A PRACTICAL GUIDE TO RISK ASSESSMENT, PRIORITIZATION, AND REMEDIATION OF DEPENDENCIES

SECURE CODING PRACTICES

WHITEPAPER

Managing Vulnerable Libraries in Enterprise Codebases

HAROON MANSOORI

DISCLAIMER

The views and opinions expressed in this whitepaper are those of the author and do not necessarily reflect the official policy or position of any employer, organization, company, or security tool vendor mentioned. This includes Snyk.io, Grit.io, and any other security tools or platforms discussed.

The information provided is based on the author's personal experience, observations, and research. It should not be considered as legal, professional, or security advice. The security landscape is constantly evolving, and practices that are effective at the time of writing may need to be adapted as new threats and technologies emerge.

Readers are encouraged to:

1. Seek the advice of qualified professionals before making any decisions based on the content of this whitepaper.
2. Conduct their own research and testing to verify the applicability and effectiveness of the strategies discussed in their specific contexts.
3. Stay informed about the latest developments in application security and continuously reassess their security practices.

The author and any associated organizations disclaim any liability for any damages or losses that may result from the use or misuse of the information contained in this whitepaper.

CONTENTS

MANAGING VULNERABLE LIBRARIES IN ENTERPRISE CODEBASES

A PRACTICAL GUIDE TO RISK ASSESSMENT, PRIORITIZATION, AND REMEDIATION OF 3RD PARTY DEPENDENCIES

SECURE CODING WHITEPAPER

HAROON MANSOORI

If you ignore security in your DevOps pipeline,
You're not delivering features,
You're delivering vulnerabilities.

And they weren't talking about code quality.

This is a systemic issue that goes beyond just writing secure code. It touches on processes, culture, and overall approach to software development.

Reconsider your priorities and practices.

CHAPTER 1
OVERVIEW

In modern software development, the use of third-party libraries and frameworks is ubiquitous. These components, often open-source, accelerate development but introduce potential security risks. This document addresses the challenge of managing vulnerabilities in these dependencies within large-scale enterprise applications, particularly when upgrading is not straightforward.

The problem we're solving is how to effectively manage and mitigate security risks associated with third-party libraries in complex, interconnected enterprise systems, especially when immediate upgrades are impractical or risky.

CHAPTER 2
DETAILED EXPLANATION AND CHALLENGES

THE DEPENDENCY ECOSYSTEM

Enterprise applications today are built on a foundation of numerous third-party libraries and frameworks. These dependencies serve various purposes, from handling low-level operations to providing high-level functionalities. While they significantly speed up development and reduce the need to "reinvent the wheel," they also introduce external code into the application's ecosystem.

Sources of these dependencies include:

1. Open-source repositories (e.g., npm, PyPI, Maven Central)
2. Commercial libraries
3. Code snippets from forums or other online sources
4. Internal shared libraries within the organization

The extensive use of these dependencies creates a complex web of interconnected components, each with its own update cycle, potential vulnerabilities, and maintenance status.

VULNERABILITY MANAGEMENT IN ENTERPRISE CONTEXTS

In large organizations, applications are often built within a predefined architectural framework. This framework may dictate:

- Approved libraries and versions
- Coding standards and practices
- Integration patterns with other systems

When a vulnerability is discovered in a library used across multiple systems, the impact can be far-reaching. The traditional approach of "update to the latest version" becomes complicated in enterprise environments due to several factors:

1. **Interconnected Systems**: Changes in one component may have ripple effects across multiple systems.
2. **Legacy Code**: Older systems may rely on specific versions of libraries that are no longer compatible with the latest releases.
3. **Compliance Requirements**: Certain industries have strict regulations about software changes, requiring extensive testing and documentation.
4. **Resource Constraints**: Updating and testing across all affected systems can be time-consuming and expensive.

THE ROLE OF SECURITY TOOLS

To manage these risks, many organizations employ Software Composition Analysis (SCA) tools, such as Mend (formerly Whitesource) or Snyk. These tools:

- Scan codebases to identify third-party components
- Cross-reference components against known vulnerability databases
- Alert teams to newly discovered vulnerabilities
- Provide recommendations for updates or patches

While these tools are invaluable, they also present challenges:

- **False Positives**: Not all identified vulnerabilities may be exploitable in the specific context of the application.
- **Update Conflicts**: Recommended updates may conflict with other dependencies or application requirements.
- **Overload**: In large systems, the volume of alerts can be overwhelming, leading to alert fatigue.

CHALLENGES IN DETAIL

1. **Version Compatibility**: Major version updates often include breaking changes. Upgrading a core library might require extensive refactoring across multiple systems.

