World Journal of Advanced Engineering Technology and Sciences

eISSN: 2582-8266

Cross Ref DOI: 10.30574/wjaets

Journal homepage: https://wjaets.com/

(REVIEW ARTICLE)

Check for updates

Secure-by-design principles in Agile SDLC: Leveraging Formal Verification and AI-Enhanced Code Review in CI/CD Environments

Tim Abdiukov *

NTS Netzwerk Telekom Service AG, Australia.

World Journal of Advanced Engineering Technology and Sciences, 2023, 09(01), 494-503

Publication history: Received on 09 April 2023; revised on 19 May 2023; accepted on 27 May 2023

Article DOI: https://doi.org/10.30574/wjaets.2023.9.1.0149

Abstract

Conducting security in the Agile Software Development Life Cycle (SDLC) is becoming increasingly important in the current digitalized world, where protecting the system is paramount to avoid significant losses through costly security breaches. This paper presents research that examines the use of secure-by-design principles in Agile frameworks and how formal verification approaches and AI-enabled code review can enhance security within Continuous Integration and Continuous Deployment (CI/CD) pipelines. Based on research conducted between 2000 and 2022, the paper focuses on integrating security into development at an early stage, encompassing DevOps and DevSecOps. The results indicate that formal techniques give mathematical correctness of software, whereas AI tools augment code inspection by automatically identifying flaws and implementing coding conventions. Additionally, the article outlines the tangible benefits of implementing these methods, supported by case studies, best practices, and industry benchmarks, including compliance rates, cost savings on rework, and response times. Future trends, despite minimal changes in user behavior, tool integration, and cultural resistance, are likely to persist in innovation in security automation. Therefore, security-by-design is not only possible but is quickly becoming inevitable in contemporary Agile.

Keywords: Secure-by-Design; Agile SDLC; CI/CD; Formal Verification; AI Code Review; DevSecOps; Threat Modeling; Static Analysis; Software Security; Secure Development Lifecycle

1. Introduction

In an era where rapid software delivery is critical, integrating security into the Agile Software Development Life Cycle (SDLC) has become both a necessity and a challenge. Traditional post-development security checks are no longer sufficient in modern CI/CD-driven environments. This paper explores a Secure-by-Design approach that embeds security early and continuously throughout the Agile SDLC. By combining formal verification techniques with AI-enhanced code review tools, we demonstrate how development teams can proactively identify and mitigate vulnerabilities during development, ensuring resilient and trustworthy software systems from the ground up. Secure Software Development in Agile SDLC

1.1 Overview of Agile Software Development Life Cycle (SDLC)

Agile software development refers to an iterative and incremental software development methodology that focuses on customer cooperation, adaptability, and rapid release. Projects are scheduled in brief periods of work or working teams (sprints or iterations) with constant interaction between the development of new features and testing. In this model, work is done in phases or increments, and planning is made on a just-in-time basis. Teams can respond to changes fast. Switch quickly to agile methods, which were implemented as a promising means of keeping up with faster-paced innovation and quickly evolving needs, as noted by Hohl et al. The idea behind Kanban is to encourage fast responsiveness, specifically in response to changing customer demands.

* Corresponding author: Tim Abdiukov.

In contrast to inflexible phase-gated processes, Agile practitioners prefer cross-functional teams that self-organize and add small increments to the product continuously and steadily. For example, core Agile activities include test-driven development (TDD) and sprint-based development, whereas integration routines, such as Continuous Integration (CI), are considered crucially important. Indeed, industry best practices suggest that CI, where developers often merge and construct code, has become one of the most effective practices in SDLC Agile, assisting in detecting integration errors early. (Larry Conklin and Gary Robinson, 2017)

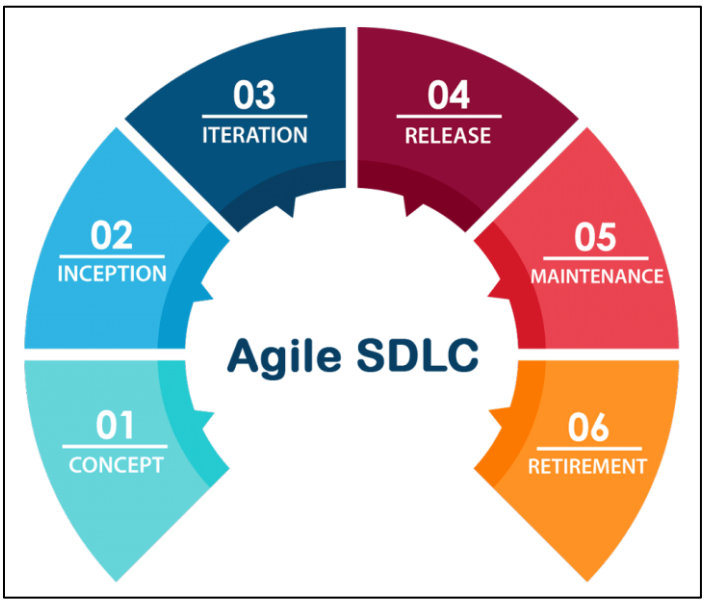

Figure 1 Agile SDLC (Software Development Life Cycle)

