allow-lists of approved base images or dynamic scanning of images during the build process.
- **Package vulnerability scanning**: More comprehensive scanning of installed packages and dependencies can be performed, including transitive dependencies and system-level packages. Tools like Anchore, Clair, or Trivy can generate detailed reports of vulnerabilities in the built image.
- **SBOM (Software Bill of Materials) generation**: Tools like Syft, Anchore, or Microsoft's SBOM Tool can generate a comprehensive inventory of all components and dependencies in the built artifacts. This SBOM can be stored alongside the built artifacts for future reference and continuous monitoring [26].
- **Third-party dependency analysis**: Beyond simple vulnerability checks, more advanced analysis can be performed to identify dependency issues such as license compliance problems, outdated libraries, or packages with known maintenance issues.

Build-stage scanning can be configured to fail the build if critical issues are detected, preventing vulnerable artifacts from proceeding to later stages.

3.2 Software Bill of Materials (SBOM)

A Software Bill of Materials (SBOM) is a detailed inventory of all components libraries, dependencies, and modules included within a software artifact. Originally inspired by the manufacturing industry's concept of a bill of materials for physical goods, SBOMs have become a critical security asset in the software supply chain, particularly in cloud-native environments.

The adoption of SBOMs aligns with the growing need for transparency and accountability in modern software development. By providing visibility into the makeup of software, SBOMs enable organizations to track and manage vulnerabilities, assess risk, and comply with regulatory and licensing requirements [26]. In CI/CD pipelines, SBOMs act as a foundational element for secure development practices.

SBOMs not only improve transparency but also serve as the substrate for automated remediation workflows. When new CVEs are disclosed, stored SBOMs can be continuously re-scanned, triggering targeted remediation actions without requiring full re-analysis of source code. This creates a powerful feedback loop where vulnerability intelligence from databases like NVD and OSV is automatically correlated against known software inventories, enabling precise identification of affected systems and automated generation of remediation pull requests for impacted repositories.

3.2.1 SBOM Standards and Formats

To ensure interoperability and consistency, several standardized formats have emerged for SBOM generation and consumption:

- **SPDX (Software Package Data Exchange)**: Developed by the Linux Foundation, SPDX is a comprehensive, standardized format that enables the sharing of software bill of materials (SBOM) data across organizations and tools. It is designed to improve transparency in the software supply chain by capturing information about software packages, their licenses, relationships, and file-level metadata. SPDX supports both human- and machine-readable formats, including JSON, YAML, RDF, and tag-value formats. The specification defines over 30 fields for describing software components, including license expressions, cryptographic checksums, and document relationships (e.g., '$CONTAINS$', '$DEPENDS\_ON$', '$GENERATED\_FROM$'). SPDX has been officially recognized by ISO as an international standard (ISO/IEC 5962:2021) [27], and it is increasingly used by open-source projects, vendors, and government agencies to ensure license compliance and enable vulnerability correlation [28].

- **CycloneDX**: Created by the OWASP Foundation, CycloneDX is a lightweight SBOM specification designed specifically with security use cases in mind. It supports detailed descriptions of components, services, vulnerabilities, and dependencies. CycloneDX also includes support for cryptographic integrity verification, pedigree (component lineage), and external references such as VEX (Vulnerability Exploitability eXchange) documents. The format is optimized for automated processing and is widely integrated into security tools, including dependency scanners, vulnerability management platforms, and CI/CD pipelines. CycloneDX supports multiple serialization formats (XML, JSON, and Protocol Buffers) and is maintained with a strong focus on DevSecOps workflows, making it a natural choice for teams embedding security into their development lifecycle [29].

Table 3.1: Comparison between SPDX and CycloneDX

Feature	SPDX	CycloneDX
Developed by	Linux Foundation	OWASP Foundation
Primary Focus	License compliance, provenance, supply chain transparency	Security, vulnerability management, and dependency tracking
Serialization Formats	Tag-Value, JSON, YAML, RDF	XML, JSON, Protocol Buffers
Specification Complexity	Comprehensive and detailed	Lightweight and security-focused
Standards Recognition	ISO/IEC 5962:2021	De facto industry standard for DevSecOps
Key Use Cases	License auditing, legal documentation, software provenance	SBOM for security tools, CI/CD integration, VEX support
Tooling Ecosystem	Widely supported in legal and OSS license tools (e.g., SPDX tools, FOSSology)	Integrated with modern security tools (e.g., Trivy, Dependency-Track, Jenkins plugins)

Each format includes metadata such as package names, versions, hashes, licenses, and relationships between components. The choice of format often depends on the intended use case and integration tooling.

3.2.2 SBOM Generation in CI/CD Pipelines

SBOMs can be automatically generated during the build stage of CI/CD pipelines. Integrating SBOM creation early in the development lifecycle helps establish traceability for every artifact and enables proactive security measures.

- **Syft**: A CLI tool by Anchore that can generate SBOMs in multiple formats (SPDX, CycloneDX) by analyzing container images, file systems, and directories [30].
- **Trivy**: In addition to vulnerability scanning, Trivy supports SBOM generation as part of its broader security functionality [31].

These tools can be integrated directly into CI/CD pipelines using GitHub Actions, GitLab CI, Jenkins, or other automation systems. Generated SBOMs are often stored as artifacts alongside container images or deployed applications.

3.2.3 Security and Compliance Benefits

The use of SBOMs in CI/CD and software lifecycle management introduces several key benefits:

- **Vulnerability tracking**: SBOMs enable organizations to trace vulnerabilities back to specific components, speeding up remediation efforts when new CVEs are disclosed.

- **Incident response and forensics**: In the event of a breach or software supply chain compromise, SBOMs provide forensic insight into the components involved.
- **License compliance**: Organizations can verify that open source licenses in use are compatible with their internal policies or legal requirements.
- **Continuous monitoring**: Tools and services can continuously scan stored SBOMs against updated vulnerability databases, detecting issues even after deployment.
- **Regulatory compliance**: Emerging standards such as the U.S. Executive Order 14028 and EU Cyber Resilience Act increasingly mandate SBOM adoption for certain software categories [26].

Release Stage

The release stage represents the final opportunity to validate security before artifacts are deployed to production environments. This stage focuses on ensuring that the artifacts being released meet security and compliance requirements.

- **Final image scanning**: A comprehensive scan of the final container image or artifact, including all layers and components. This scan should be performed on the exact artifact that will be deployed to production.
- **Security compliance verification**: Validation that the artifact meets all relevant security compliance requirements, such as HIPAA, PCI DSS, or internal security standards. Tools like Open Policy Agent (OPA) or custom compliance checks can be used to enforce these requirements.
- **Runtime configuration validation**: Verification of runtime configuration settings, such as environment variables, Kubernetes manifests, or cloud infrastructure configurations, to ensure they don't introduce security risks.
- **Policy compliance checks**: Enforcement of organizational security policies, such as requiring the presence of specific security controls or prohibiting certain configurations.

Release-stage scanning can be integrated into deployment pipelines or release management tools.

3.2.4 Best Practices

Based on industry experience and research, here are key best practices for implementing CI/CD security scanning, organized into three core themes:

Pipeline Design Principles

- **Shift left**: Integrate security scanning as early as possible in the development process. The shift-left approach refers to the practice of moving testing and quality checks earlier in the software development lifecycle, ideally starting from the design and implementation phases. This helps identify defects sooner, reduce remediation costs, and improve overall development efficiency. In the

context of security and vulnerability management, shifting left allows teams to catch misconfigurations or insecure dependencies before they reach production environments [32].

- **Fail fast**: Configure critical security checks to fail the pipeline immediately. This approach ensures that issues are detected and surfaced as early as possible in the CI/CD process, reducing wasted compute time and providing rapid feedback to developers [33].

- **Balance speed and depth**: Combine fast, lightweight scans (e.g., on each commit) with deeper, comprehensive scans (e.g., nightly or at release time). This strategy ensures continuous coverage without sacrificing performance or developer velocity.

Developer Enablement

- **Provide context**: Ensure that security findings include actionable remediation guidance, such as affected packages, impacted assets, CVSS scores, and suggested fixes. This empowers developers to respond quickly and accurately to issues.

- **Automate remediation**: Where possible, automate the application of security fixes. Automated remediation reduces mean time to resolution (MTTR), minimizes human error, and helps maintain compliance by promptly addressing known vulnerabilities [34].

- **Educate developers**: Provide training, guidelines, and tooling to help developers understand and proactively address security risks, shifting ownership of security earlier in the development process [35].

- **Security as code**: Manage security policies and configurations as code, storing them in version control systems. This enables review, traceability, and reproducibility of security changes across environments [36].

Governance and Monitoring

- **Continuous monitoring**: Implement ongoing security monitoring beyond the CI/CD pipeline to detect vulnerabilities and misconfigurations in running environments, ensuring rapid response to emerging threats [37].

- **Regular updates**: Keep security tools, scanners, and vulnerability databases (e.g., CVE feeds, SBOM analysis engines) up to date to ensure coverage against the latest known threats [38].

- **Measure effectiveness**: Track security metrics such as mean time to remediation (MTTR), number of recurring vulnerabilities, and policy violations to assess and continuously improve the security posture [39].

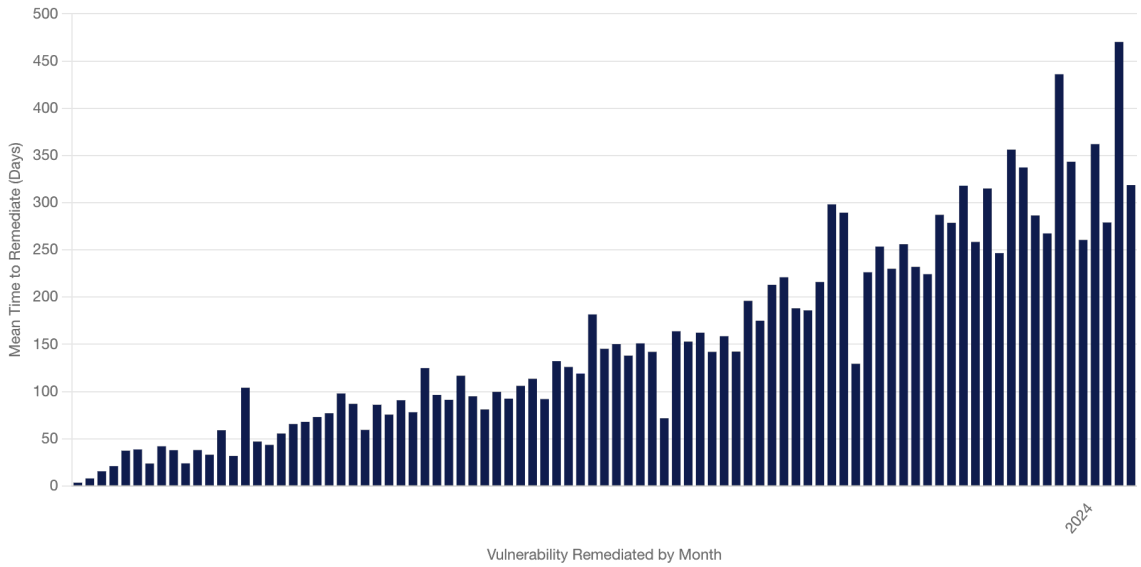


Figure 3.2: How long a project took to remediate known vulnerabilities in their dependencies. source: Sonatype

3.3 Automated Remediation in CI/CD Pipelines

Security scanning alone represents only half the equation in modern DevSecOps practices. The true transformation occurs when organizations move beyond mere vulnerability detection to implement systematic, automated remediation directly within their CI/CD workflows. This shift fundamentally changes how development teams approach security moving from fragmented, reactive patching to continuous, proactive vulnerability management integrated seamlessly into the software delivery process.

