Computers & Security 137 (2024) 103639

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Secure software development and testing: A model-based methodology

Valentina Casola [a], Alessandra De Benedictis [a,*], Carlo Mazzocca [b], Vittorio Orbinato [a]

[a] Department of Electrical Engineering and Information Technologies, University of Naples Federico II, Naples, Italy
[b] Department of Computer Science and Engineering, University of Bologna, Bologna, Italy

ARTICLE INFO

Keywords:
Secure software development
SecDevOps methodology
DevSecOps methodology
Model-based testing
Automated security testing plan generation
Secure containerized microservice applications

ABSTRACT

Modern industries widely rely upon software and IT services, in a context where cybercrime is rapidly spreading in more and more sectors. Unfortunately, despite greater general awareness of security risks and the availability of security tools that can help to cope with those risks, many organizations (especially medium/small-size ones) still lag when it comes to building security into their services. This is mainly due to the limited security skills of common developers/IT project managers and to the typically high costs of security procedures. In fact, while automated tools exist to perform code analysis, vulnerability scanning, or security testing, the manual intervention of security experts is still required not only for security analysis and design, but also to configure and elaborate the output of the security testing tools.

In this paper, we propose a novel secure software development methodology aimed at supporting developers from security design to security testing, suitable for integration within modern DevOps pipelines according to a DevSecOps (or SecDevOps) approach. The proposed methodology leverages a model-based process that enables identifying existing threats, selecting appropriate countermeasures to enforce, and verify their mitigation effectiveness through both static assessment procedures and targeted security tests. To demonstrate our approach's feasibility and concretely illustrate the devised activities, we provide a step-by-step description of the whole process concerning a containerized microservice-based application case study. In addition, we discuss the application of the proposed methodology, in its threat modeling and security testing phases, to a well-known vulnerable web application widely used for security training purposes, to illustrate that we can identify most of the existing vulnerabilities and determine appropriate test plans to assess and mitigate such vulnerabilities.

1. Introduction

In recent years, information security has become increasingly crucial, as demonstrated by the impressive number of security incidents that periodically affect large companies and government agencies[1] with severe consequences in terms of data breaches and service compromise. Most of the time, cyber-attacks exploit vulnerabilities of affected information systems (Tomas et al.), which have not been properly identified and fixed during development. As a matter of fact, the testing and coverage of security requirements is still an open issue and, despite the availability of security guidelines and practices for secure development, they are still too often neglected. This is particularly true in the context of modern development methodologies based on Agile and DevOps paradigms (Leite et al., 2019), which rely upon iterative development processes and a high level of integration and automation to significantly reduce the time-to-market. These methodologies, which are widely adopted nowadays in several contexts (Agrawal and Rawat, 2019; Govil et al., 2020; Waseem and Liang, 2017; Šćekić et al.), did not originally devise the application of security practices during the development cycle, while they typically include security testing activities only at the end of development, with clear issues related to the timeliness in the identification of possible security problems.

To bridge this gap, many companies already practicing DevOps have started to change their approach by explicitly engaging security teams in the development lifecycle, and by embracing the so-called SecDevOps paradigm (Sánchez-Gordón and Colomo-Palacios, 2020), which entails including security activities in all the stages of the DevOps workflow by adopting security best practices related to threat modeling, secure coding, dynamic security testing, vulnerability scanning, security monitoring, etc.

The integration of security engineering practices in a DevOps workflow requires a high level of automation (Haindl and Plösch, 2019).

* Corresponding author.
E-mail address: alessandra.debenedictis@unina.it (A. De Benedictis).

[1] https://www.csis.org/programs/strategic-technologies-program/significant-cyber-incidents.

https://doi.org/10.1016/j.cose.2023.103639
Received 30 August 2023; Received in revised form 15 November 2023; Accepted 3 December 2023
Available online 6 December 2023

Unfortunately, the analysis activities devised during design and development usually need substantial support from human experts and, similarly, security assessment and testing (during the development and after the final deployment) require deep security skills to be performed. A significant level of security automation can be currently achieved today thanks to the multitude of development and testing tools currently available on the market (Kumar and Goyal, 2020). Nevertheless, available technological solutions require broad and advanced technological skills, which are rarely owned by small/medium enterprises. For example, static and dynamic testing tools (SAST and DAST), which are widely employed while performing testing, even in SecDevOps processes, provide automation capabilities. Security experts are still required to handle, for instance, the high number of false positives generated by SAST tools or to write tests, and fine-tune DAST tools. Moreover, existing security testing tools often do not help identify which assets present security issues and which security controls should be improved.