The Agile methodology is typically built on the foundations of the Agile Manifesto (2001) and encompasses frameworks and features such as Scrum and Kanban. These stresses are on regular, functional software delivery and constant testing with stakeholders. Teams conduct brief daily meetings and sprint review meetings to check and adjust the product. The result is a software that grows according to the user feedback and changing market requirements. Atlassian describes agile as a means of planning, executing, and evaluating flexibly, with the result being quicker and more adaptive. Another important principle is that of just enough upfront design, wherein teams release small chunks of functionality, seek user feedback, and incorporate it into the subsequent cycle (https://www.atlassian.com/agile). Practically, having fully documented designs and long-term planning is discouraged and should be replaced with the creation of functional features that can be easily modified and updated. Agility has been known to offer a higher quality and faster time-to-market than traditional waterfall methods. (https://owaspsamm.org/, Hohl et al 2018).

1.2 Importance of Security in Agile Practices

Software is highly ubiquitous and mission-critical today, and thus, sensitive security flaws may lead to catastrophic consequences. Organizations typically incur massive expenses due to security incidents and breaches. As one study remarks, the cost of implementing and supporting software throughout its lifetime is highly dependent on security incidents and threats (Rindell et al., 2021). Neglect of security can increase the costs of development and maintenance effortlessly. Ironically, the prioritizing of speed and leanness promoted by Agile may be antagonistic to careful security engineering. It is commonly reported by practitioners that agile methodologies conflict with traditional security procedures (Rindell et al, 2021). According to OWASP guidance, it is challenging to build security into an Agile SDLC process because we constantly need the participation of security professionals or security-centric developers at every team (Larry Conklin & Gary Robinson, 2017). Ironically, the prioritizing of speed and leanness promoted by Agile may be antagonistic to careful security engineering. It is commonly reported by practitioners that agile methodologies conflict with traditional security procedures (Rindell et al, 2021). According to OWASP guidance, it is challenging to build security into an Agile SDLC process because we constantly need the participation of security professionals or security-centric developers at every team (Larry Conklin & Gary Robinson, 2017). For instance, a 2021 survey of practitioners found that Agile teams are most likely to conduct security tasks during requirements and implementation, with their initial-stage activities being deemed the most influential (Rindell et al., 2021). That is, it is much more effective to mitigate risk through proactive practices, such as threat modeling and secure design, than through reactive solutions.

Therefore, a concept of a security being on the left side (earlier stages) of the time scale, where the left shift is typically promoted in an Agile context.

1.3 Purpose of Integrating Secure-by-Design Principles

The role of secure-by-design is to incorporate security into the software at the earliest stage, rather than adding it as a bolt-on. In practice, this involves collecting security requirements in addition to functional requirements, conducting threat analysis at the design stage, and composing safety measures into the code and design. Secure-by-design is compatible with such notions as a Secure SDLC or DevSecOps. For example, Check Point Software has defined Secure SDLC as a process that integrates security into the development process to ensure security testing is performed in parallel and risk analysis is implemented as early as possible. Through adherence to these principles, some teams should be able to release products that are robust by default (e.g., secure device settings, authenticity, logging) and contain extremely few exploitable vulnerabilities. The inclusion of secure-by-design will minimise vulnerability and its effects. To use the words of one industry guide, Secure SDLC is significant, in short, because the security and integrity of software matter. It allows reducing the likelihood of security vulnerabilities in your software products in production. Of reducing the effects on vulnerabilities in case an attack is discovered". That is, early consideration of security ensures that most faults are identified and avoided during release. And this makes roles more transparent too: secure-by-design offers us "a common structure to ascertain roles, enhancing visibility and the quality of planning and tracking and minimizing risk". This state of mind allows continuous assurance in an Agile CI/CD pipeline, where all build checks and code reviews run automatically and automatically check security issues. The ability to save costs is another important advantage. Applying security controls (including static analysis, automated verification techniques, or formal specifications) early (on the first day of development) does not require costly rework and hotfixes. As an example, Check Point observes that early detection of vulnerabilities (and inserting fixes at the same time development is happening) results in the end of patching after deployment, which brings tremendous savings. Moreover, in-built security helps simplify the process of adhering to regulations and norms (e.g., data protection by design rules, industry security paradigms). Overall, the employment of secure-by-design principles in Agile processes implies that rapid development will not compromise safety: Agile will deliver greater trust and quality at a lower long-term cost.