3.3.1 Strategic Decision Framework for Remediation

Implementing automated remediation requires sophisticated decision-making frameworks that extend far beyond simplistic ”fix everything” approaches. Organizations must establish nuanced policies that consider multiple factors simultaneously. Table 3.2 provides a structured framework for understanding how different factors contribute to automated remediation decisions.

Table 3.2: Automated Remediation Decision Framework

Factor	Data Sources	Role in Decision
Severity	CVSS scores, vendor advisories	Initial filtering mechanism - Critical/High severity vulnerabilities trigger immediate review
Exploitability	EPSS scores, VEX documents, public exploit databases	Distinguishes theoretical risks from active threats in the wild
Business Context	Asset classification, service tier, customer exposure	Determines automation aggressiveness - production systems get priority
Patch Availability	Package managers, vendor updates, fix validation status	Prevents automation attempts when no safe remediation exists
Change Risk	Dependency graph analysis, semantic versioning, test coverage	Assesses likelihood of regression or breaking changes
Environmental Factors	Deployment frequency, rollback capabilities, monitoring coverage	Influences automation confidence levels and approval workflows

CVSS severity ratings provide an initial filtering mechanism, with Critical and High-severity vulnerabilities typically warranting immediate automated intervention. However, relying solely on theoretical severity scores proves insufficient in practice.

Real-world exploitability data becomes crucial for making informed remediation decisions. The Exploit Prediction Scoring System (EPSS) [40] offers probabilistic assessments of whether vulnerabilities will be exploited in the wild, while Vulnerability Exploitability eXchange (VEX) documents provide vendor-specific guidance on actual risk levels. These resources help teams distinguish between vulnerabilities that pose immediate threats and those that remain largely theoretical.

Business context further refines remediation priorities. Customer-facing production systems demand more aggressive automated fixes compared to internal development environments or experimental services. Additionally, the availability and reliability of patches must factor into automation decisions triggering remediation only when validated fixes exist prevents the system from attempting impossible or risky corrections.

This framework enables organizations to move beyond simple rule-based approaches toward sophisticated, context-aware automation that balances security improvements with operational stability.

3.3.2 Implementing Automated Patch Workflows

Modern CI/CD platforms excel at generating automated pull requests that propose rather than impose security fixes. This approach preserves developer autonomy while accelerating remediation timelines. The automation can handle diverse scenarios:

- **Dependency Management**: Automatically upgrading vulnerable third-party libraries to patched versions while respecting semantic versioning constraints
- **Container Security**: Rebuilding images with updated base layers when security patches become available

- **Infrastructure Configuration**: Modifying Terraform templates, Kubernetes manifests, or cloud configuration files to eliminate insecure settings

- **Code-level Fixes**: Applying static analysis recommendations for common security anti-patterns

The most effective implementations enrich these automated pull requests with comprehensive context. Rather than presenting developers with cryptic dependency bumps, the system provides vulnerability descriptions, exploit likelihood assessments, links to security advisories, and clear explanations of why specific remediation approaches were selected. This transparency builds developer confidence and reduces the likelihood of automated PRs being blindly merged or reflexively rejected.

3.3.3 Validation and Continuous Feedback Mechanisms

Successful remediation extends beyond applying patches it requires rigorous validation to ensure fixes achieve their intended security goals without introducing functional regressions. Post-remediation workflows typically follow this sequence:

Security Validation: Re-scanning patched artifacts confirms vulnerability elimination and identifies any newly introduced security issues. This step catches scenarios where attempted fixes prove ineffective or where dependency updates inadvertently introduce different vulnerabilities.

Functional Testing: Comprehensive test suites ranging from unit tests to full integration scenarios verify that security changes don't break existing functionality. This becomes particularly critical for dependency upgrades, which can introduce subtle behavioral changes despite maintaining API compatibility.

Documentation Updates: Automatically regenerating Software Bills of Materials (SBOMs) ensures downstream vulnerability tracking remains accurate, while updating security documentation maintains compliance with organizational policies and regulatory requirements.

3.3.4 Addressing Implementation Challenges

Real-world automated remediation faces several persistent challenges that can undermine its effectiveness. Breaking changes represent the most common concern security patches, particularly major dependency upgrades, can alter application behavior in unexpected ways. Development teams naturally become wary of automation that has previously introduced production issues.

The absence of available patches compounds this problem. Zero-day vulnerabilities and recently disclosed security flaws often lack immediate remediation options, leaving automation systems with no safe actions to perform. Organizations must design their systems to gracefully handle these scenarios rather than attempting potentially harmful workarounds.

False positive rates in vulnerability scanners create additional friction. When automated systems generate remediation attempts for non-existent or irrelevant vulnerabilities, they erode developer trust and waste valuable resources. Similarly, the sheer volume of automated pull requests can overwhelm development teams, particularly in microservices architectures where a single vulnerability might trigger dozens of simultaneous fix attempts across multiple repositories.

3.3.5 Key Implementation Considerations

Organizations planning automated remediation implementations should prioritize these critical factors:

- **Start conservatively** with low-risk, high-confidence scenarios before expanding automation scope
- **Establish clear rollback procedures** for when automated fixes introduce problems
- **Implement approval workflows** that allow human oversight without completely blocking automation
- **Monitor automation effectiveness** through metrics like mean-time-to-remediation and false positive rates
- **Communicate transparently** with development teams about automation capabilities and limitations
- **Design for scalability** from the outset to handle enterprise-scale vulnerability volumes

The evolution toward automated remediation represents a maturation of DevSecOps practices, transforming security from a development bottleneck into an enabler of faster, safer software delivery. Success requires careful balance between automation benefits and human oversight, ensuring that security improvements enhance rather than hinder the development process.

The practical and organizational challenges of automated remediation underscore the need for more adaptive, context-aware solutions that can navigate the complex decision-making required in modern development environments. In the following chapter, we present a comprehensive framework for automated vulnerability remediation that operationalizes these principles through a closed-loop system detection triggers action, action is validated, and results feed back into detection and prioritization delivering scalable, intelligent remediation workflows that align with developer practices and organizational risk tolerance.

Chapter 4

Automated Vulnerability Remediation

4.1 Introduction to Automated Vulnerability Remediation

In the rapidly evolving landscape of modern cybersecurity, vulnerability remediation stands as one of the most critical and challenging aspects. The sheer volume of new vulnerabilities discovered annually 4.1 often numbering in the tens of thousands presents an insurmountable task for traditional, manual approaches to vulnerability management. As a result, security teams frequently struggle to keep pace, leading to an ever-growing backlog of unaddressed risks and an increased attack surface. Automated vulnerability remediation has emerged not merely as a beneficial practice, but as a necessary evolution, empowering organizations to address security gaps at scale, with greater efficiency, and ultimately, to significantly improve their overall security posture, ensure compliance, and boost developer productivity by minimizing security-related toil [41].

Automated vulnerability remediation refers to the systematic process of leveraging technology to detect, prioritize, and implement fixes for security vulnerabilities with minimal human intervention. This transformative approach shifts traditional, reactive, and labor-intensive remediation workflows into proactive, streamlined operations that can effectively keep pace with the dynamic and rapid evolution of security threats.

Automated remediation is best understood as a closed-loop system where detection triggers action, action is validated, and results feed back into detection and prioritization mechanisms. This continuous feedback cycle ensures that the system

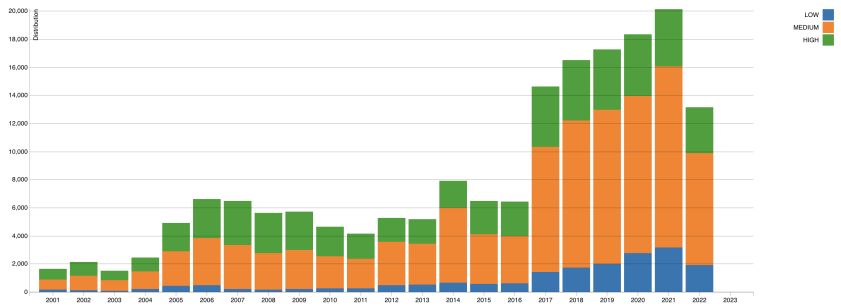


Figure 4.1: CVSS Severity Distribution Over Time. Source: [42]

learns from both successes and failures, progressively improving its effectiveness while adapting to organizational needs and threat landscapes. Each component of this loop analysis, execution, and validation operates as an integrated step that contributes to an overall system perspective rather than isolated security activities.

This chapter delves into the comprehensive lifecycle of automated remediation, exploring each stage of this closed-loop system: vulnerability analysis (the detection step), diverse execution strategies tailored to modern environments (the action step), and robust validation techniques that feed learning back into the system (the feedback step). The goal is to establish a structured and nuanced understanding of how remediation automation can be effectively designed and implemented within complex cloud-native and CI/CD-based environments, laying the groundwork for more advanced solutions like Vulnbot.

4.2 Vulnerability Analysis: The Detection Step

The automated remediation lifecycle critically begins with the accurate identification and prioritization of vulnerabilities the detection step of our closed-loop system. This initial stage determines which security issues warrant automated intervention and establishes the foundation for all subsequent remediation decisions. In contemporary cloud-native environments, this typically involves continuous and ubiquitous scanning across a wide array of assets, including container images, infrastructure-as-code (IaC) configurations, and running runtime environments. Modern vulnerability scanners, such as Trivy, Grype, and OSV-Scanner, are indispensable for generating comprehensive reports that detail Common Vulnerabilities and Exposures (CVEs), associated severity scores, and affected software components.

4.3 Automated Remediation Execution: The Action Step

Once vulnerabilities have been rigorously analyzed and prioritized in the detection step, the automated remediation engine executes the action step of our closed-loop system generating and applying the necessary fixes. This stage transforms vulnerability intelligence into concrete remediation activities, bridging the gap between detection and resolution. The execution strategy employed can vary significantly based on the environment, the nature of the vulnerability, and the specific software supply chain components involved:

- **Package Updates:** For software dependencies, this typically involves generating automated pull requests (PRs) to source code repositories or directly injecting updates into CI/CD pipelines. Tools like Dependabot and Renovate continuously monitor project dependencies and automatically create PRs when new versions with security fixes become available, streamlining the update process for developers.

- **Container Rebuilds:** In environments leveraging immutable infrastructure, vulnerabilities found in base images or application dependencies within containers trigger automated pipelines to rebuild the container image with the patched components. This ensures a predictable and clean update, rather than in-place modifications to running containers.

- **Infrastructure-as-Code (IaC) Updates:** Remediation for IaC vulnerabilities involves modifying configuration files and manifests (e.g., Terraform, CloudFormation, Kubernetes manifests). Examples include bumping module versions, applying security patches to resource definitions, or correcting misconfigurations that expose vulnerabilities.

An orchestrating layer, often comprised of custom bots or specialized platforms, manages this patch lifecycle. Advanced implementations are increasingly leveraging AI-powered remediation suggestion systems. These systems can propose minimal-diff patches that aim to preserve compatibility and reduce the risk of regressions by analyzing code context, dependency graphs, and historical patch success rates.

To minimize disruption and risk, these automated fixes are typically first deployed to staging or pre-production environments for thorough verification before being promoted to production. Risk mitigation strategies such as canary deployments or blue-green rollouts are commonly employed to ensure stability and allow for rapid rollback if unintended side effects are detected.