In our previous work (Casola et al., 2020a), we presented a Security SLA-based Security-by-Design Development methodology (SSDE methodology for short) aimed to support developers during the phases of threat modeling, risk analysis, security design, and static security assessment, through semi-automated techniques. The SSDE methodology presents a major limitation, i.e., it is only focused on development (from design to coding), not considering any dynamic security testing activities. Security testing and its automation were the focus of another previous work of ours (Casola et al., 2020b), where we made a first attempt toward proposing a model-based testing approach to integrate within a SecDevOps process. However, that approach did not foresee an automatic security test plan generation, nor did it provide the set of security tests to reproduce.

In this paper, we leverage and extend our previous work by proposing a development methodology meant to support developers from the early security analysis stages to security testing, by means of a model-based process that enables them to partially automate the most critical actions that typically require the intervention of experts. In particular, our methodology relies upon a comprehensive reference security model that includes and correlates information on system assets, threats, countermeasures, vulnerabilities, attacks, and related testing strategies. The model, aimed at codifying security experts' knowledge, is leveraged throughout the development lifecycle to identify existing security risks, determine appropriate security controls to enforce, and build a targeted security testing plan whose feedback can be directly used to identify suitable mitigation strategies to be applied during the next design and coding iterations.

Our contribution can be summarized as follows:

- we present a *secure development methodology* that enables partially automating the security design and security testing activities based on a threat-centric approach by leveraging suitable system and security models. In particular, we discuss a novel model-based *testing strategy* that entails the identification of targeted test plans taking into account the existing security risks and the specific application features, and that exploits popular and widely-used attack pattern catalogues;
- we release a complex *knowledge base*, implementing the security model and leveraging a large set of information derived from standards, scientific papers, best practices, and guidelines related to a wide range of software systems (e.g., IoT, cloud-based applications, containerized services);
- we demonstrate the effectiveness of the proposed methodology by means of a microservice-based application case study, discussed step-by-step. We also discuss the application of the methodology, in its threat modeling and security testing phases, to a well-known vulnerable web application typically used for security training, showing how it is able to identify most of existing vulnerabilities and identify appropriate test plans to exploit and assess such vulnerabilities.

The remainder of this paper is structured as follows. Section 2 provides the technical background from our previous work on secure development, outlines related limitations, and clarifies the novel contribution presented in this paper. Section 3 presents an overview of the methodology stages, focusing on those devoted to security design, while Section 4 presents the details of the model-based security testing stage, which is one of the main contributions of this work. In Section 5, we demonstrate the proposed approach by illustrating its adoption in the design, implementation, and testing of a microservice-based application, and provide some final discussion about the approach and its limitations. Finally, Section 6 provides an overview of the most relevant related work regarding secure development methodologies and security testing techniques with a special focus on model-based testing. To conclude, Section 7 draws our conclusions and presents future work. In order to improve readability, frequently used abbreviations are summarized in Table 1.

Table 1
Acronyms.

Acronym	Full Term
CIA	Confidentiality, Integrity, Availability
CSP	Cloud Service Provider
DAST	Dynamic Application Security Testing
IaaS	Infrastructure-as-a-Service
MACM	Multi-purpose Application Composition Model
SaaS	Software-as-a-Service
SAST	Static Application Security Testing
SLA	Service Level Agreement
SSDE	Security SLA-based Security-by-Design Development
SSDLC	Secure Software Development Life Cycle
VM	Virtual Machine

2. Technical background and contribution

Since the methodology presented in this paper stems from and extends our previous work on secure application development, in this section we provide the required background and highlight current limitations in order to clarify the novelty of the contribution.