2. **Testing Complexity**: In interconnected systems, a change in one component necessitates testing not just that component, but all connected systems. This can be a massive undertaking.

3. **Operational Risk**: Updating a library in a production system carries the risk of introducing new bugs or breaking existing functionality.

4. **Cost-Benefit Analysis**: The effort required to update a library must be weighed against the actual risk posed by the vulnerability in the specific context of the application.

5. **Continuous Integration/Continuous Deployment (CI/CD) Pipelines**: Automated build and deployment processes may need to be updated to accommodate new library versions.

6. **Documentation and Knowledge Transfer**: Changes in library versions often require updates to internal documentation and knowledge bases.

. . .

7. **Vendor Lock-in**: Some commercial libraries or frameworks may have vulnerabilities, but switching to alternatives could be prohibitively expensive.

8. **Cultural Resistance**: Development teams may resist frequent updates, viewing them as disruptive to their workflow.

9. **Compliance and Auditing**: In regulated industries, every change may require documentation and approval, slowing down the update process.

10. **Technical Debt Accumulation**: Delaying updates can lead to a buildup of technical debt, making future updates even more challenging.

11. **Volume of Dependencies**: Modern applications often rely on hundreds or even thousands of libraries, either directly or as transitive dependencies. This sheer volume makes comprehensive management and upgrading a daunting task.

12. **Organizational Constraints**: Various organizational factors can impede the ability to upgrade libraries:

- Strict change control processes that limit the frequency or scope of updates.
- Budgetary constraints that prevent allocation of resources for library management.
- Strategic decisions to maintain compatibility with specific versions for business reasons.

13. **Resource Limitations**: Development teams often face significant resource constraints:

- Developer bandwidth is typically allocated well in advance,

sometimes for multiple quarters, leaving little room for unplanned library upgrades.
- The extensive testing required for library upgrades often exceeds available quality assurance resources.
- Hiring additional staff to manage library upgrades may not be feasible due to budget constraints, lack of planning, or difficulties in sourcing qualified personnel quickly.

14. **Competing Priorities**: In many organizations, feature development and bug fixes take precedence over library upgrades, especially when the impact of vulnerabilities isn't immediately apparent.

15. **Lack of Expertise**: Teams may lack the specific expertise needed to evaluate the impact of library upgrades, especially for complex or specialized libraries.

16. **Obscure or Poorly Maintained Libraries**: Some projects may depend on unpopular or niche libraries that have poorly written, buggy, or unmaintained code. Developers may not have a direct way of knowing about these issues, making it difficult to assess the risks associated with these dependencies.

17. **Unused Imported Dependencies**: Many programming languages package imported dependencies through libraries, some of which may never be used in the application's codebase. These unnecessary dependencies increase the attack surface and complicate the upgrade process.

18. **Volume of Libraries**: Projects often use a large number of libraries, each with its own release cycle and dependencies. Keeping up with all the updates can be overwhelming for development teams.

19. **Compatibility Issues**: Upgrading one library might necessitate upgrading others or might not be compatible with other parts of the

system. This can lead to situations where teams cannot upgrade a library due to breaking changes, API modifications, or dependencies that require a specific version of a library.

20. **Testing Overhead**: Developers need to thoroughly test the application after upgrading libraries to ensure that changes do not break the application. This can be time-consuming, especially for large applications or when there are significant changes in the library.

CHAPTER 3
BEST PRACTICES FOR REMEDIATION

Addressing the challenges of managing vulnerable libraries in enterprise environments requires a multi-faceted approach. Here are some best practices:

1. **Implement a Robust Vulnerability Management Program**

- Establish a dedicated team or role responsible for vulnerability management.
- Define clear processes for identifying, assessing, and addressing vulnerabilities.
- Regularly scan codebases using SCA tools to identify vulnerable components.
- Prioritize vulnerabilities based on their severity and the specific context of your applications.

2. **Adopt a Risk-Based Approach**

Not all vulnerabilities pose the same level of risk. Assess each vulnerability in the context of your application:

- Is the vulnerable component directly exposed to user input?
- Does it handle sensitive data?
- What are the potential consequences of exploitation?

- Focus remediation efforts on high-risk vulnerabilities first.

3. **Implement Compensating Controls**

When immediate updates are not feasible, consider implementing additional security measures:

- Network segmentation to limit exposure
- Input validation to prevent exploitation of known vulnerabilities
- Runtime application self-protection (RASP) tools

4. **Maintain a Comprehensive Inventory**

- Keep an up-to-date inventory of all third-party components used across your applications.
- Include version information, known vulnerabilities, and any compensating controls in place.
- This inventory will be invaluable during security audits and when planning updates.