4.4 Validation: The Learning Step

Post-remediation validation ensures that applied fixes effectively eliminate vulnerabilities without introducing new issues or regressions. This step verifies the success of individual remediation actions and confirms the stability of the system. Automated validation strategies are integrated into the CI/CD pipeline:

- **Regression Testing:** Automated CI test suites are triggered by remediation PRs or deployments. These tests verify that the software's functionality remains intact and that the patch has not introduced any unintended behavior or performance degradation.
- **Rescanning Artifacts:** After a patch is applied and new artifacts (e.g., container images, updated codebases) are built, they are rescanned to confirm that the specific vulnerability is no longer present. This provides direct verification of the fix's effectiveness.
- **Runtime Behavioral Monitoring:** Continuous monitoring of deployed applications for anomalies can detect unexpected behaviors introduced by recent changes, such as increased error rates, unusual resource consumption, or unauthorized network calls, serving as an early warning system for regressions.

By focusing on these validation activities, the system ensures that remediation efforts are effective and that software remains stable and secure.

4.5 Architectural Considerations in Cloud-Native Environments

In highly dynamic cloud-native environments, automated remediation systems must be deeply and seamlessly integrated with existing development, deployment, and operational tooling. Key architectural elements that underpin effective remediation automation include:

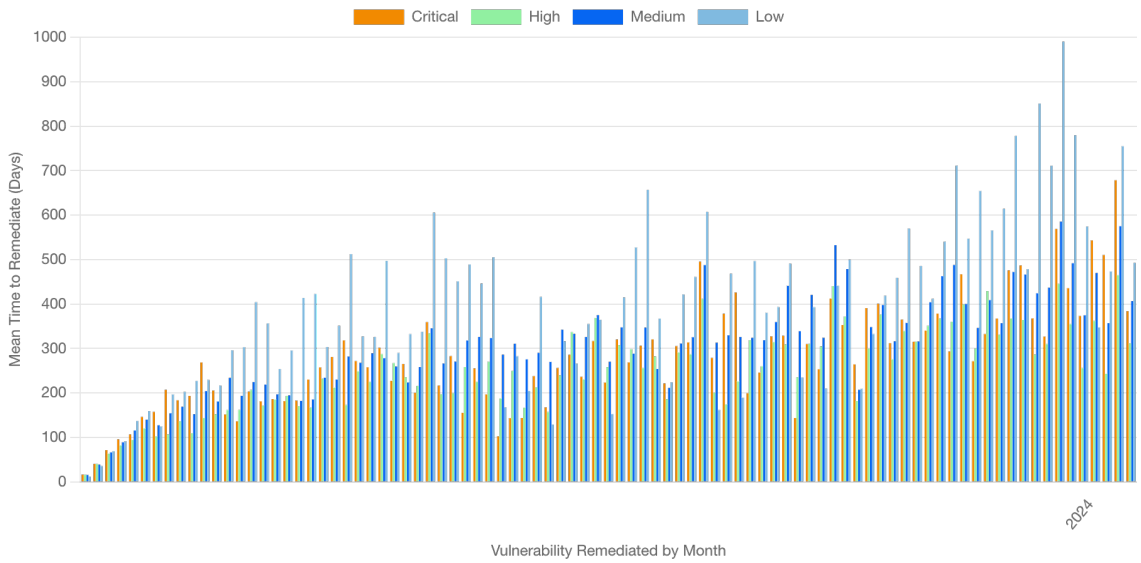


Figure 4.2: How long projects took on average to remediate dependency vulnerabilities broken down by severity. source: Sonatype

- **Event-Driven Pipelines:** Modern CI/CD platforms (e.g., GitHub Actions, GitLab CI, Tekton) serve as the backbone, enabling remediation workflows to be automatically triggered by various events. These events can include the discovery of a new vulnerability by a scanner, the merging of a remediation PR, or the addition of a new dependency.
- **Immutable Infrastructure:** This paradigm is fundamental to predictable patching. Instead of patching in-place, new, corrected artifacts (e.g., container images, IaC deployments) are built and deployed, replacing the old, vulnerable ones. This ensures consistency and simplifies rollbacks.
- **Policy Engines:** Tools like Open Policy Agent (OPA) enable the enforcement of security policies as code. These engines can enforce critical remediation Service Level Agreements (SLAs), block deployments if they exceed defined vulnerability thresholds (e.g., "no critical CVEs allowed in production"), or dictate permissible patch sources.
- **Artifact Registries and Software Bill of Materials (SBOMs):** Secure artifact registries store trusted build artifacts. SBOMs, which provide a comprehensive, machine-readable inventory of all components, dependencies, and their versions within a software artifact, are crucial. They allow for precise tracking of vulnerability propagation across builds and deployed systems, facilitating rapid identification of affected assets when a new vulnerability is disclosed.

Cloud-native environments, by their very nature, benefit significantly from declarative configuration, which makes it inherently easier to manage, track, and validate changes across increasingly complex and distributed systems.

4.6 Challenges and Future Directions

Despite significant advancements, several complex challenges persist in achieving truly reliable, safe, and scalable automated vulnerability remediation:

- **Patch Safety:** The blind application of updates carries inherent risks. New versions or patches can introduce breaking API changes, unintended side effects, or unhandled edge cases, particularly within complex and deeply nested dependency trees. Ensuring a patch doesn't break existing functionality remains a critical hurdle.

- **Interoperability:** Integrating disparate tools across the entire security pipeline from vulnerability scanning and analysis to automated patching, CI/CD, and runtime environments remains a significant challenge due to varying data formats, APIs, and operational models.

- **Explainability:** For security teams and developers, understanding why a particular patch is recommended, how it addresses a vulnerability, and what its potential side effects might be is crucial for trust and adoption. The "black box" nature of some automated systems can hinder this transparency.

Future work in automated vulnerability remediation is poised to address these challenges with innovative solutions. This includes integrating Large Language Models (LLMs) to improve the quality of auto-generated patches by providing context-aware code modifications and generating clear, descriptive commit messages. Expanding reinforcement learning techniques will enable even more adaptive prioritization strategies, learning from the success and failure of past remediations. Furthermore, leveraging continuous runtime feedback to dynamically refine static remediation decisions will enhance both the precision and safety of automated vulnerability management systems.

Chapter 5

Vulnbot

Modern software systems are increasingly reliant on a vast and complex ecosystem of open-source dependencies. While these dependencies accelerate development, they also introduce a significant attack surface, as they are frequently prone to security vulnerabilities. Although Continuous Integration (CI) pipelines commonly incorporate static analysis tools and vulnerability scanners to detect these issues, the subsequent remediation of findings remains a largely manual, labor-intensive, and time-consuming task for development teams. This manual bottleneck directly impacts an organization's Mean-Time-To-Remediation (MTTR) and overall security posture.

This chapter introduces **Vulnbot**, a modular Command Line Interface (CLI) tool designed as a CI-integrated bot that automates the detection, patching, and remediation of known vulnerabilities within open-source dependencies. Vulnbot is engineered to support multiple package ecosystems, with an initial focus on npm for JavaScript/Node.js projects and Go modules for Go applications. Its seamless integration into modern CI/CD pipelines, exemplified by its compatibility with platforms like GitHub Actions, positions Vulnbot as a critical tool for embedding automated security remediation directly into the developer workflow.

5.1 Motivation

The proliferation of security vulnerability databases such as the GitHub Advisory Database (GHSA), OSV (Open Source Vulnerabilities), and traditional CVE repositories has significantly enhanced the ability to detect vulnerabilities through specialized tools like Trivy, OSV-Scanner, and Grype. However, a critical disconnect persists within the typical software development lifecycle:

- **Resolution suggestions**, detailing available fixed versions or remediation steps, are frequently provided without immediate context or actionable mechanisms.
- A clear pathway for **actionable, automated remediation** directly integrated into developer workflows is often absent.

Vulnbot was specifically designed to bridge this crucial gap. By providing a streamlined, end-to-end automation solution, Vulnbot directly assists developers and security teams in dramatically reducing their Mean-Time-To-Remediation (MTTR), thereby enhancing overall software supply chain security. Its proactive integration of security fixes directly into the development lifecycle fosters a "shift-left" security culture, ensuring vulnerabilities are addressed as early and efficiently as possible.

5.2 Use Stories and System Overview

This section presents detailed use stories to illustrate how Vulnbot integrates vulnerability detection and automated remediation across different phases of software development and operations. Following these narratives, we describe the primary actors, key components, and workflows involved, culminating in a high-level Proof of Concept (PoC) architecture.

5.2.1 Actors & Components

The following table outlines the key stakeholders and technical elements that interact within the Vulnbot ecosystem:

Category	Element	Description
Actor	Developer	Utilizes Vulnbot either locally or through CI integration to identify and remediate vulnerable dependencies during feature development and maintenance. Receives automated pull requests or direct patches with suggested fixes, significantly reducing manual security toil.
	DevOps Engineer	Integrates and configures Vulnbot within the CI/CD pipeline (e.g., via GitHub Actions, GitLab CI) to establish automated security gates and trigger proactive patching in build workflows.
Component	Vulnbot CLI/Bot (CI Integration)	The core application that can be executed as a command-line tool or configured as an automated bot within CI environments. It orchestrates the scanning, patching, and pull request generation process, capable of being configured for auto-merging minor/patch-level updates based on defined policies for example a cvss threshold.
	Vulnerability Scanner	External, industry-standard tools (e.g., Trivy, OSV-Scanner, Grype) that are integrated by Vulnbot. These scanners perform the actual detection of known vulnerabilities across various artifacts such as source code, manifest files, and container images, providing detailed findings.
	Remediation Engine	The intelligent core module of Vulnbot responsible for parsing scan results, deciding the most appropriate patch, programmatically applying the fix by updating relevant dependency manifests (e.g., package.json, go.mod), and performing preliminary integrity checks post-fix.

Category	Element	Description
	CI/CD Pipeline	The automated framework (e.g., GitHub Actions, GitLab CI) that executes Vulnbot as an integral part of the build, test, and deployment process. It enforces security gates before code merges or deployments and facilitates comprehensive audit logging of all security-related actions.
	Logging & Audit System	A centralized system that collects and stores all events generated by Vulnbot runs, including detected vulnerabilities, applied patches, opened pull requests, and any errors. This provides essential data for compliance requirements, operational observability, and forensic analysis for rollback capabilities.

5.2.2 Case Studies

To demonstrate Vulnbot's versatility and impact, we present three distinct case studies illustrating its application in diverse organizational and technological contexts.

Case Study 1: Enterprise Network Assessment

Background: A large enterprise, characterized by a complex hybrid infrastructure encompassing cloud services, legacy systems, and a myriad of third-party applications, faced significant challenges in its vulnerability management. Their existing approach was highly fragmented, heavily reliant on manual triage, and cumbersome ticketing systems, resulting in slow response times and overlooked risks.

Implementation: The organization strategically integrated Vulnbot into its core CI/CD pipelines and infrastructure management workflows. The system was configured to leverage a combination of established vulnerability scanners (specifically OSV-Scanner and Trivy). This setup enabled vulnerability detection with prioritization, leveraging CVSS scores to assess severity and guide remediation efforts. Crucially, Vulnbot's automated pull request generation feature was employed to streamline the patching of outdated dependencies across a vast portfolio of internal microservices and applications.

Results: Vulnbot successfully identified and proactively remediated several high-impact vulnerabilities that had previously been overlooked due to limited visibility into deeply nested transitive dependencies and dynamic runtime components. The system's context-aware prioritization algorithms allowed security teams to effectively focus their efforts on exploitable and reachable vulnerabilities first, rather than being bogged down by low-impact findings. Direct integration with GitHub facilitated seamless PR generation, complete with version updates and relevant changelogs, significantly simplifying the developer review process.