2. Understanding Secure-by-Design Principles

2.1 Definition and Key Concepts

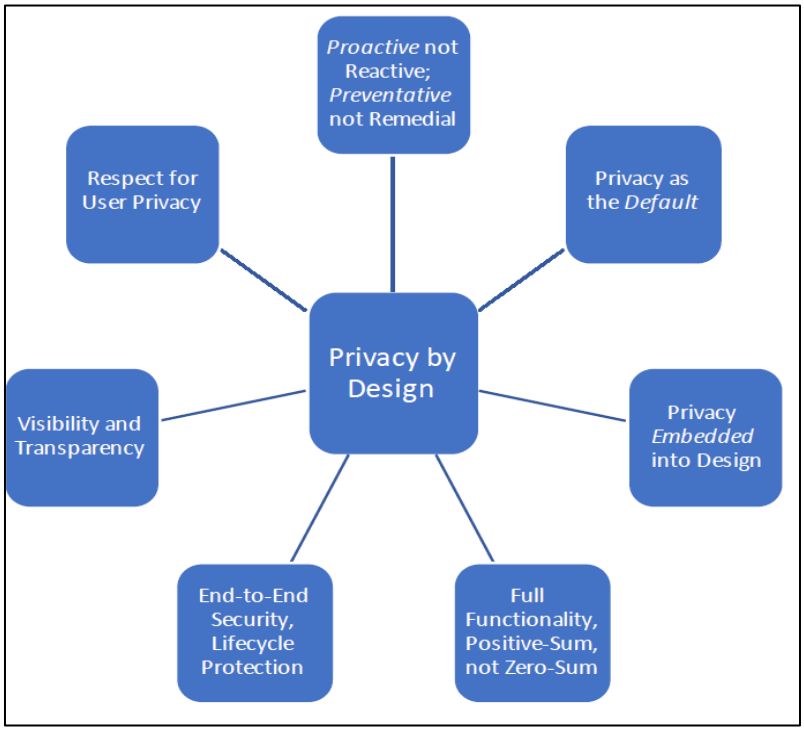

Figure 2 Secure-by-Design Principles

Secure-by-design is the process of actively applying security principles and controls throughout all phases of software development, ensuring that security is not regarded as an add-on, but rather as a mandatory functional requirement. This idea was developed to counter the rising software vulnerability resulting from reactive or bolt-on security measures. Howard and Lipner (2006) define the concept of secure-by-design, stating that it requires developers to think like attackers and discover security vulnerabilities early in the lifecycle. It complies with the principles of the smallest privilege, defense in depth, fail-safe defaults, and secure defaults (Saltzer & Schroeder, 1975), which lead to the architecture and implementation of resilient and robust software.

The principle is similar to the suggestions outlined in ISO/IEC 27034, a guideline for implementing application security during software development. The concept of Secure-by-Design is closely associated with Secure Software Development Lifecycle (SSDLC) models, where activities such as threat modeling, secure coding, and penetration testing are integrated into the iterative cycles (McGraw, 2006; Microsoft, 2012). This is critical, especially in an Agile environment, where constant changes require security to be flexible, automated, and built into rapid development.

2.2 Benefits of Adopting Secure-by-Design Approaches

The technical and economic benefits of adopting secure-by-design principles are good. At a technical level, it makes deployed software less vulnerable and reduces the number of vulnerabilities. Other research suggests that flaws detected during the design phase are much less costly to resolve than those identified after release (NIST, 2002). Moreover, secure-by-design solutions help achieve compliance with legal and regulatory frameworks, such as the General Data Protection Regulation (GDPR), which requires the use of data protection by design and by default (Voigt & Von dem Bussche, 2017). The other major advantage is increased customer confidence. As these hang-ups undermine brand loyalty, particularly in times of data breaches, companies with a high-level security stance take the lead. Furthermore, secure design methods decrease incident response time by reducing the complexity of auditing and forensics, thanks to enhanced logging, traceability, and modularity of the system (Schneier, 2000). Increased code resilience and maintainability are also a result of secure defaults and threat mitigation at the earliest stage.

2.3 Common Secure-by-Design Practices in Software Development

General secure-by-design measures are threat modeling to analyze probable harms, attack paths, and vulnerabilities during the design process (Shostack, 2014); collecting the security requirements by including non-functional security requirements along with functional ones; application of secure code standards such as OWASP Secure Coding Guidelines or SEI CERT Coding Standards; validation and sanitization of user inputs to avert blocks and masses; implementation of role-based access controls (RBAC) to present the strict criteria on access, and the law of least privilege; application of automated tools, similar To remain relevant, these practices have to constantly be revisited during Agile processes, to keep pace with the application and the threats it faces.

3. The Role of Formal Verification

3.1 Overview of Formal Verification in Software Engineering

Formal verification is defined as the process of proving a software system correct concerning its specifications and is performed mathematically. It employs airtight logics and formal techniques — including rigorous model checking, formal verifications, theorem proving, and formal semantics — so that a piece of software functions as desired in all circumstances (Clarke, Grumberg, & Peled, 2000). Unlike testing, which tests individual execution paths, formal verification aims to analyse exhaustively all program behaviours.