5. **Establish a Patching Strategy**

Define different approaches for minor updates (which often fix vulnerabilities) and major updates (which may introduce breaking changes):

- Aim to apply minor updates quickly and regularly.
- Plan major updates as part of your application's roadmap, allowing time for thorough testing and refactoring.

6. **Automate Where Possible**

Use automated tools to:

- Detect new vulnerabilities
- Test compatibility of updates

- Generate reports for stakeholders
- Integrate vulnerability scanning into your CI/CD pipeline to catch issues early.

7. **Implement Gradual Rollout Strategies**

- For critical updates, consider strategies like canary releases or blue-green deployments to minimize risk.
- This allows you to test updates in a production-like environment with limited exposure.

8. **Foster a Security-Conscious Culture**

- Provide regular training to development teams on secure coding practices and the importance of dependency management.
- Encourage developers to consider security implications when introducing new dependencies.

9. **Develop and Maintain Internal Libraries**

- For core functionalities used across multiple projects, consider developing and maintaining internal libraries.
- This gives you more control over security and reduces dependency on external components.

10. **Establish a Vulnerability Disclosure Policy**

- Create a clear process for external researchers to report vulnerabilities in your applications.
- This can help you identify and address issues that automated tools might miss.

. . .

11. **Regular Security Assessments**

- Conduct periodic security assessments of your applications, including penetration testing.
- This can help identify vulnerabilities that may not be detected by automated tools.

12. **Vendor Management**

- For commercial libraries, establish relationships with vendors to stay informed about security updates.
- Consider negotiating support contracts that include security patching commitments.

13. **Plan for Library Deprecation**

- Develop strategies for phasing out libraries that are no longer maintained or have chronic security issues.
- This might involve gradual refactoring or planning for larger modernization efforts.

14. **Continuous Monitoring and Improvement**

- Regularly review and refine your vulnerability management processes.
- Stay informed about new threats and evolving best practices in the security community.

15. **Prioritize and Triage**

Given resource constraints, it's crucial to prioritize library upgrades based on risk and impact:

- Focus on libraries with critical vulnerabilities that are actively exploited in the wild
- Prioritize upgrades for libraries that handle sensitive data or are exposed to untrusted input
- Consider the usage context of each library when assessing risk
- Develop a triage process to quickly assess and categorize new vulnerability reports

16. **Leverage Automation for Resource Efficiency**

Invest in tools that can automate parts of the upgrade process:

- Use dependency update bots (like Dependabot) to automatically create pull requests for non-breaking updates
- Implement automated compatibility testing to quickly identify potential issues with upgrades
- Use static analysis tools to detect usage patterns of libraries, helping prioritize upgrades based on actual usage

17. **Implement Gradual and Partial Upgrades**

When full upgrades aren't feasible, consider strategies for partial improvements:

- Upgrade to intermediate versions that address critical vulnerabilities but don't require extensive refactoring
- Implement library upgrades gradually across different modules or services to spread the work over time
- Use dependency resolution tools to upgrade only specific vulnerable components when possible

18. **Build a Case for Resources**

Develop a strong business case for dedicating resources to library management:

- Quantify the potential cost of security breaches due to known vulnerabilities
- Highlight regulatory compliance requirements that mandate addressing known vulnerabilities
- Demonstrate how proactive library management can reduce long-term technical debt and development costs

19. **Leverage Cross-Functional Collaboration**

Foster collaboration between development, security, and operations teams:

- Create a dedicated working group for dependency management that meets regularly
- Encourage knowledge sharing about library upgrades across different teams and projects
- Involve product owners and stakeholders in prioritizing security-related work

20. **Implement a Library Governance Model**

Establish guidelines for introducing new libraries to minimize future upgrade challenges:

- Create a process for vetting and approving new library dependencies
- Maintain a list of preferred libraries that are known to be well-maintained and security-focused
- Consider limiting the number of allowed libraries for common functionalities to reduce complexity

21. **Explore Alternative Mitigation Strategies**

When upgrades are not immediately feasible, investigate alternative approaches:

- Use virtual patching at the application or network level to mitigate specific vulnerabilities
- Implement additional monitoring and alerting for systems using vulnerable libraries
- Consider containerization or micro-segmentation to isolate vulnerable components

22. **Prioritization Strategies for Dependency Upgrades**

- **Direct vs Transitive Dependencies**: Focus on direct dependencies first, as these are the ones your codebase explicitly calls and relies on. Address transitive dependencies in a secondary wave or when they have known critical issues.
- **Business Value Assessment**: Prioritize upgrades for applications that are critical to business operations or have a high impact on users. Maintain more rigorous update schedules for applications that directly generate revenue or are mission-critical.
- **External vs Internal Applications**: Prioritize external-facing applications for upgrades due to higher risk of exposure to security vulnerabilities. While internal applications may have some leeway, they should not be neglected.
- **Risk Assessment Formula**: Employ a formula like Risk = Probability x Impact to assess the risk of not upgrading a dependency. Consider both the exploitability of the outdated dependency (Probability) and the potential damage it could cause (Impact).