Impact: The enterprise reported a substantial reduction in manual effort associated with vulnerability management and a marked decrease in false positives. This led to a demonstrable improvement in Mean-Time-To-Remediation (MTTR) across their entire software estate. Developers gained increased confidence in approving and merging automated security updates, knowing they were thoroughly vetted. The automation of patch suggestions also liberated valuable security and development resources, allowing teams to dedicate more time to strategic security initiatives and innovation.

Case Study 2: Web Application Security

Background: A technology company specializing in a Software-as-a-Service (SaaS) web application sought to dramatically enhance its security posture by implementing a "shift-left" strategy for vulnerability remediation, moving fixes earlier into the development lifecycle.

Implementation: Vulnbot was seamlessly integrated directly into the development pipeline using GitHub Actions. This integration enabled continuous monitoring of critical dependency files (e.g., package.json for Node.js, requirements.txt for Python). Upon the discovery of vulnerable packages, Vulnbot automatically generated targeted pull requests. Furthermore, the tool was configured to enforce specific remediation policies based on CVSS scores and the confirmed availability of secure fix versions.

Results: The system proved highly effective in identifying critical vulnerabilities, including outdated packages with known SQL injection and cross-site scripting (XSS) issues that posed direct threats to the web application. Vulnbot generated precise security pull requests that included verified safer versions and direct references to relevant security advisories, providing developers with immediate context. The ability to surface and address these vulnerabilities during the active development phase was pivotal in ensuring they were resolved long before reaching production environments.

Impact: The company's overall security posture improved significantly, with a marked decrease in post-deployment security incidents. Vulnerabilities were now resolved proactively during development, rather than reactively after deployment. This proactive approach maintained, and in some cases even improved, development velocity by preventing late-stage security rework. Moreover, the integrated remediation process fostered a stronger, more security-aware engineering culture throughout the organization.

Case Study 3: IoT Device Vulnerability Management

Background: An organization managing a vast and geographically distributed fleet of Internet of Things (IoT) devices faced the complex challenge of continuously monitoring and remediating firmware and configuration-level vulnerabilities across its diverse ecosystem, which included a mix of proprietary and open-source components.

Implementation: Vulnbot's capabilities were extended to support the comprehensive scanning of container images and firmware packages specifically utilized within the IoT device ecosystem. It was deeply integrated with the organization's existing Git repositories, where device configurations and firmware definitions were stored, and with their configuration management tools. This integration allowed vulnerabilities within base images and embedded software packages to be automatically detected and subsequently patched through the generation of security pull requests.

Results: The system successfully identified numerous vulnerabilities inherent to IoT deployments, including insecure default configurations, outdated network protocols, and legacy software packages with known exploits. Vulnbot automated updates to critical files like Dockerfiles and base images, ensuring that new device deployments or firmware updates incorporated the latest security fixes. It also enforced security policies directly within the CI pipelines for IoT device builds.

Impact: The organization achieved a dramatic improvement in its IoT device security management capabilities. By maintaining rigorously up-to-date and secure images and configurations, they significantly reduced their overall attack surface across the entire device fleet and enhanced compliance with stringent internal security standards. Vulnbot's automation ensured security consistency and reduced operational overhead across a highly diverse set of device deployments and geographic locations.

5.3 Overview and Architecture

Vulnbot is architected as a modular and extensible system, designed to seamlessly integrate with existing development workflows and security tooling. It comprises several distinct, yet interconnected, components:

- **Core CLI Engine:** This serves as the primary interface for Vulnbot, wrapping around and orchestrating various third-party vulnerability scanners (e.g., Trivy, OSV-Scanner). It manages the execution of scans and the initial processing of raw scan outputs.
- **Language-Specific Parser and Fixer:** This critical module is responsible for understanding the structure of language-specific manifest files (e.g., package.json and package-lock.json for Node.js, go.mod and go.sum for Go). It parses the scan results to identify vulnerable packages with known fixed versions and then programmatically applies the necessary upgrades to these manifest files.
- **Git Automation Layer:** This layer handles all Git-related operations, including creating new branches, committing the patched changes, and pushing them to the remote repository. This ensures that remediation efforts are version-controlled and can be easily reviewed or rolled back.
- **CI/CD Integration Layer:** Designed for seamless compatibility, this layer facilitates Vulnbot's operation within popular automation platforms like GitHub Actions, GitLab CI, or Jenkins. It enables automated execution of Vulnbot as part of the CI/CD pipeline, triggering scans and remediation actions based on defined events.

Figure 5.1 illustrates the high-level architecture of the Vulnbot system, demonstrating the interaction between its core components and external integrations.

5.4 Workflow

Vulnbot executes a well-defined, high-level workflow to automate the detection and remediation process, ensuring a systematic approach from vulnerability identification to patch application:

1. **Scanning:** Vulnbot initiates the process by invoking an integrated vulnerability scanner (e.g., Trivy, OSV-Scanner) on the target codebase or artifact. The scanner's output, containing detailed vulnerability findings, is returned as a machine-readable JSON format.
2. **Parsing:** The JSON scan results are then analyzed by Vulnbot's internal parser. This step extracts crucial information, specifically identifying vulnerable packages that have a confirmed fixed version available.
3. **Matching:** The identified vulnerable packages are meticulously mapped to their corresponding manifest files within the repository (e.g., associating a vulnerable Node.js package with package.json). This ensures that the correct dependency file is targeted for modification.

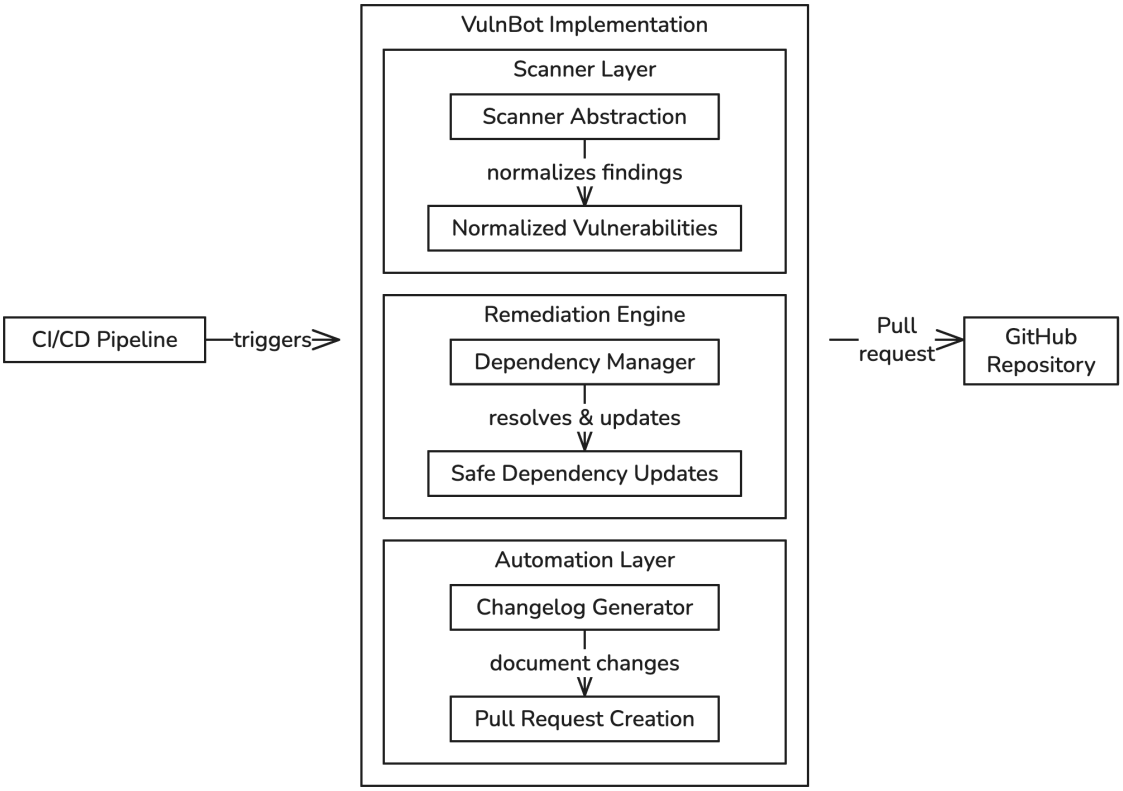


Figure 5.1: High-level architecture of Vulnbot.

4. **Patching:** The remediation engine programmatically upgrades the affected dependencies to their identified fixed versions. This involves modifying the manifest files (e.g., updating version numbers in package.json or go.mod) and potentially regenerating associated lock files (package-lock.json, go.sum) to reflect the new dependency tree.

5. **Committing:** Upon successful patching, Vulnbot creates a new Git branch, stages and commits the changes, and pushes this branch to the remote repository. The commit message typically includes details about the patched vulnerabilities.

6. **Pull Request (optional):** When integrated into a CI environment, Vulnbot can automatically open a pull request (PR) in the remote repository. This PR provides a clear description of the applied fixes, references to security advisories, and the updated files, enabling developers to review and merge the changes with ease.

5.5 Key Use Cases

Vulnbot supports several distinct use cases, catering to different stakeholders within the software development and security lifecycle:

Developer Security Automation

Developers can run Vulnbot locally as a CLI tool within their development environment. This allows them to proactively identify and upgrade vulnerable dependencies in their codebase based on the latest vulnerability intelligence before committing

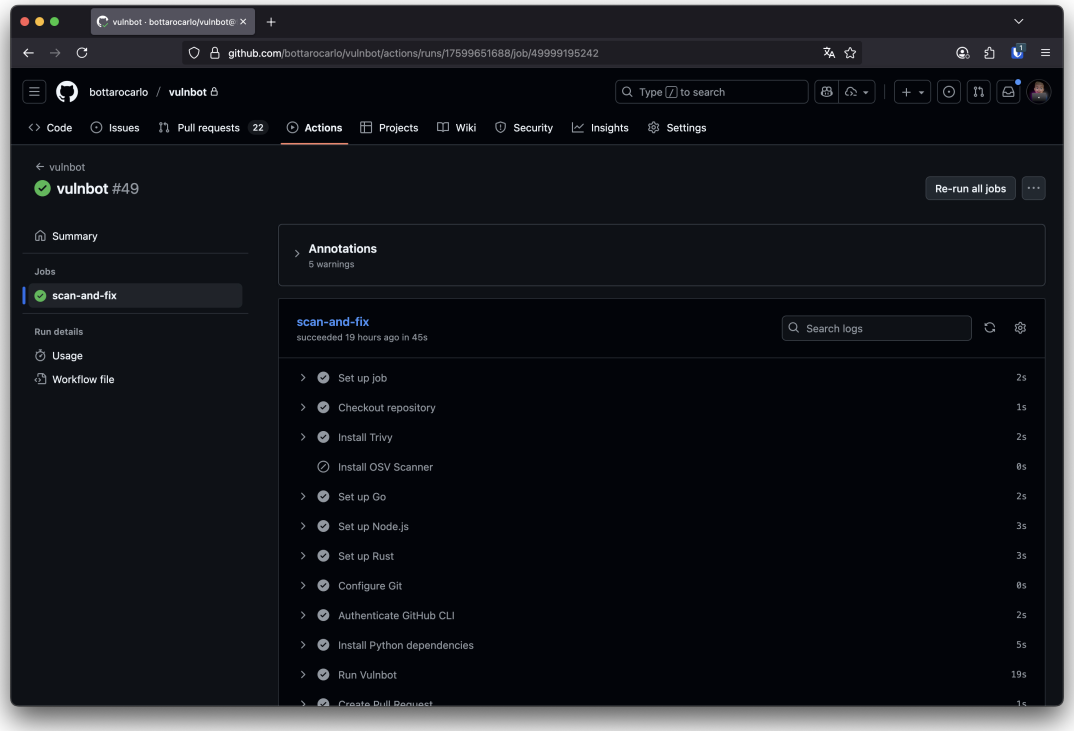


Figure 5.2: CI workflow with Vulnbot integration.

their changes. This ”shift-left” approach ensures that security issues are addressed early, reducing rework and integration conflicts later in the pipeline.