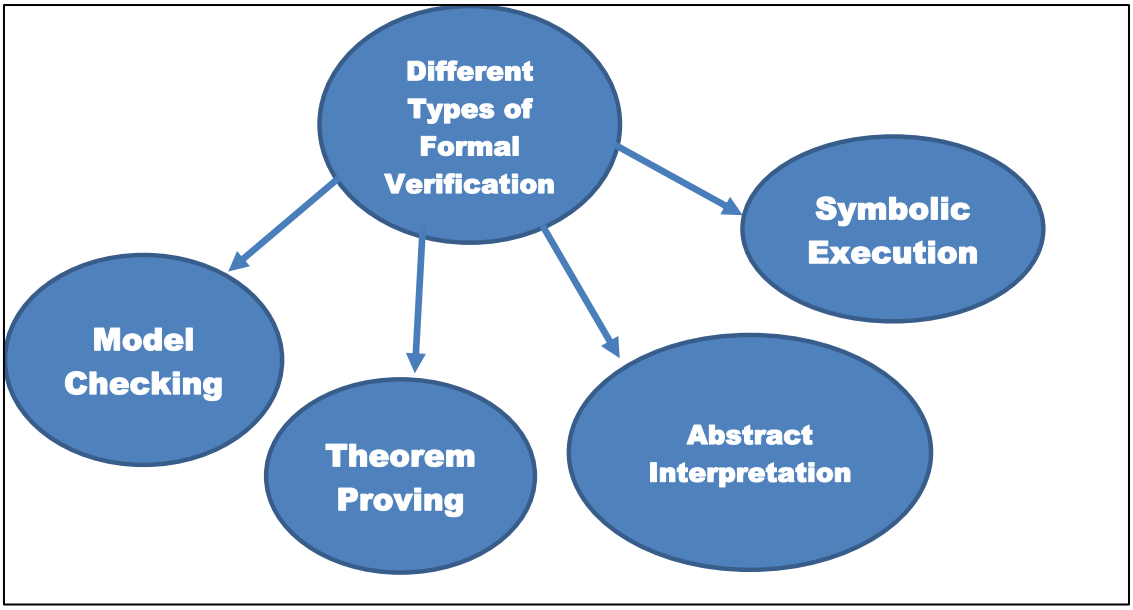

Figure 3 Different Types of Formal Verification

Formal methods have long been used in safety-critical applications, such as avionics, defense, and cryptography (Rushby, 1993). However, they are finding more frequent application in conventional software engineering as a means towards secure-by-design approaches. In its ever-faster progression, the entry barrier has now been reduced to the point where it is possible to use formal verification even in an iterative development process, adhering to the formality of tools such as SPIN, Coq, TLA+, and Boogie, created by Microsoft.

3.2 Techniques and Methodologies for Formal Verification

There are several prominent techniques:

- Model Checking: A system automatically verifies that a model of a system satisfies a prescribed property (e.g., in temporal logic). SPIN and NuSMV are some of the available tools.
- Theorem Proving involves applying logical proofs to determine whether the system's behavior can be demonstrated to follow its specification. Such examples are Coq, Isabelle, and HOL4.
- Abstract Interpretation: A method over-approximating program behaviour to reason about properties such as safety and termination.
- Symbolic Execution: Traces computer program paths with symbolic, rather than concrete, inputs, searching for logic errors and security violations.

Formal methods are often applied in Agile situations to verify critical subunits, such as authentication modules or cryptographic routines, but not the entire systems themselves.

3.3 Advantages of Using Formal Verification for Security Assurance

Assurance is the most important advantage of formal verification in security: it is the user of a mathematically verified system that is provided with guarantees that cannot be achieved by testing alone. Suppose safety-related properties, such as memory safety, invariants of access control, or the correctness of certain protocols, are specified appropriately. In that case, it can rule out classes of vulnerabilities (e.g., buffer overflows, race conditions) (Abadi & Blanchet, 2005). Additionally, formal verification can facilitate the development of satisfying frameworks, such as DO-178C (for avionics) or Common Criteria (for government systems), which frequently require formal assurance levels. For instance, the formal methods employed by Microsoft during the verification of parts in the Hyper-V hypervisor significantly reduced security vulnerabilities in the core infrastructure (Hunt & Suh, 2010). Resource-intensive as it is, formal verification is more feasible when applied to high-risk modules or as part of CI/CD pipelines, particularly for vital functionality. The selective and incremental use enables Agile teams to experience the advantages of mathematical rigor without sacrificing velocity.