As anticipated in the Introduction, to cope with the need for cost-effective solutions to properly address security design and assessment issues in modern development processes, in our previous work (Casola et al., 2020a) we recently proposed the *Security SLA-based Security-by-Design Development (SSDE) methodology*. The methodology, originally focused on the secure development process of cloud-based applications, was aimed at simplifying and automating as much as possible both (i) the identification of existing risks and the selection of appropriate mitigations to improve system design according to security-by-design principles, and (ii) the (post-development) security assessment, aimed at (statically) verifying the actual security capabilities offered by the application, depending not only on the security controls enforced at each component but also on the possible impact of components interconnections and deployment configurations.

As sketched in Fig. 1, the SSDE methodology consists of four main steps, devoted respectively to system *modeling*, *risk analysis*, *per-component security assessment*, and *per-application* security assessment. **System modeling** leverages the MACM formalism introduced in (Rak, 2017), enabling to describe the high-level architecture of an application in terms of a graph, whose nodes represent the main components of the application, belonging to a set of pre-defined types with specific properties, and whose edges represent the interconnections of the components. MACM nodes enable to represent both SW components/modules belonging to the application business logic and infrastructural components used for deployment. In particular, the formalism was originally introduced to model cloud-based applications, hence MACM nodes included Software-as-a-Service (SaaS), Infrastructure-as-a-Service (IaaS), and CPS node types, as well as *provide, host, use* relationships (a simple example is reported in Fig. 2).

In order to automate risk analysis and security assessment tasks, the SSDE methodology relies upon a **reference security data model** (Granata and Rak, 2021; Granata et al., 2022) (depicted in Fig. 3) that

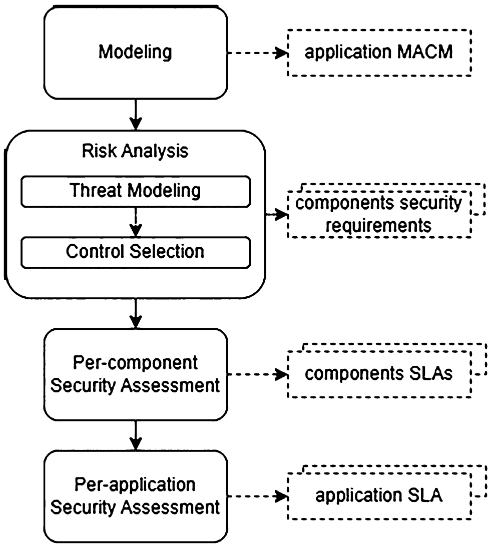
Modeling
application MACM
Risk Analysis
Threat Modeling
Control Selection
components security requirements
Per-component Security Assessment
components SLAs
Per-application Security Assessment
application SLA

Fig. 1. The SSDE methodology (Casola et al., 2020a).

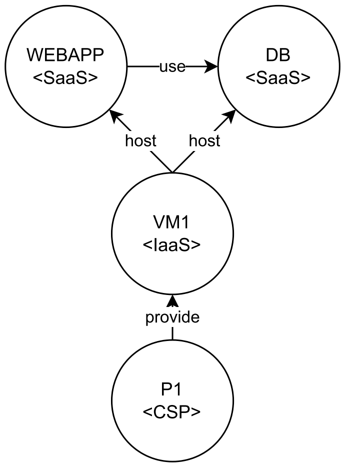

Fig. 2. A simple application modeled with MACM.

correlates *threats* to technical *assets*, intended as the *components* belonging to the application architecture, and to *security controls*, which represent the countermeasures against existing threats. Each asset belongs to a given *component type*, where component types basically correspond to the node types modeled by the MACM formalism. The model is instantiated by a complex knowledge base named *Threat Catalogue*, which collects and correlates well-known threats, assets, security controls, and other relevant security-related concepts. To date, the catalogue includes more than 100 distinct threats related to different assets (e.g., web applications, storage services, cloud-based resources) that have been identified by analyzing relevant scientific literature and/or international projects (e.g., the catalogue includes the current version of the OWASP top 10 Threats for web-based software components, the Cloud Security Alliance top threats for cloud applications, the ENISA threat taxonomy, etc.). All of the reported threats are classified according to the STRIDE taxonomy (Microsoft, 2016). With regard to security controls, the catalogue currently supports the NIST Security Control Framework (National Institute of Standards and Technology, 2013), which counts more than 900 security controls.