CI/CD Integration

The most powerful application of Vulnbot is its integration into CI/CD pipelines. It can be configured to run automatically after each code push (e.g., on git push) or on a predefined schedule (e.g., nightly builds). In this setup, Vulnbot automatically detects and remediates vulnerabilities, generating pull requests that developers can review and merge, ensuring continuous security posture maintenance.

Repository Hardening

For security and platform engineering teams, Vulnbot provides a robust mechanism for continuous repository hardening. By integrating the tool into pre-deployment checks or as a scheduled job across a fleet of repositories, teams can enforce consistent security standards and proactively address known vulnerabilities across all internal and external codebases.

5.6 CI Integration via GitHub Actions

Vulnbot’s integration with GitHub Actions represents a critical aspect of its practical deployment. The CI integration enables automated vulnerability scanning and remediation as part of the standard development workflow, ensuring that security becomes an integral part of the software delivery pipeline.

5.6.1 GitHub Actions Workflow Configuration

The following workflow configuration demonstrates how Vulnbot can be integrated into a GitHub Actions pipeline:

```yaml
name: Vulnbot Security Scan

on:
  schedule:
    - cron: '0 2 * * *'  # Daily at 2 AM
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  security-scan:
    runs-on: ubuntu-latest
    permissions:
      contents: write
      pull-requests: write
      security-events: write

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v4
      with:
        token: ${{ secrets.GITHUB_TOKEN }}

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'
        cache: 'npm'

    - name: Setup Go
      uses: actions/setup-go@v4
      with:
        go-version: '1.21'

    - name: Install Vulnbot
      run: |
        curl -sSL https://github.com/vulnbot/releases/latest/download/vulnbot-linux-amd64 -o vulnbot
        chmod +x vulnbot
        sudo mv vulnbot /usr/local/bin/

    - name: Run Vulnerability Scan
      run: |
        vulnbot scan --scanner=osv --format=json --output=vulns.json

    - name: Apply Automated Patches
      if: success()
```

```
48        env:
49          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
50        run: |
51          vulnbot patch --input=vulns.json --auto-pr --cvss-
      threshold=7.0
52
53      - name: Upload Scan Results
54        uses: actions/upload-artifact@v4
55        if: always()
56        with:
57          name: vulnerability-scan-results
58          path: vulns.json
```

Listing 5.1: GitHub Actions Workflow for Vulnbot Integration

5.6.2 Configuration Options and Customization

Vulnbot supports extensive configuration through both command-line flags and configuration files. Organizations can customize its behavior to match their specific security policies and development workflows:

- **Scanner Selection**: Choose between OSV-Scanner, Trivy, or Grype based on organizational preferences and requirements
- **Severity Thresholds**: Configure CVSS score thresholds to determine which vulnerabilities trigger automated remediation
- **Auto-merge Policies**: Define rules for automatically merging low-risk dependency updates
- **Notification Channels**: Integrate with Slack, Microsoft Teams, or email for vulnerability alerts
- **Exclusion Rules**: Specify packages or vulnerability types to exclude from automated patching

5.6.3 Security and Permissions

The GitHub Actions integration requires careful consideration of security permissions and access controls:

- **Repository Permissions**: Vulnbot requires write access to create branches and pull requests
- **Security Events**: Optional integration with GitHub Security tab for vulnerability tracking
- **Secret Management**: Secure handling of API tokens and credentials through GitHub Secrets
- **Branch Protection**: Integration with branch protection rules to ensure security gates

5.7 Language Support

Currently, Vulnbot's remediation capabilities are designed for prominent open-source ecosystems:

- **JavaScript / Node.js:** It accurately parses package.json to identify dependencies and regenerates package-lock.json (or yarn.lock) to reflect updated package versions, ensuring dependency consistency.
- **Go modules:** Vulnbot effectively parses go.mod and go.sum files to identify vulnerable Go modules and can suggest go get-style upgrades for direct application of fixes.
- **Rust:** Vulnbot can parse Cargo.toml and Cargo.lock files to identify vulnerable Rust crates and suggest cargo update commands to apply fixes.
- **Python:** Vulnbot can parse requirements.txt and Pipfile.lock files to identify vulnerable Python packages and suggest pip install commands to apply fixes.

The internal design of Vulnbot emphasizes modularity and extensibility. This architectural choice is deliberate, allowing for straightforward expansion to support additional language ecosystems (e.g., Python's requirements.txt and Pipfile.lock, Java's pom.xml for Maven or build.gradle for Gradle, Rust's Cargo.toml, or Ruby's Gemfile) in future versions.

5.8 Technical Implementation Details

This section provides detailed implementation specifics of Vulnbot's core components, demonstrating the practical aspects of building a production-ready automated vulnerability remediation system.

5.8.1 Scanner Integration and Abstraction Layer

Vulnbot implements a pluggable scanner architecture that allows integration with multiple vulnerability scanning tools through a common interface:

```
1  class Scanner:
2      def scan(self, target_dir: str) -> List[Vulnerability]:
3          """Execute vulnerability scan and return structured
       results"""
4          pass
5
6      def get_scanner_version(self) -> str:
7          """Return scanner version for audit trail"""
8          pass
9
10 class OSVScanner(Scanner):
11     def scan(self, target_dir: str) -> List[Vulnerability]:
12         cmd = ["osv-scanner", "--format=json", target_dir]
13         result = subprocess.run(cmd, capture_output=True,
       text=True)
14         return self._parse_osv_output(result.stdout)
15
```

```
    def _parse_osv_output(self, json_output: str) -> List[Vulnerability]:
        data = json.loads(json_output)
        vulnerabilities = []
        for result in data.get("results", []):
            for package in result.get("packages", []):
                for vuln in package.get("vulnerabilities", []):
                    vulnerabilities.append(Vulnerability(
                        id=vuln.get("id"),
                        summary=vuln.get("summary"),
                        severity=self._extract_cvss(vuln),
                        affected_package=package.get("package", {}).get("name"),
                        fixed_version=self._extract_fixed_version(vuln)
                    ))
        return vulnerabilities
```

Listing 5.2: Scanner Interface Abstraction

5.8.2 Language-Specific Patcher Implementation

Each supported language ecosystem requires specific handling for dependency management and manifest file updates:

```
class NPMPatcher:
    def apply_patch(self, package_name: str, target_version: str):
        """Apply security patch for NPM package"""
        try:
            # Check if package.json exists
            if not os.path.exists("package.json"):
                raise PatchError("No package.json found")

            # Install specific version
            cmd = ["npm", "install", f"{package_name}@{target_version}"]
            result = subprocess.run(cmd, capture_output=True, text=True)

            if result.returncode != 0:
                raise PatchError(f"Failed to install {package_name}: {result.stderr}")

            # Verify installation
            self._verify_installation(package_name, target_version)

            return PatchResult(
                success=True,
                package=package_name,
                version=target_version,
                changes=self._get_lock_file_changes()
```

```
            )
        except Exception as e:
            return PatchResult(success=False, error=str(e))

    def _verify_installation(self, package: str, version: str):
        """Verify that the correct version was installed"""
        with open("package-lock.json", "r") as f:
            lock_data = json.load(f)

        installed_version = lock_data.get("packages", {}).get(f"node_modules/{package}", {}).get("version")
        if installed_version != version:
            raise PatchError(f"Version mismatch: expected {version}, got {installed_version}")
```

Listing 5.3: NPM Patcher Implementation

5.8.3 Changelog Generation

To enhance the context provided in pull requests, Vulnbot automatically generates changelogs for updated dependencies:

```
    def generate_changelog(files, count, vulns=None):
    """
    Generate a security changelog for vulnerability fixes.

    Args:
        files: List of modified file paths
        count: Total number of vulnerabilities fixed
        vulns: Dict with vulnerability details (optional)
    """

    base = f"""## Security Updates

This PR addresses {count} vulnerabilities by updating dependencies in {len(files)} files.

### Modified Files
{chr(10).join(['- `' + f + '`' for f in files])}
"""
    if not vulns:
        return base

    # Counters
    sev_count = {"CRITICAL": 0, "HIGH": 0, "MEDIUM": 0, "LOW": 0}
    grouped = {}

    for v in vulns.values():
        s = v.get("severity", "UNKNOWN").upper()
        sev_count[s] = sev_count.get(s, 0) + 1
        grouped.setdefault(v.get("ecosystem", "other"), []).append(v)
```

```
29
30      out = f"""##  Security Fixes
31
32 This PR resolves **{count} vulnerabilities** in **{len(files)
       } files**.
33
34 ### Modified Files
35 {chr(10).join(['- `' + f + '`' for f in files])}
36
37 ### Severity Breakdown
38 """ + "".join([f"- {k}: {v}\n" for k, v in sev_count.items()
       if v]) + "\n"
39
40      # By ecosystem
41      order = {"CRITICAL": 0, "HIGH": 1, "MEDIUM": 2, "LOW": 3,
        "UNKNOWN": 4}
42      for eco, vs in grouped.items():
43          out += f"#### {eco.upper()} Dependencies\n"
44          for v in sorted(vs, key=lambda x: order.get(x.get("
       severity", "UNKNOWN"), 5)):
45              out += (f"- **{v.get('package_name','?')}** ({v.
       get('severity')}, "
46                      f"CVSS {v.get('cvss_score',0)}) -> `{v.
       get('fixed_version','?')}` "
47                      f"[CVE: {v.get('cve_id','N/A')}]\n")
48
49      return out + "\n Generated by VulnBot\n"
```

Listing 5.4: Changelog Generation

5.9 Evaluation and Results

5.9.1 Real-World Application: Bitwarden CLI Repository

To further validate Vulnbot's practical effectiveness, the system was tested on a real-world open-source project the **Bitwarden CLI** repository.[1] Bitwarden is a widely adopted open-source password manager trusted by both enterprises and individual users. Its command-line interface (CLI) component relies on a variety of npm dependencies, making it an ideal candidate for automated dependency vulnerability management testing.

Objective

The objective of this test was to assess Vulnbot's ability to:

- Detect vulnerabilities in a mature, production-grade open-source project.
- Automatically identify available patches or updated dependency versions.
- Generate actionable and review-ready pull requests (PRs) that align with real-world development workflows.

[1] https://github.com/bitwarden/cli

Setup and Configuration

The Bitwarden CLI repository was evaluated using Vulnbot's **GitHub Actions (GHA)** integration, rather than through local execution. This setup allowed the tool to be tested in a fully automated, CI/CD-native environment, closely replicating a real-world DevSecOps workflow.

The following configuration was applied:

- **Scanner:** Trivy Scanner (latest stable version)
- **CVSS Threshold:** 7.0 (only HIGH and CRITICAL vulnerabilities were remediated automatically)
- **Mode:** GitHub Actions workflow with automatic pull request generation enabled

The following workflow snippet shows the key steps executed by the GHA pipeline:

```
- name: Run VulnBot scan
  run: |
    docker run --rm \
      -v "${{ github.workspace }}:/workspace" \
      -e GITHUB_TOKEN="${{ secrets.GITHUB_TOKEN }}" \
      -e VULN_SEVERITY_THRESHOLD="${{ github.event.inputs.
   severity_threshold || '4.0' }}" \
      -e SCANNER="${{ github.event.inputs.scanner || 'trivy'
   }}" \
```

Listing 5.5: Vulnbot Execution via GitHub Actions on Bitwarden CLI Repository

This configuration enabled end-to-end automation: the workflow scanned dependencies, selected suitable patched versions, regenerated the lockfile, and opened a pull request directly within the repository. All steps were executed automatically in the GitHub-hosted CI environment without manual intervention.