4. AI-Enhanced Code Review

4.1 Introduction to AI in Code Review Processes

Code review is a crucial quality assurance process in which developers scrutinize code to identify flaws, ensure ease of maintainability, and verify compliance with standards. Historically, this has been done using manual peer reviews, which are unreliable, time-consuming, and prone to oversight. With increasing codebases and rapid development through continuous integration/deployment (CI/CD), manual reviews can quickly fall behind. Artificial intelligence (AI) has been leveraged more recently to automate and improve code review processes, thereby overcoming these limitations. AI-based code review tools utilize machine learning (ML), natural language processing (NLP), and deep learning techniques to scan code, identify patterns in large repositories, and detect potential bugs, security vulnerabilities, or stylistic issues. Labeled datasets, such as historical commit messages, bug fix records, and static analysis reports, may be used to train these systems to predict problem code areas or suggest improvements. It is worth noting that AI code review is also consistent with Agile and DevSecOps, in which quick iteration and automation play a central role in ensuring secure and reliable development occurs (Ray et al., 2016).

4.2 Benefits of AI-Enhanced Code Review

When combined with AI, code review becomes extremely efficient and high-quality as AI makes the approach scalable, extremely fast, as the code can be analyzed in real-time and issues can be detected almost instantly after a commit, thus shortening the time developers spend waiting and the Agile iteration cycle (Tufano et al., 2019). It offers better consistency because AI models deploy standardized sets of criteria, rather than human nodes, which are subject to inconsistencies, and systematically identify popular security and performance anti-patterns (Pradel & Sen, 2018). These tools can also provide early-stage bug and vulnerability detection to identify weaknesses that are typically difficult to manually detect, such as SQL injection, buffer overflow, or any hardcoded secrets, with training on Common Weakness Enumeration (CWE) data. AI reviewers enhance secure coding practices by adhering to OWASP and CERT standards, suggesting secure variants of constructs and reporting insecure ones. Moreover, they can continuously learn new vulnerabilities and evolving coding patterns, which makes them relevant in the long-term perspective (Ray et al., 2016). The combination of these features minimizes the rework required, enhances the effectiveness of developers, and reduces vulnerabilities in production, without eliminating human reviewers; instead, it complements them by revealing critical problems and dismissing trivial issues.

4.3 Tools and Technologies for AI-Driven Reviews

To assist in automated code review, several tools have been built on AI to provide enhanced capabilities for real-time analysis and security vulnerability detection. DeepCode (acquired by Snyk) utilizes machine learning and symbolic reasoning, and is trained on millions of open-source commits (Sadowski et al., 2015) to identify bugs and security defects. CodeGuru by Amazon will also rely on ML models, trained on massive internal codebases, to offer review suggestions and detect performance or security defects. Facebook Infer uses static analysis methods, including symbolic execution and abstract interpretation, to find problems such as null pointer dereferences, race conditions, and resource leaks. It can easily be plugged into your CI pipelines, allowing you to receive feedback promptly. Modern transformer-based models, such as CodeBERT and GraphCodeBERT, which have been trained on code and comments, have been shown to be highly effective in defect prediction and vulnerability classification (Feng et al., 2020). In DeepBugs, NLP and neural networks are used to detect semantic bugs and comprehend the semantic use of identifiers and functions (Pradel & Sen, 2018). These tools are typically part of a Git workflow, used as build automation, pull request bots, or issue trackers in CI/CD environments, ensuring that security checks run on every commit or merge and provide continuous security assurance.

5. Integrating Secure-by-Design Principles in CI/CD Environments

5.1 Overview of Continuous Integration/Continuous Deployment (CI/CD)

Continuous Integration and Continuous Deployment (CI/CD) is a crucial aspect of contemporary Agile software development, enabling teams to deliver frequent updates in a fast and reliable manner. CI is a practice of automatically creating and testing code whenever a team member makes a change to version control, to identify defects early in the process. CD is founded on this by automating the deployment into production or staging environment, encouraging quick time-to-market and higher-frequency releases (Fowler, 2006; Humble & Farley, 2010). Automation, version control, and testing are key principles of the CI/CD paradigm, which can be translated into an Agile iterative approach

that makes it relatively easy to introduce a security process into the development cycle and implement security assurance.