23. **Implement Reachability Analysis**

Reachability analysis is a powerful method to understand and prioritize vulnerabilities based on the actual use and reachability of vulnerable code paths in your application. Implement the following steps:

- **Dependency Graph from Manifests**: Create a graph of all dependencies as declared in your project's manifest files (e.g., package.json, requirements.txt).
- **Code Scan**: Perform a comprehensive code scan to identify which dependencies are actually used in the code. This helps distinguish between declared and effectively used dependencies.
- **Create a Dependency Graph**: Generate a comprehensive graph that includes all dependencies and their interconnections.
- **Traverse the Graph for Function Calls**: Analyze the graph to understand the chain of function calls. This helps identify if vulnerable functions are ever called in the application's normal operation.

Use this analysis to make informed decisions about which upgrades are most critical based on actual usage and potential impact.

24. **Manage Obscure and Unused Dependencies**

- Regularly audit your project for obscure or poorly maintained libraries. Consider replacing these with more popular, well-maintained alternatives when possible.
- Implement a process to identify and remove unused imported dependencies. This can involve static code analysis tools and regular code reviews.
- For necessary but poorly maintained libraries, consider forking and maintaining a version internally, or contributing to the open-source project to improve its quality.

25. **Streamline Testing for Library Upgrades**

- Develop a comprehensive suite of automated tests that can quickly validate core functionality after library upgrades.
- Implement canary releases or feature flags to gradually roll out upgraded libraries in production environments, allowing for real-world testing with minimal risk.

- Consider maintaining a separate test environment that mirrors production but is always running the latest library versions. This can help identify potential issues before they affect the main development or production environments.

CHAPTER 4
SUMMARY AND ACTION-BASED INSIGHTS

SUMMARY:

Managing vulnerable libraries in enterprise codebases is a complex but critical aspect of maintaining secure applications. The key is to balance security needs with operational realities, employing a risk-based approach that prioritizes the most critical vulnerabilities while maintaining system stability.

ACTION-BASED INSIGHTS:

1. **Implement Now:**

- Establish a vulnerability management program with clear roles and processes.
- Deploy SCA tools integrated into your development pipeline.
- Create and maintain a comprehensive inventory of third-party components.

2. **Plan for Short-Term:**

- Develop a risk assessment framework for evaluating vulnerabilities in your specific context.
- Implement automated testing for library updates to streamline the update process.

- Provide security training to development teams, focusing on dependency management.

3. **Long-Term Strategies:**

- Develop a library deprecation strategy to gradually phase out problematic dependencies.
- Invest in building internal libraries for core functionalities to reduce external dependencies.
- Establish relationships with key vendors to ensure timely security updates.

4. **Continuous Improvement:**

- Regularly review and refine your vulnerability management processes.
- Stay informed about emerging threats and evolving best practices.
- Foster a security-conscious culture across the organization.

5. **Resource Optimization:**

- Implement automation tools to streamline the upgrade process and reduce manual effort.
- Develop a strong business case for allocating resources to library management.
- Explore gradual and partial upgrade strategies to work within existing constraints.

6. **Cultural Shift:**

- Foster a culture that values ongoing maintenance and security alongside feature development.
- Encourage cross-functional collaboration to tackle library management challenges.
- Implement a library governance model to prevent accumulation of technical debt.

7. **Smart Prioritization:**

- Implement a structured prioritization process for dependency upgrades based on business value, exposure, and risk assessment.
- Use reachability analysis to focus efforts on dependencies that pose the greatest actual risk to your applications.