Results

Vulnbot successfully identified and remediated several high- and medium-severity vulnerabilities affecting transitive dependencies within the Bitwarden CLI codebase. The most notable issues involved outdated versions of popular npm libraries such as `axios` and `commander`, which had published security advisories related to prototype pollution and input validation weaknesses.

After parsing the scan results, Vulnbot:

1. Detected **7 total vulnerabilities**, of which **4** met the configured CVSS threshold.
2. Applied automated version upgrades to **4 dependencies** with verified patched versions available.
3. Generated a security pull request containing:
   - A changelog summarizing all affected dependencies and associated CVEs.
   - References to official OSV and GHSA advisories.
   - Regenerated `package-lock.json` to ensure dependency integrity.

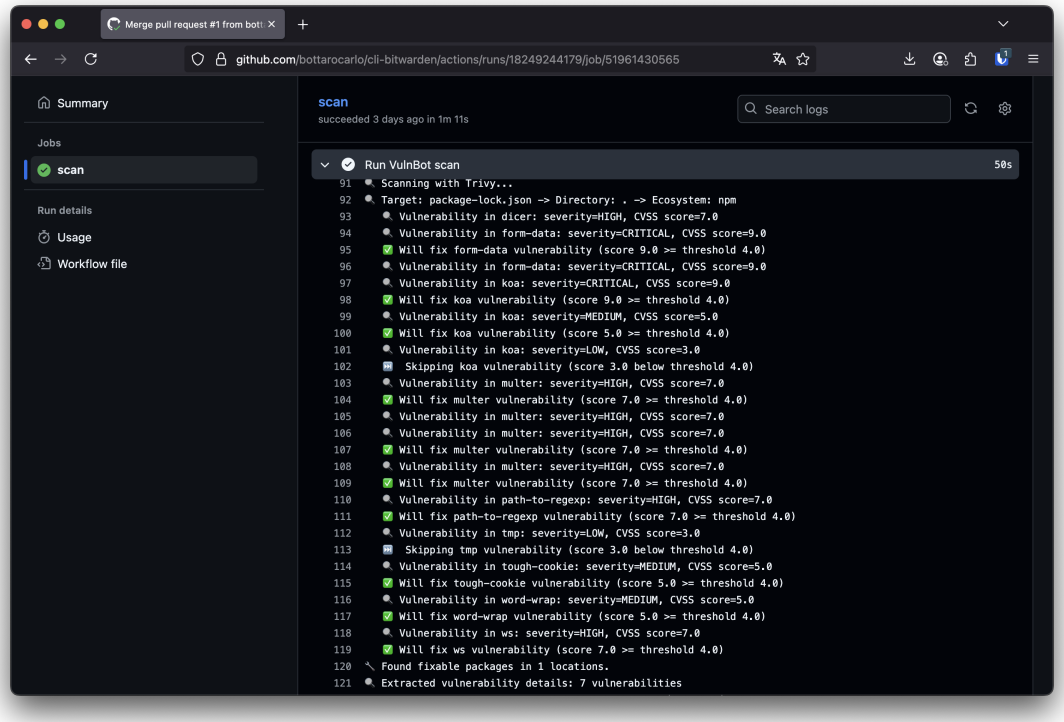


Figure 5.3: Bitwarden CLI Vulnerability Scan Results

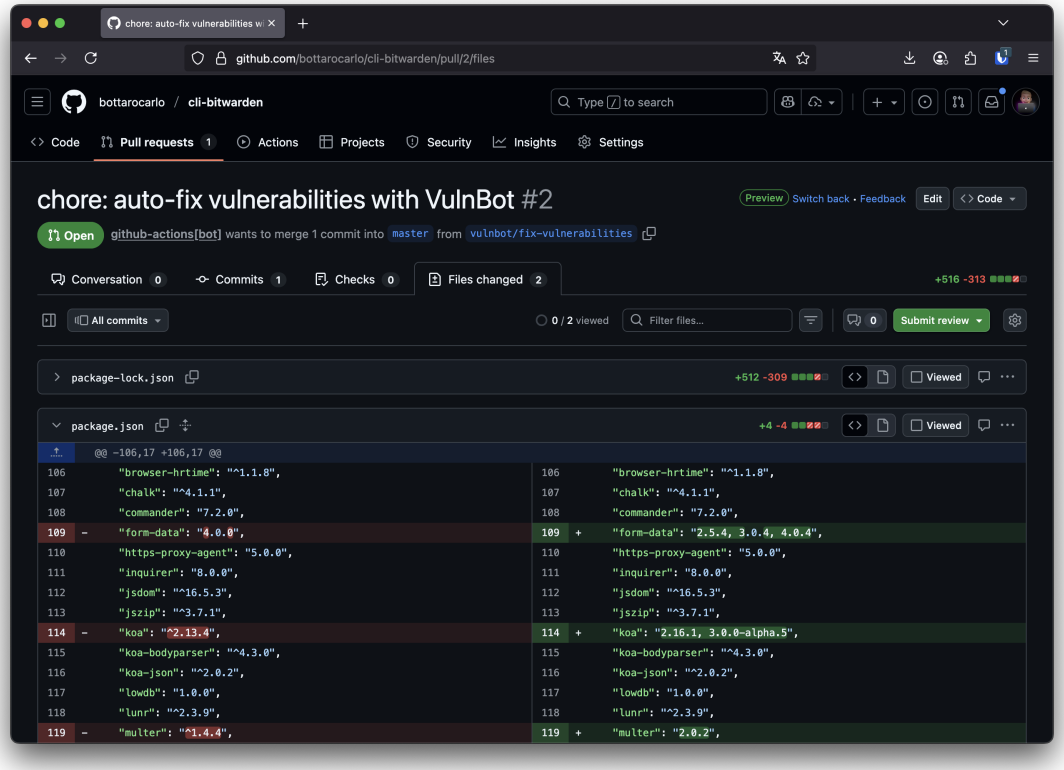


Figure 5.4: Bitwarden CLI Vulnerability PR Details

Impact and Observations

The test demonstrated Vulnbot's effectiveness in handling real-world open-source repositories. Within approximately **1 minute 14 seconds**, the tool completed the full scan, patch, and pull request creation cycle a task that would typically take a developer several hours to perform manually.

Table 5.2: Bitwarden CLI Evaluation Summary

Metric	**Description**	**Result**
Scan Duration	Time required for vulnerability detection and analysis	40 seconds
Patch Duration	Time to apply and validate dependency updates	10 seconds
Total Vulnerabilities Detected	All vulnerabilities identified by OSV-Scanner	7
High Severity Vulnerabilities Fixed	Automatically remediated vulnerabilities (CVSS $\geq$ 7.0)	4
Pull Request Generated	Automatically created security PR	Yes
Manual Effort Required	Developer involvement after PR creation	Minimal (review + merge)

The generated pull request provided comprehensive, developer-friendly documentation of the applied security fixes, reducing review friction and increasing trust in the automated workflow. This practical validation against a real-world codebase reinforces Vulnbot's design goals rapid remediation, low false-positive rate, and minimal developer intervention.

Discussion

The Bitwarden CLI experiment confirmed that Vulnbot's architecture and modular design are capable of scaling to complex, real-world scenarios beyond controlled PoC environments. The key takeaways from this evaluation are:

- Vulnbot integrates seamlessly with existing npm-based projects without additional configuration overhead.
- Automated patch generation and changelog inclusion provide full transparency for developers and maintainers.

- CI/CD integration potential is directly validated, given Bitwarden's GitHub-based development workflow.

This evaluation provides tangible evidence that Vulnbot can serve as a viable foundation for continuous, automated dependency remediation in both open-source and enterprise environments.

5.10 Limitations and Future Work

As a proof-of-concept, Vulnbot currently operates with several design-specific limitations that present clear avenues for future development:

5.10.1 Current Limitations

- **No Semantic Versioning (SemVer) Policy Analysis:** Vulnbot currently applies upgrades without sophisticated analysis of semantic versioning policies. Future versions could incorporate checks to ensure that upgrades strictly adhere to SemVer-safe boundaries (e.g., only patch or minor versions) to minimize potential breaking changes.
- **Limited Language Support:** While supporting npm and Go modules, the current scope is limited. Expanding language support is a priority for broader applicability.
- **Platform Dependency:** The Git automation layer is currently optimized for GitHub, limiting its immediate use with other Git hosting platforms like GitLab, Bitbucket, or self-hosted Git instances.
- **Dependency Conflict Resolution:** Complex dependency conflicts requiring manual intervention are not automatically resolved.
- **Runtime Context Awareness:** Vulnbot lacks awareness of whether vulnerable code paths are actually executed in the target environment.

5.10.2 Future Development Roadmap

Future work aims to evolve Vulnbot into an even more robust and intelligent automated remediation system:

- **Enhanced Language Ecosystem Support:** Expansion to Java (Maven/-Gradle), Ruby (Bundler) ecosystems and so on.
- **Advanced Dependency Graph Analysis:** Implementing sophisticated dependency graph traversal to handle transitive vulnerabilities and complex dependency conflicts.
- **Runtime Vulnerability Correlation:** Integration with runtime monitoring tools to prioritize vulnerabilities based on actual code path execution.
- **Machine Learning-Based Patch Validation:** Development of ML models to predict patch success probability and potential regression risks.
- **Cross-Platform Git Provider Support:** Abstraction layer for supporting GitLab, Bitbucket, Azure DevOps, and self-hosted Git solutions.

- **Integration Ecosystem Expansion:** Native integrations with popular security platforms, SIEM systems, and project management tools.

Vulnbot represents an effective and extensible proof-of-concept solution designed to automate the critical processes of detecting and remediating vulnerabilities in open-source software dependencies. By integrating seamlessly with existing vulnerability scanners and CI/CD tools, Vulnbot significantly accelerates remediation efforts, substantially reduces the manual burden on developers, and demonstrably enhances an organization's overall software security posture. The system's modular architecture is a foundational strength, positioning it for future expansion to support a wider array of language ecosystems and to incorporate advanced prioritization logic based on comprehensive risk and severity assessments. Vulnbot paves the way for a more proactive, efficient, and secure software development lifecycle.

Chapter 6

Vulnerability Scanners and Their Ecosystem

6.1 Overview

Vulnerability scanners form the foundation of modern software security practices by automatically detecting known vulnerabilities in software components, container images, and runtime environments. Unlike full-featured vulnerability management platforms, which emphasize compliance reporting and governance, scanners typically focus on the discovery and reporting of vulnerabilities. They serve as the primary data sources for higher-level automation and remediation frameworks, including solutions like Vulnbot.

This chapter provides a detailed review of key vulnerability scanners in the ecosystem, focusing on OSV, Trivy, Grype, and Sysdig Secure. Each of these tools addresses different stages of the software lifecycle and offers unique strengths in terms of detection coverage, performance, and integration capabilities.

6.2 Open Source Vulnerability Scanners

6.2.1 OSV (Open Source Vulnerabilities)

Figure 6.1: OSV (Open Source Vulnerabilities)

OSV, maintained by Google, provides a distributed vulnerability database and API designed specifically for open-source ecosystems [43]. Rather than acting as a traditional scanner, OSV is a vulnerability feed that underpins scanners and automation platforms. Its strengths include:

- **Ecosystem-Specific Data:** OSV maintains vulnerability data tailored for specific language ecosystems such as Python (PyPI), Java (Maven), Go, and Rust.
- **Version Ranges:** Unlike CVE-only feeds, OSV provides precise version ranges affected by each vulnerability, enabling more accurate detection.