Based on the system model, built according to the MACM formalism, and on the Threat Catalogue, the SSDE methodology enables automating the two steps of **risk analysis**, namely *threat modeling* and *control selection*. Threat modeling consists of identifying the threats that affect an application: this step can be completely automated since assets and component types are directly mapped onto applicable threats. Such threats can be further refined by asking developers to reply to a simple automatically generated questionnaire regarding the implementation details of involved components.

The aim of the *control selection* step is to identify the countermeasures to adopt, in terms of *security controls* to enforce, in order to mitigate the existing threats identified during the previous step. Our methodology makes this task quite straightforward, as the Threat Catalogue directly maps threats to the related countermeasures. As said, the catalogue currently supports the controls specified by the NIST Security Control Framework, which refers to both organizational and technical aspects. Since the number of security controls can be huge, our methodology also devises the possibility to refine the controls set based on the results of a risk assessment activity, carried out according to the OWASP Risk Rating Methodology (OWASP Foundation, 2016). This approach evaluates risk based on likelihood and impact levels. In particular, likelihood parameters refer both to specific skills and motivations of threat agents and to aspects related to existing vulnerabilities (e.g., ease of discovery, ease of exploit, awareness, and intrusion detection). Impact parameters take into account both technical factors (loss of confidentiality, loss of integrity, loss of availability, and loss of accountability) and business factors (related to financial damage, reputation damage, non-compliance, and privacy violation). Each of the above parameters is assigned a value in [0;9] and the overall likelihood and impact levels are computed as the average of the values of respective parameters. The final risk is obtained by suitably combining respective likelihood and impact levels based on a predefined match table. While the business factors are dependent on the specific application business model and must be set necessarily by the system owner, the other parameters relate to technical aspects mainly depending on the considered threats and the involved assets. Hence, the assignment of proper values to likelihood and technical impact parameters can be simplified with the introduction of default values, suitably set by security experts based on the knowledge of the specific asset. The results of this risk assessment can be used by developers to prioritize the enforcement of countermeasures for identified threats based on their severity. The interested reader can refer to (Casola et al., 2020a), where the details of these activities are presented. With regard to the subsequent two steps of the SSDE methodology, they are devoted to performing a static **security assessment** both at the component level (via a code review approach leveraging suitable checklists) and at the application level (via a rule-based reasoning approach described in (Rak, 2017)).

2.1. Limitations of the SSDE methodology and contribution

The SSDE methodology was successfully validated in two real-world applications in the context of the MUSA European Project[2] and proved to be effective in increasing the awareness of the involved DevOps teams with respect to security issues while offering good results in terms of efficiency, usability, and time. Despite this, it had a major limitation, i.e., the lack of integration with dynamic security testing techniques.

In our previous work (Casola et al., 2020b), we presented an automated penetration testing approach that was a first attempt towards building automated model-based testing within DevOps. The proposed process, however, lacked the automatic generation of a security test plan for the system under test. In that approach, the list of the collected vulnerabilities and weaknesses was used, as a starting point, to build tests that had to be implemented by the developers. Therefore, there were no direct relationships between threats, attacks, and tests.

To bridge this gap, as discussed in detail in the following sections, in this paper we extended the SSDE methodology introduced above by devising a model-based security testing strategy aimed to generate and execute targeted security tests in a semi-automated way. The security

[2] http://musa-project.eu/.

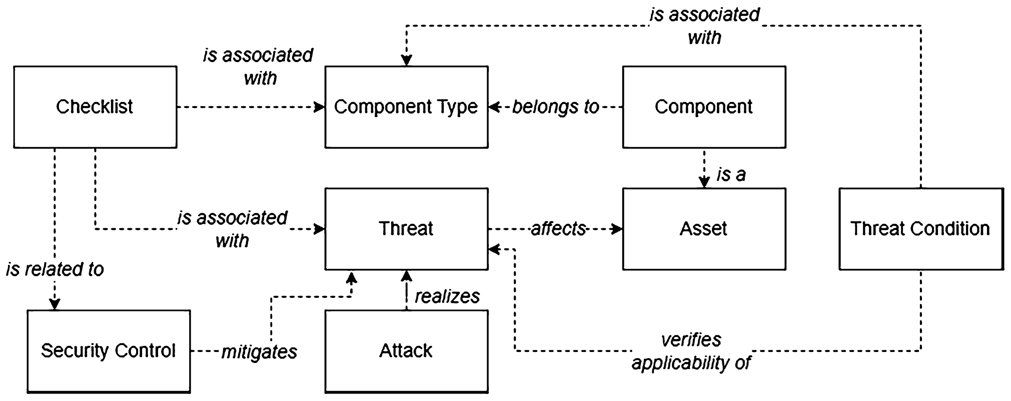

Fig. 3. The reference security model used by the SSDE methodology.