8. **Dependency Hygiene:**

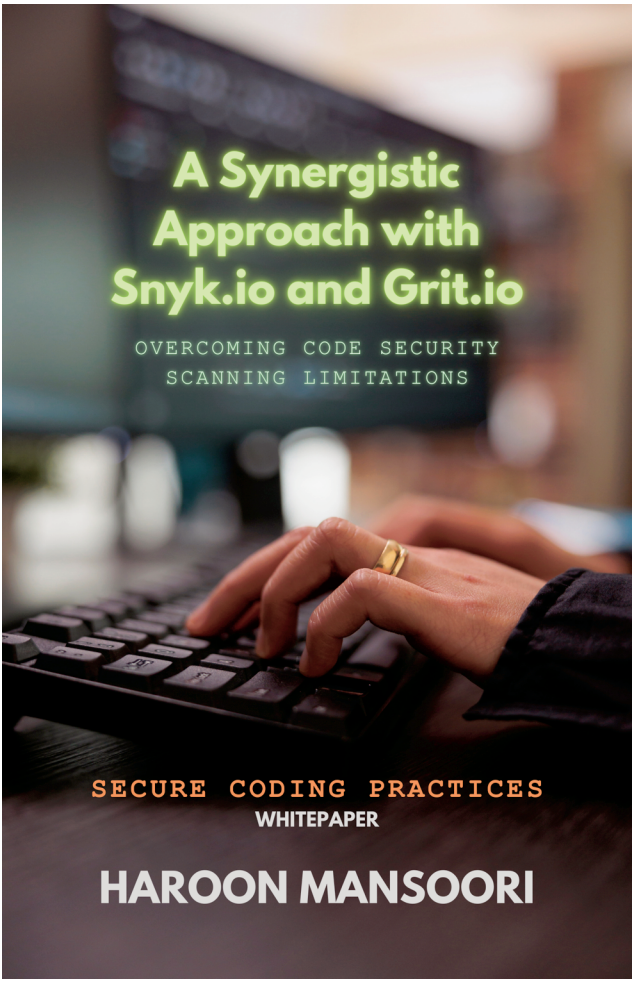
- Regularly audit and clean up unused or obscure dependencies to reduce the overall attack surface and simplify future upgrades.
- Develop a process for vetting and approving new dependencies before they're introduced into your projects.

Organizations can greatly enhance their capacity to handle and reduce risks linked to third-party libraries by adopting these approaches, leading to more secure and resilient applications over time. Recognizing the practical challenges posed by limited resources, competing priorities, and the sheer volume of dependencies, organizations can develop more realistic and effective strategies for managing vulnerable libraries. The key is finding a middle ground between optimal security measures and realistic methods that function within current limitations while progressively advancing toward a stronger and more sustainable library management framework.

By integrating these advanced strategies for prioritizing, analyzing, and managing dependencies, organizations can more effectively navigate the complex landscape of library vulnerabilities. This approach empowers teams to concentrate their scarce resources on the most pressing concerns, aligning security requirements with operational realities and business priorities.

ALSO BY THE AUTHOR

Check out my other white paper “A Synergistic Approach with Snyk.io and Grit.io” that helps Overcoming Code Security Scanning Limitations of tools such as Snyk.



ABOUT THE AUTHOR

Haroon Mansoori is a Cybersecurity Subject Matter Expert with over 27 years of IT experience and more than 31 years, serving clients and end-customers across North America, Europe, the Middle East, and Asia-Pacific regions from several sectors of industries. He has demonstrated leadership through senior and middle management roles, as well as Center-of-Excellence (COE) ownership positions. His proven ability extends to leading and managing IT practices and Cybersecurity teams of over 130 personnel, while also excelling in specialist SME consulting roles.

Currently based in Canada, he is an active contributor with global experience, including several years of previous work in the US and India. Mansoori has successfully built and scaled IT practices from the ground up for multiple employers, generating multi-million dollar revenue. In addition, he has also led technical and functional departments with several millions of dollars in annual budgets for his previous employers. He has conducted and supervised hundreds of workshops to private corporate groups across various topics.

As a trusted technology advisor to over a dozen Fortune 500 clients, including SAP, FedEx, and Humana, Mansoori has delivered specialist consulting expertise and established IT programs involving hundreds of engineers. His deep cybersecurity expertise is evidenced by multiple certifications and memberships in prestigious organizations such as OWASP (Lifetime), ISACA (Gold Professional), and OCEG (Professional). Additionally, Mansoori is a certified auditor, possesses expertise in GRC and

policy management, and is passionate about coaching in security, privacy, and secure SDLC.

Currently pursuing a Doctorate in Leadership and Management, Mansoori is also a certified NLP and REPT/EFT Tapping expert. He has authored several books on leadership and maintains blogs focused on philosophical mindset changes and positive thinking. These resources are particularly valuable for individuals facing disorders, addressing what's most important for people - the rest of their lives.

 linkedin.com/in/hmansoori

This page has been intentionally left blank.