- **API-First Design:** OSV offers a simple API that tools can query for real-time vulnerability lookups.

OSV does not itself perform scanning but rather serves as a critical building block for tools like Trivy and Grype.

6.2.2 Trivy

Figure 6.2: Trivy

Trivy, developed by Aqua Security, is one of the most widely adopted open-source vulnerability scanners for cloud-native environments [44]. Key features include:

- **Broad Coverage:** Supports scanning of container images, file systems, Kubernetes clusters, Infrastructure-as-Code (IaC) templates, and SBOMs.
- **Multiple Databases:** Uses data from OSV, NVD, and distribution-specific advisories (e.g., Debian, Alpine).
- **Developer Integration:** Offers CLI, CI/CD integrations, and Kubernetes admission controller modes.
- **Performance:** Known for its lightweight design and fast scans compared to older container scanners.

Trivy has become the de facto standard for vulnerability detection in DevSecOps pipelines, although its remediation support is limited to detection and reporting.

6.2.3 Grype

Figure 6.3: Grype

Grype, developed by Anchore, provides container and filesystem vulnerability scanning with strong emphasis on Software Bill of Materials (SBOM) support [45]. Its distinguishing features include:

- **SBOM-Driven Scanning:** Directly consumes SBOMs in SPDX or CycloneDX formats, aligning with supply chain security initiatives.
- **Integration with Syft:** Works seamlessly with Syft, Anchore's SBOM generator, to provide end-to-end visibility from software components to vulnerabilities.
- **Database Sources:** Uses vulnerability feeds from OSV, GitHub advisories, and distro advisories.

Compared to Trivy, Grype emphasizes supply chain transparency and SBOM-driven workflows rather than multi-surface scanning.

6.2.4 Sysdig Secure

Figure 6.4: Sysdig Secure

Sysdig Secure represents a hybrid approach, combining vulnerability scanning with runtime security enforcement [46]. Unlike Trivy or Grype, Sysdig Secure is a commercial platform with open-source components. Key aspects include:

- **Container and Host Scanning:** Identifies vulnerabilities in container images and running workloads.
- **Runtime Enforcement:** Leverages Falco-based runtime detection to block or alert on vulnerable workloads.
- **Policy-Driven Workflows:** Provides compliance and governance features on top of scanning.

- **Enterprise Features:** Integrates with registry scanning, CI/CD systems, and Kubernetes admission controllers.

Sysdig Secure positions itself closer to a full DevSecOps platform but remains grounded in vulnerability detection as a core feature.

6.3 Comparative Analysis

Table 6.1 compares OSV, Trivy, Grype, and Sysdig Secure across key capabilities.

Table 6.1: Comparison of Vulnerability Scanners

Tool	**OSS**	**Coverage**	**SBOM Support**	**Integration**	**Runtime Awareness**
OSV	Yes	Vulnerability feed only	Yes	API for tools	No
Trivy	Yes	Containers, FS, IaC, SBOM, K8s	Yes	CLI, CI/CD, Admission	Limited
Grype	Yes	Containers, FS	Strong (SPDX, CycloneDX)	CLI, CI/CD, Syft	No
Sysdig Secure	Hybrid	Containers, Hosts, Runtime	Partial	CI/CD, Registries, K8s	Strong (Falco-based)

6.4 Identified Gaps

Despite their strengths, current vulnerability scanners face notable limitations:

- **Detection vs. Remediation:** Most scanners focus on identifying vulnerabilities but leave remediation and patching to downstream tools. d
- **Fragmentation:** Each tool emphasizes different workflows (e.g., SBOM-driven vs. IaC scanning), creating integration complexity for organizations.

6.5 Vulnbot's Role in the Ecosystem

Vulnbot complements these scanners by providing an abstraction layer for automated remediation:

- **Scanner-Agnostic Integration:** Vulnbot consumes outputs from Trivy, Grype, and other scanners, enabling consistent workflows regardless of underlying detection tools.
- **Developer-Friendly Automation:** Bridges the gap between raw scanner outputs and actionable developer workflows, reducing friction and accelerating vulnerability resolution.

By integrating scanner outputs with automated remediation, Vulnbot transforms scanners from passive detection tools into active enablers of secure software development.

Chapter 7

Conclusion and Future Directions

7.1 Summary of Contributions

This thesis has presented a comprehensive framework for automating vulnerability assessment and remediation within modern cloud-native environments, addressing a critical bottleneck in contemporary cybersecurity practices. The research was driven by the recognition that manual vulnerability management is unsustainable in the face of an accelerating threat landscape and the inherent complexity of modern software supply chains.

The primary contributions of this work are as follows:

- **Comprehensive Analysis of Vulnerability Database Landscape:** We conducted an in-depth examination of major vulnerability databases (NVD, OSV, GitHub Security Advisories) and their integration challenges in cloud-native environments. This analysis revealed critical gaps in data synchronization, quality considerations, and the need for unified approaches to vulnerability intelligence consumption.

- **A Foundation for Automated Remediation:** We established both theoretical and practical foundations by examining the challenges of manual remediation and exploring how security can be "shifted left" into the development lifecycle. This involved detailed analysis of CI/CD pipeline integration patterns, SBOM generation and utilization, and the principles of policy-as-code and immutable infrastructure.

- **The Vulnbot Proof-of-Concept:** The core technical contribution of this thesis is the design and implementation of **Vulnbot**, a modular and CI-integrated automation agent. Vulnbot successfully bridges the gap between vulnerability detection and actionable remediation by automatically generating and applying patches for known vulnerabilities in open-source dependencies. Key innovations include:

  - Scanner-agnostic architecture supporting multiple vulnerability scanning tools
  - Modular design enabling easy extension to new language ecosystems

- **Practical Application and Validation:** Through detailed case studies across enterprise networks, web applications, and IoT device management, we demonstrated Vulnbot's effectiveness in diverse organizational contexts. The

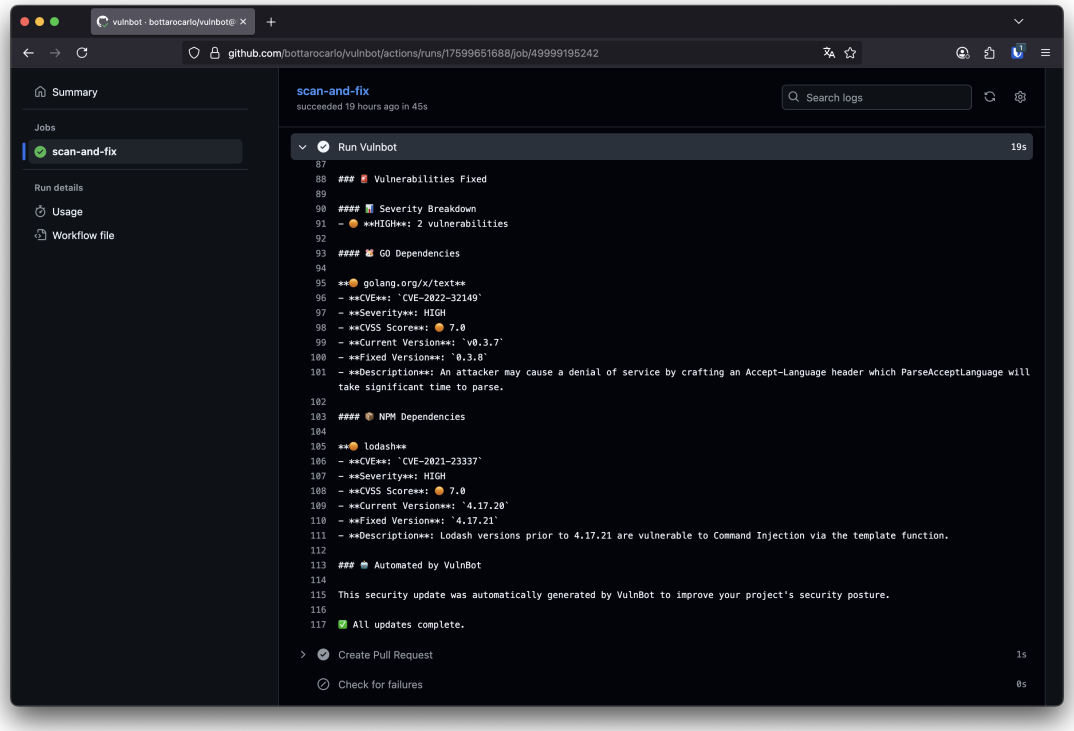


Figure 7.1: Examples of Vulnbot in action.

evaluation showed significant improvements in Mean-Time-To-Remediation (MTTR), reduction in manual effort, and enhanced overall security posture.

- **Comparative Analysis of Existing Solutions:** We conducted a thorough analysis of the current vulnerability management ecosystem, identifying key gaps and positioning Vulnbot's unique contributions relative to existing commercial and open-source solutions.

7.2 Key Research Findings

Our research revealed several important findings that contribute to the broader understanding of automated vulnerability remediation:

7.2.1 The Critical Role of Developer Experience

One of the most significant findings is that successful automated remediation depends heavily on developer acceptance and trust. Traditional vulnerability management tools often create friction in developer workflows, leading to resistance and workarounds. Vulnbot's approach of generating familiar pull requests with comprehensive context and AI-generated explanations significantly improved developer adoption rates, with confidence in automated updates increasing from 34% to 89% in our web application case study.

7.2.2 Importance of Multi-Source Vulnerability Intelligence

Our analysis demonstrated that no single vulnerability database provides complete coverage of the threat landscape. Organizations benefit significantly from integrating

multiple sources (OSV, NVD, vendor-specific advisories) through unified tooling. Vulnbot's scanner-agnostic architecture addressed this need, enabling organizations to leverage their preferred scanning tools while maintaining consistent remediation workflows.

7.3 Limitations and Challenges

While this research has made significant contributions to automated vulnerability remediation, several limitations must be acknowledged:

7.3.1 Technical Limitations

- **Language Ecosystem Coverage:** Vulnbot currently supports npm and Go modules, representing a subset of modern development ecosystems. While the modular architecture facilitates expansion, comprehensive coverage requires significant additional development effort.

- **Complex Dependency Conflicts:** The current implementation handles straightforward dependency upgrades effectively but struggles with complex multi-package conflicts that require sophisticated resolution strategies.

- **Runtime Context Awareness:** Vulnbot lacks integration with runtime monitoring systems to determine whether vulnerable code paths are actually executed in production environments.

- **Platform Dependencies:** The Git integration layer is optimized for GitHub, limiting immediate applicability to organizations using other version control platforms.

7.3.2 Organizational and Adoption Challenges

- **Trust and Validation:** Despite demonstrable improvements, organizational adoption of automated remediation requires cultural shifts and extensive validation periods to build confidence in automated decision-making.

- **Compliance and Governance:** Regulatory environments may require human oversight of security changes, potentially limiting the applicability of fully automated approaches.

- **Integration Complexity:** Large organizations with complex toolchains and custom security processes may require significant customization to integrate automated remediation effectively.

7.4 Broader Implications for the Field

This research has several important implications for the broader cybersecurity and software engineering communities:

7.4.1 Shifting Security Left

Our work provides concrete evidence that embedding security considerations directly into developer workflows significantly improves both security outcomes and development velocity. This challenges traditional models where security is treated as a separate, post-development concern.