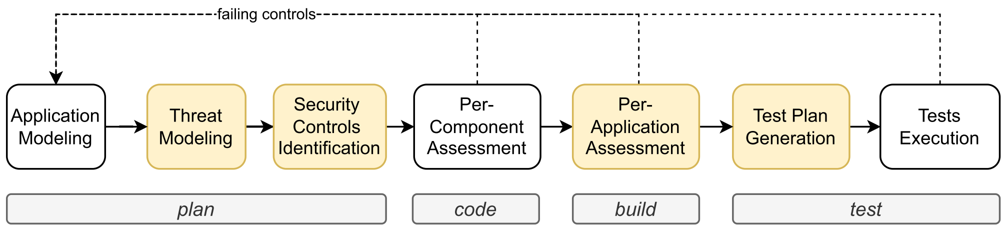

Fig. 4. Main stages of the model-based secure development methodology. Yellow boxes represent activities that can be fully automated, while the remaining steps require a limited and guided intervention from the development teams. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

test plan, as well as the suite of tests to execute in order to exploit identified vulnerabilities and weaknesses, are directly linked with the model of the application under test and with the results of threat modeling. As a further contribution, while the SSDE methodology was originally oriented toward cloud-based applications, we extended the MACM formalism and the Threat Catalogue to allow the modeling and analysis of containerized applications, which are recently becoming very popular and are expected to witness a major boost in the next future (Gartner, Inc., 2020).

3. A model-based secure development methodology: from design to testing

As anticipated, this paper presents a secure development methodology meant to support developers from the early security analysis stages to post-development security testing. The methodology, whose steps are sketched in Fig. 4, leverages automated and semi-automated processes that enable to (i) identify the needed security controls based on an analysis of existing threats, and to (ii) verify whether and how security requirements are fulfilled, based on both static security assessment and security testing activities. The achieved security automation level of the proposed methodology makes it fit within a general SecDevOps process, with particular reference to the development phase, which typically requires most of the security experts' (manual) efforts.

In the remainder of this section, we will discuss each phase in detail, by highlighting the main enhancements proposed with respect to the SSDE methodology summarized in Section 2.

3.1. The planning stage

According to SecDevOps and, more in general, to security-by-design principles, our development methodology takes security into account from the planning stage, by providing support for threat modeling and countermeasure selection. In particular, as discussed regarding the SSDE methodology, we devise automated *threat modeling* and *security controls identification* steps that leverage a suitable model of the application under development, built during *application modeling* by using the MACM formalism, and the Threat catalogue, both introduced in the previous section.

As anticipated, the first extension made to the SSDE methodology was to include support for container-based applications. Containers are a lightweight solution for virtualization which has become very popular in the last few years thanks to their flexibility, portability, scalability, isolation, and security properties. They use Operating System (OS) virtualization, unlike Virtual Machines (VMs), which are based on hardware virtualization. Each VM has got its own OS, kernel, and hypervisor employed to emulate hardware resources such as CPU, memory, I/O, and network devices. On the other hand, a container runs on an OS sharing the host kernel with other containers, enabling it to save resources. Nowadays, containers are widely adopted for agile delivery and microservices-based architectures. The most popular containerization technology is currently represented by Docker,[3] an open-source platform that enables developing and executing applications using containers that run natively on Linux OS. Docker containers are built by (possibly) customizing so-called Docker images, which include everything needed to run an application - the code or binary, runtimes, dependencies, and any other required filesystem object. Containers can be created, started, stopped, moved, or deleted by using the Docker API, which is implemented by a Docker Engine daemon. Due to their popularity, tools responsible for their lifecycle management started to become increasingly used. In particular, orchestration tools (RedHat Inc., 2020) have been introduced for managing container configuration, scheduling, scaling, load balancing, monitoring, and communication. Docker has its own orchestration tool called Docker Swarm,[4] however third-party tools such as Kubernetes[5] have also become very popular to

3 https://www.docker.com/.
4 https://docs.docker.com/engine/swarm/.
5 https://kubernetes.io/.