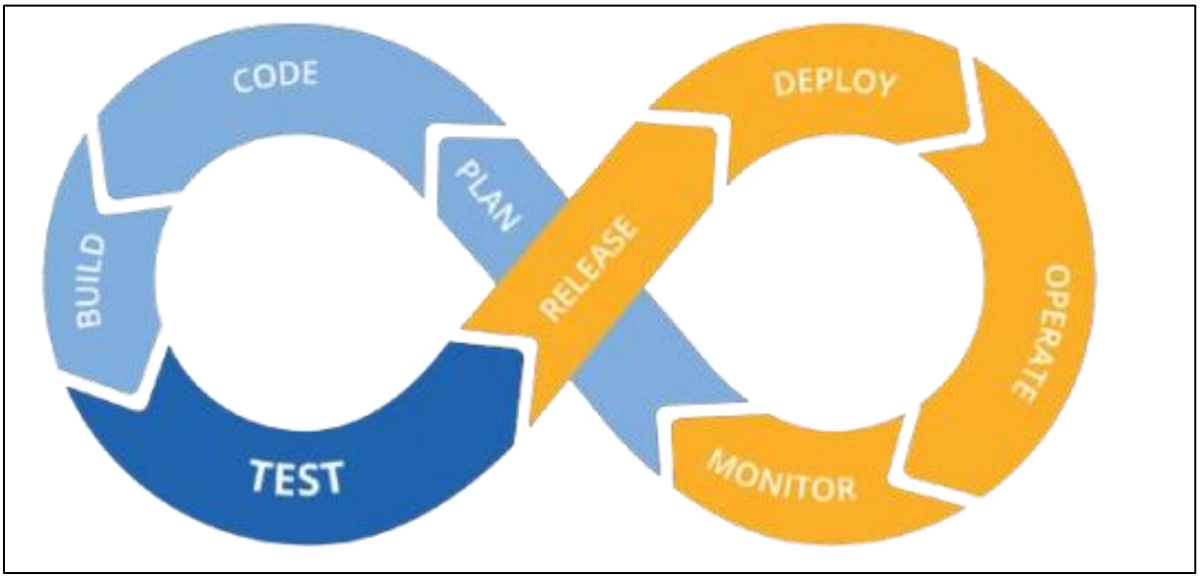

Figure 4 Continuous Integration/Continuous Deployment (CI/CD)

5.2 Steps for Integrating Security Within CI/CD Pipelines

Organizations must be strategic and layered as they integrate secure-by-design into their CI/CD pipelines. First, threat modeling should be conducted during the planning phase to identify potential attack vectors and architectural vulnerabilities earlier (Shostack, 2014). Second, applying Acardia and Doordia (static and dynamic application security testing) tools to the CI pipeline ensures that known weaknesses and insecure patterns are verified in the code during the development and testing stages (Chess & West, 2007). The inclusion of Software Composition Analysis (SCA) facilitates the identification of vulnerabilities in third-party dependencies, which is one of the most vulnerable attack surfaces of modern software systems (Kula et al., 2015). The third important step is to apply secure coding standards by adding linters and security-based code review processes to pull requests and creating CI builds (McGraw, 2006). Fourth, container and infrastructure-as-code (IaC) scanning tools are necessary to identify misconfigurations and ensure that the deployment environment is secure. Lastly, CI/CD tooling and infrastructure must also be hardened, with the least privilege access, managing secrets using encryption, and audit logs all applied to secure the pipeline as the means of software delivery (NIST, 2015).

5.3 Best Practices for Maintaining Security Throughout the Development Process

Several best practices for maintaining security during code development and deployment in a CI/CD environment have been established. To begin with, taking a shift-left path or leading security testing and review earlier in the SDLC ensures that vulnerabilities are detected at the time when they are the least costly and easiest to repair. Incorporating security champions into Agile teams fosters a culture of security awareness, ensuring that security issues are not sidelined due to delivery pressure (Xia et al., 2019). Continuous developer training on secure coding practices further reinforces this mindset. It is also beneficial to incorporate automated feedback on security test results, allowing developers to be aware of problems in real-time and enabling prompt rectification and risk minimization. Policy-as-code tools enable the automation of governance, compliance rules, and security configurations across environments. Lastly, it may be beneficial to assess the quality of safe practices by measuring the time to detect (MTTD) and time to remediate (MTTR) security problems, allowing the team to constantly refine its security posture (Forrester, 2018) constantly. Effective adoption of secure-by-design considerations in CI/CD ecosystems is a long-term goal that necessitates a whole-of-life methodology to combine technical controls, team processes, and organizational culture to protect software systems against an ever-evolving threat landscape.

6. Case Studies and Practical Applications

The practical implementation of secure-by-design design principles into the CI/CD pipeline can be best explained by providing real-world examples of both successes and challenges. An example is the Security Development Lifecycle

(SDL) by Microsoft, which has played a significant role in incorporating security aspects throughout the software development process, allowing the company to minimize vulnerabilities in its products via threat modeling, secure code standards, and the integration of automated testing. In a similar vein, the fact that Google implemented BeyondCorp, a cloud-native security framework that expands zero-trust principles to application deployment, underscores the efficiency and tight integration of security in containerized environments. What these cases can teach us is the importance of threat modeling early on, ongoing training for developers to promote secure coding, and how automation can minimize human error. Moreover, examples of companies such as Capital One and Netflix demonstrate that incorporating tools like SonarQube, Checkmarx, and bespoke DAST pipelines into their CI/CD processes significantly enhances the speed of defect detection and remediation. Based on these applications, best practices have emerged, including the injection of security checkpoints into the version control pipeline, the use of immutable infrastructure, and red team/blue team training. In order to assess the success of these insertions, engineers have resorted to metrics that include time to remediate vulnerabilities, number of security vulnerabilities per release, mean time to detect (MTTD), and mean time to respond (MTTR), which provide quantitative measures that reflect on the maturity and reactivity of secure-by-design pipelines.