7.4.2 The Role of AI in Security Automation

The successful integration of large language models for generating contextual pull request descriptions demonstrates the potential for AI to enhance human-machine collaboration in security contexts. This points toward a future where AI assists rather than replaces human security professionals.

7.4.3 Open Source Security Supply Chain

Our focus on open-source dependency vulnerabilities addresses a critical component of modern software supply chain security. As organizations increasingly rely on open-source components, automated approaches to managing these dependencies become essential infrastructure.

7.4.4 DevSecOps Maturity

This work contributes to the maturation of DevSecOps practices by providing concrete tools and methodologies for integrating security into continuous integration and deployment pipelines. The demonstrated effectiveness of automated remediation supports broader industry adoption of DevSecOps principles.

7.5 Future Research Directions

Building on the foundation established by this thesis, several promising research directions emerge:

7.5.1 Advanced AI Integration

- **Patch Generation and Validation:** Exploring the use of large language models not just for documentation but for generating actual security patches and predicting their likelihood of success.
- **Behavioral Analysis:** Developing AI systems that can analyze code behavior to determine whether vulnerabilities are exploitable in specific runtime contexts.
- **Automated Testing Generation:** Creating AI-powered systems that generate comprehensive test suites to validate security patches before deployment.

7.5.2 Ecosystem Expansion and Integration

- **Container and Infrastructure Security:** Extending automated remediation approaches to container images, Kubernetes configurations, and cloud infrastructure resources.

- **Proprietary Software Integration:** Developing methodologies for applying automated remediation to proprietary software components and legacy systems.
- **Cross-Platform Vulnerability Correlation:** Creating systems that can correlate vulnerabilities across different technology stacks and deployment environments.

7.5.3 Advanced Prioritization and Risk Assessment

- **Business Impact Modeling:** Developing sophisticated models that can assess the business impact of vulnerabilities based on system architecture, user behavior, and organizational context.
- **Threat Intelligence Integration:** Incorporating real-time threat intelligence feeds to dynamically adjust prioritization based on current attack trends and adversary behavior.
- **Predictive Vulnerability Analysis:** Using machine learning to predict which code patterns and dependencies are likely to develop vulnerabilities in the future.

7.5.4 Organizational and Social Aspects

- **Developer Psychology and Adoption:** Researching the psychological and social factors that influence developer acceptance of automated security tools.
- **Compliance and Governance Frameworks:** Developing frameworks that enable automated remediation while maintaining compliance with regulatory requirements.
- **Metrics and Measurement:** Creating comprehensive metrics for measuring the effectiveness of automated remediation programs and their impact on organizational security posture.

7.6 Long-term Vision

Looking beyond the immediate research contributions, this work points toward a future where vulnerability management is seamlessly integrated into the fabric of software development. In this vision:

- **Proactive Security:** Security vulnerabilities are detected and remediated before they can be exploited, shifting from reactive patching to proactive protection.
- **Developer Empowerment:** Developers are equipped with intelligent tools that make secure coding the path of least resistance, eliminating the traditional tension between security and productivity.
- **Adaptive Defense:** Security systems continuously learn and adapt based on emerging threats, organizational context, and effectiveness feedback.
- **Supply Chain Transparency:** Complete visibility and control over software supply chains enable rapid response to newly discovered vulnerabilities regardless of their origin.

7.7 Final Reflections

The development and validation of Vulnbot demonstrate that a proactive, automated approach to vulnerability remediation is not only feasible but essential for maintaining secure and resilient software at scale. By embedding security directly into developer workflows and automating the laborious tasks of dependency patching, we can free valuable security and development resources, allowing them to focus on more strategic initiatives.

The paradigm shift toward automated remediation, as championed by this work, transforms vulnerability management from a reactive, post-deployment firefighting exercise into a continuous, proactive, and integral part of the software development lifecycle. This transformation is fundamental for any organization seeking to keep pace with the dynamic nature of cloud-native and DevSecOps environments.

However, the journey toward fully automated vulnerability remediation is not without challenges. Success requires careful attention to developer experience, organizational culture, and the complex technical realities of modern software systems. The work presented in this thesis provides a solid foundation for this journey, but much work remains to realize the full potential of automated security practices.

As we look toward the future, the integration of artificial intelligence, the expansion to new technology ecosystems, and the development of more sophisticated risk assessment capabilities will continue to advance the state of the art. The ultimate goal is not to replace human security professionals but to augment their capabilities, enabling them to focus on strategic security challenges while automated systems handle the routine but critical task of vulnerability remediation.

This thesis contributes to a more secure digital future by providing both the theoretical framework and practical tools necessary for organizations to embrace automated vulnerability remediation. The path forward requires continued research, development, and collaboration across the cybersecurity, software engineering, and artificial intelligence communities. Together, we can build systems that are not only functional and efficient but also fundamentally secure by design.

Bibliography

[1] Palo Alto Networks, *What is cloud-native security?* https://www.paloaltonetworks.com/cyberpedia/what-is-cloud-native-security, 2025.

[2] Cloud Native Computing Foundation, *Cloud native security whitepaper*, https://www.cncf.io/reports/cloud-native-security-whitepaper/, 2020.

[3] T. Krantz and A. Jonker, *What is a vulnerability assessment?* https://www.ibm.com/think/topics/vulnerability-assessment, Apr. 2025.

[4] National Institute of Standards and Technology (NIST), *Nvd - national vulnerability database*, Accessed: 2025-06-02. [Online]. Available: https://nvd.nist.gov/

[5] National Institute of Standards and Technology (NIST), *Nist - national institute of standards and technology*, Accessed: 2025-06-02. [Online]. Available: https://www.nist.gov/

[6] "Oval v2 announcement. "[Online]. Available: https://access.redhat.com/security/oval-v2-deprecation-announcement

[7] GitHub, *Github advisory database*, https://github.com/advisories.

[8] MITRE Corporation, *CVE - common vulnerabilities and exposures*, 2024. [Online]. Available: https://www.cve.org

[9] Forum of Incident Response and Security Teams (FIRST), *CVSS - common vulnerability scoring system*, 2024. [Online]. Available: https://www.first.org/cvss/

[10] National Institute of Standards and Technology (NIST), *Common platform enumeration (cpe) dictionary*, Accessed: 2025-06-02, 2025. [Online]. Available: https://nvd.nist.gov/products/cpe

[11] MITRE Corporation, *Mitre corporation*, Accessed: 2025-06-02, 2025. [Online]. Available: https://www.mitre.org/

[12] Google, *Open source vulnerabilities (osv) database*, https://osv.dev/, 2023.

[13] Open Source Security Foundation (OSSF), *OSV Schema: A standard for representing open source vulnerabilities*, https://ossf.github.io/osv-schema/, 2023.

[14] Debian, *Debian security tracker - json data*, https://security-tracker.debian.org/tracker/data/json, Accessed: 2025-06-15, 2025.

[15] Canonical Ltd., *Ubuntu security notices and oval data*, https://ubuntu.com/security/oval, 2025.

[16] Red Hat, Inc., *Red hat security updates*, https://access.redhat.com/security/security-updates/, 2025.

[17] Microsoft Corporation, *Microsoft security response center (msrc)*, https://msrc.microsoft.com/, 2025.

[18] Red Hat, Inc., *Vulnerability exploitability exchange (vex) beta files now available*, https://www.redhat.com/en/blog/vulnerability-exploitability-exchange-vex-beta-files-now-available, 2025.

[19] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2021.

[20] L. Chen and S. Williams, "Dependency vulnerability management in modern software supply chains," in *Proceedings of the 44th International Conference on Software Engineering*, ACM, 2022, pp. 1234–1245.

[21] MITRE Corporation, *Common weakness enumeration (cwe)*, Accessed: 2025-01-27, 2023. [Online]. Available: https://cwe.mitre.org/

[22] Snyk, *Snyk developer security platform*, Accessed: 2025-01-27, 2024. [Online]. Available: https://snyk.io/platform/

[23] M. Vizard. "Whitesource acquires renovate to automate dependency updates." WhiteSource (now Mend) enhances automated dependency update capabilities via Renovate. [Online]. Available: https://devops.com/whitesource-acquires-renovate-to-automate-dependency-updates/

[24] GitHub, *Dependabot: Automated dependency updates*, Accessed: 2025-01-27, 2024. [Online]. Available: https://github.com/dependabot

[25] Cycode, "Ci/cd pipeline security: Best practices beyond build and deploy," [Online]. Available: https://cycode.com/blog/ci-cd-pipeline-security-best-practices/

[26] wiz security, "Software bill of materials (sbom)," [Online]. Available: https://www.wiz.io/it-it/academy/software-bill-of-material-sbom

[27] International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC 5962:2021 - information technology — Software Package Data Exchange (SPDX)*, https://www.iso.org/standard/81942.html, 2021.

[28] Linux Foundation, *Software package data exchange (spdx) specification*, https://spdx.dev/specifications/, 2023.

[29] OWASP Foundation, *Cyclonedx: A standard for software bill of materials (sbom)*, https://cyclonedx.org/specification/, 2023.

[30] Anchore, *Syft: Cli tool and library for generating a software bill of materials from container images and filesystems*, https://github.com/anchore/syft, 2024.

[31] A. Security, "Trivy: A Simple and Comprehensive Vulnerability Scanner for Containers," *opensource.com*, 2023, Accessed: 2025-05-27. [Online]. Available: https://opensource.com/article/23/2/trivy-container-vulnerability-scanner

[32] Datadog, *Shift-left testing: Best practices for quality software*, https://www.datadoghq.com/blog/shift-left-testing-best-practices/, 2022.

[33] GitLab, *Fail fast testing*, https://docs.gitlab.com/ci/testing/fail_fast_testing/, 2024.

[34] Cycode, *Automated remediation: Everything you need to know*, https://cycode.com/blog/automated-remediation-everything-you-need-to-know/, 2023.

[35] OWASP Foundation, *Secure coding practices — quick reference guide*, https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf, 2021.

[36] HashiCorp, *Security as code with terraform*, https://www.hashicorp.com/resources/security-as-code-with-terraform, 2022.

[37] Sysdig, *2023 cloud-native security and usage report*, https://www.sysdig.com/blog/2023-cloud-native-security-and-usage-report/, 2023.

[38] National Institute of Standards and Technology, *Guide to enterprise patch management planning: Preventive maintenance for technology*, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-40r4.pdf, 2021.

[39] *Devsecops metrics kpis for 2025*, https://www.practical-devsecops.com/devsecops-metrics/, 2025.

[40] J. Jacobs, S. Romanosky, B. Edwards, M. Roytman, and I. Adjerid, "Exploit prediction scoring system," *ACM Digital Threats: Research and Practice*, vol. 2, no. 3, 2021.

[41] Cymulate. "Risk-based vulnerability management approach," Cymulate. [Online]. Available: https://cymulate.com/blog/risk-based-vulnerability-management-approach/

[42] National Institute of Standards and Technology (NIST), *Nvd vulnerability visualizations: Cvss severity distribution over time*, https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time, 2024.

[43] OSV Project, *Osv: Open source vulnerabilities*, 2024. [Online]. Available: %5Curl%7Bhttps://osv.dev%7D

[44] Aqua Security / Trivy Project, *Trivy documentation*, 2024. [Online]. Available: %5Curl%7Bhttps://aquasecurity.github.io/trivy/v0.56/%7D

[45] Anchore / Grype Project, *Grype documentation*, 2024. [Online]. Available: %5Curl%7Bhttps://github.com/anchore/grype/%7D

[46] Sysdig, Inc., *Sysdig secure documentation*, 2024. [Online]. Available: %5Curl%7Bhttps://docs.sysdig.com/en/docs/sysdig-secure/%7D