7. Challenges and Limitations

Despite increased use of automated security testing capabilities in the DevSecOps pipeline, several issues remain that can limit its functionality. One of the considered technical difficulties is the connection of the different tools within the CI/CD lifecycle, as many SAST, DAST, and IAST tools are executed in an isolated manner. This contributes to fragmented and incoherent testing environments and outcomes. Moreover, the fact that false positives are numerous and that many tools lack contextual intelligence tends to overwhelm developers and security teams, making them less efficient. Scalability is also an issue, especially when testing large codebases or microservices, as performance overheads can increase deployment cycles.

Additionally, automated tools may struggle to identify logic errors, business rule violations, or bugs in obfuscated or dynamically generated code. Although regulations and compliance are improving, a significant amount of manual intervention is still required to achieve industry-specific compliance and regulatory validation. The existing professional skill base, which hinders understanding and action, is a major problem. Finally, the exercise of incorporating automated security testing without compromising the pace of agile development is a balancing act, where constant coordination among DevOps, security, and compliance teams is inevitable.

8. Future Trends in Secure Software Development

The combination of artificial intelligence, increased automation, and innovation driven by regulation is shaping the future of secure software development. Machine learning and AI will facilitate the discovery of vulnerabilities more accurately, conduct dynamic risk evaluation, and enable predictive threat detection in real-time, allowing for more proactive and less reactive security operations through complex codebases. DevSecOps pipelines will gradually evolve to incorporate a continuous compliance validation mechanism, a policy-as-code practice, and enhanced developer feedback loops. Developer-oriented tools, such as security-aware IDEs and integrated threat profiling aids, will enable teams to identify and fix vulnerabilities independently and at an earlier stage. Security-as-Code methods will emerge, and Zero-trust principles will gain popularity, particularly when applied to the security process on cloud-native and microservices platforms. Technologies that preserve privacy, such as homomorphic encryption and federated learning, will also gain mainstream status, ensuring that data is not compromised during processing. Additionally, democratization of access to state-of-the-art practices, facilitated by the convergence of collaborative security ecosystems and threat intelligence sharing, will lead to more agile, scalable, and focused secure development, aligning it with the community's priorities.

9. Conclusion

The secure-by-design principle is a core approach to reducing risks to the security of Agile software development, particularly when it is built into the CI/CD process. As demonstrated in this article, organizations can proactively implement security in an SDLC through formal verification methods and AI-aided code review, potentially gaining greater assurance of software quality and increased resistance to evolving threats. Formal methods enable the verification of essential system components concerning stringent specifications, whereas tools with AI assist in vulnerability detection and enhance uniformity in code verification. The feasibility and effectiveness of these strategies are demonstrated through actual applications by tech giants such as Microsoft, Google, and Capital One. The issues associated with tooling complexity, the inability to scale, and developer resistance nonetheless remain some of the

obstacles to widespread popularity. To tackle them, security should be viewed as a joint responsibility and supported by cultural transformations, ongoing education, and automated guardrails. In the future, secure software development can be reinvented with new machine learning tools, automated compliance, and zero-trust systems. Companies that make investments today to build secure-by-design concepts into their operations find themselves in a stronger position to address the twin challenges of speed and safety in the new digital landscape.

References

[1] Abran, A., Bourque, P., Dupuis, R., & Moore, J. W. (Eds.). (2001). Guide to the Software Engineering Body of Knowledge (SEBoK). IEEE Press..

[2] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... & Thomas, D. (2001). Manifesto for Agile Software Development. Retrieved from https://agilemanifesto.org/

[3] Check Point Software Technologies. (2022). What Is Secure SDLC? Definition, Process, and Phases. Retrieved from https://www.checkpoint.com/cyber-hub/cloud-security/what-is-secure-sdlc/

[4] Larry Conklin & Gary Robinson. (2017). Code Review Guide. Creative Commons (CC) Attribution Free Version at: https://www.owasp.org. https://owasp.org/www-project-code-review-guide/assets/OWASP_Code_Review_Guide_v2.pdf#:~:text=The%20term%20%E2%80%98Continuous%20Integration%E2%80%99%20originated,requires%20developers%20to%20check%20code

[5] Hohl, P., Klünder, J., van Bennekum, A., Lockard, R., Gifford, J., Münch, J., & Schneider, K. (2018). Back to the future: Origins and directions of the "Agile Manifesto" – views of the originators. Journal of Software: Evolution and Process, 30(10), e1952.

[6] Khan, H. U., & Al-Yasiri, A. (2021). Challenges in integrating security with Agile development: An empirical study. Information and Software Technology, 137, 106609.

[7] McGraw, G. (2006). Software security: building security in. Addison-Wesley Professional.

[8] National Institute of Standards and Technology (NIST). (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report 02-3. https://www.nist.gov/publications/economic-impacts-inadequate-infrastructure-software-testing

[9] OWASP Foundation. (2021). OWASP SAMM: Software Assurance Maturity Model. Retrieved from https://owaspsamm.org/

[10] Ståhl, D., & Bosch, J. (2014). Modeling the differences in continuous integration practices across industry software development. Journal of Systems and Software, 87, 48–59. https://doi.org/10.1016/j.jss.2013.08.030

[11] https://www.atlassian.com/agile#:~:text=Agile%20is%20an%20approach%20that,more%20responsive%20and%20successful%20outcomes

[12] Rindell, K., Ruohonen, J., Holvitie, J., Hyrynsalmi, S., & Leppänen, V. (2020). Security in agile software development: A practitioner survey. Information and Software Technology, 131, 106488. https://doi.org/10.1016/j.infsof.2020.106488

[13] Abadi, M., & Blanchet, B. (2005). Computer-Assisted Verification of Security Protocols. IEEE Security & Privacy, 3(1), 18–28.

[14] Clarke, E. M., Grumberg, O., & Peled, D. (2000). Model Checking. MIT Press.

[15] Howard, M., & Lipner, S. (2006). The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software. Microsoft Press.

[16] Hunt, G., & Suh, G. E. (2010). Enabling Secure Systems Through Hardware and Software Support. Communications of the ACM, 53(2), 60–68.

[17] ISO/IEC. (2011). ISO/IEC 27034-1: Information technology — Security techniques — Application security — Part 1: Overview and concepts.

[18] McGraw, G. (2006). Software Security: Building Security In. Addison-Wesley.

[19] Microsoft. (2012). Security Development Lifecycle (SDL) Practices. Retrieved from https://www.microsoft.com/en-us/securityengineering/sdl

[20] NIST. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report 02-3.

[21] Rushby, J. (1993). Formal Methods and the Certification of Critical Systems. SRI International Computer Science Laboratory.

[22] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. Proceedings of the IEEE, 63(9), 1278–1308.

[23] Schneier, B. (2000). Secrets and Lies: Digital Security in a Networked World. Wiley.

[24] Shostack, A. (2014). Threat Modeling: Designing for Security. Wiley.

[25] Voigt, P., & Von dem Bussche, A. (2017). The EU General Data Protection Regulation (GDPR). Springer.

[26] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv preprint arXiv:2002.08155.

[27] Pradel, M., & Sen, K. (2018). DeepBugs: A Learning Approach to Name-based Bug Detection. Proceedings of the ACM on Programming Languages, 2(OOPSLA), 1–25.

[28] Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2016). A Large-scale Study of Programming Languages and Code Quality in GitHub. Communications of the ACM, 60(10), 91–100.

[29] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., & Winter, C. (2015). Tricorder: Building a Program Analysis Ecosystem. Proceedings of the 37th International Conference on Software Engineering - Volume 1, 598–608.

[30] Tufano, M., Watson, C., Bavota, G., Poshyvanyk, D., Di Penta, M., & White, M. (2019). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(4), 1–29.

[31] Chess, B., & West, J. (2007). Secure programming with static analysis. Addison-Wesley.

[32] Fowler, M. (2006). Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html

[33] Forrester (2018). The State of Application Security. Forrester Research, Inc.

[34] Humble, J., Farley, D. Addison-Wesley (2010).. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. https://books.google.com.ng/books?id=9CAxmQEACAAJ

[35] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. (2018). Do developers update their library dependencies? Empirical Softw. Engg. 23, 1 (February 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5McGraw, G. (2006). Software security: Building security in. Addison-Wesley.

[36] NIST. (2015). Security and Privacy Controls for Federal Information Systems and Organizations (2013). https://doi.org/10.6028/nist.sp.800-53r4

[37] Shostack, A. (2014). *Threat modeling: Designing for security*. Wiley.

[38] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan and S. Li, "Measuring Program Comprehension: A Large-Scale Field Study with Professionals," in IEEE Transactions on Software Engineering, vol.. 44, no. 10, pp. 951–976, October 1, 2018, doi: 10.1109/TSE.2017.2734091.

[39] Krishan Kumar (September 25, 2022). Agile Software Development Lifecycle. https://k21academy.com/scrum-master/agile-sdlc/

[40] Silva, Paulo & Monteiro, Edmundo & Simoes, Paulo. (2021). Privacy in the Cloud: A Survey of Existing Solutions and Research Challenges. IEEE Access. PP. 1-1. 10.1109/ACCESS.2021.3049599.

[41] Kelsey Meyer (Published July 12, 2019) A Complete Breakdown of CI/CD: Differences, Benefits, and Tools. https://blog.american-technology.net/continuous-integration-vs-delivery-vs-deployment/