Table 2
MACM-CA node types. Entries with ** symbol represent extensions to the MACM model.

Node Type	Description
*Container***	Container object
*ContainerRuntime***	Runtime environment and engine to run and manage container objects
*ContainerOrchestrator***	Orchestration platform
CSP	Service Provider that offers infrastructure and software services (possibly cloud-based)
IaaS	Infrastructure resource, i.e., Virtual Machine (possibly cloud-based)
SaaS	Software service/component, either developed ad-hoc or offered directly by an external provider, which needs an infrastructure resource or a container for deployment and execution. Currently, two types of services are supported, namely *web-application* and *storage*

achieve extended orchestration functionalities and to reduce the dependency on the specific container technology.

In order to model the specific components of a containerized application, we extended the MACM formalism and added nodes and relationships specific to such technology. Table 2 reports a summary of the node types devised by the MACM for container-based Applications (MACM-CA) extension, which was introduced to model a typical application built by leveraging the container technology. They enable a developer to easily represent a complex application made of several containerized components, possibly orchestrated by an orchestration platform like Kubernetes. In particular, the `Container` node type enables to represent a container instance (e.g., a Docker container object), the `ContainerRuntime` node type models the runtime that enables containers execution (e.g., Docker Engine daemon), and the `ContainerOrchestrator` node type represents the orchestration platform (e.g., Kubernetes). The other components that belong to the original MACM formalism, enabling to model cloud-based resources and providers, have been introduced in Section 2. In our MACM-CA extension, in particular, SaaS and IaaS nodes are used with a wider scope to model either cloud-based or on-premise VMs and generic software services/components (either offered as-a-service according to the cloud paradigm or developed ad-hoc). Table 3 reports the updated node relationships introduced with the MACM-CA extension.

The MACM formalism' extension with container technology-related nodes required also the extension of the Threat Catalogue, in order to explicitly include and map, onto new component types, relevant threats extracted from recent literature (Sultan et al., 2019) and projects[6].[7]

Fig. 5 shows an example application modeled by means of the MACM-CA formalism. The application is made of one main logic component, represented by the service `S`, which is deployed in a Docker container `CS` executed on the Docker runtime `DockerEngine`. The latter, hosted on a virtual machine `VM2`, is managed through Kubernetes orchestrator `K8S`, which is deployed onto another virtual machine `VM1`. Both `VM1` and `VM2` are provided by the same cloud provider, `CSP1`.

3.2. The coding stage

During the coding stage, developers should verify that the required security controls, identified during the planning stage, have been properly implemented and configured in each component. In our methodology, this is performed by a code review approach through the provisioning of custom *checklists*. This activity corresponds to the *per-component security assessment* step of the original SSDE methodology, which has been introduced in Section 2 and that relies upon well-known assessment questionnaires and best practices suitably codified within the Threat Catalogue. As shown in Fig. 3, checklists are directly mapped to security controls and component types in the reference security model, hence they can be automatically retrieved and presented to developers and the replies from developers can be directly employed to statically assess the current security implementation state of each control for all components. Adopted checklists have been derived from well-known assessment questionnaires and best practices (e.g., the Cloud Security Alliance's CAIQ (Cloud Security Alliance, 2011), the Security Controls definition from NIST SP-800-53, the Code Review Guide by OWASP (OWASP Foundation, 2017a), the Berkeley DB Best Practices,

Table 3
MACM-CA relationships. Entries with ** symbol represent extensions to the MACM model.

Relationship	Description
provides	Links a node of type `CSP` to a node of type `IaaS:Service` or `SaaS:Service`, and models the provisioning of infrastructure resources or software services by a provider
*hosts***	Links a node of type `IaaS:Service` or `Container` to a node of type `SaaS:Service` and models the execution environment for services. Moreover, it can model the relationship between a `IaaS:Service` and a `Container` when containers are hosted by VMs
use	Links two nodes of type `SaaS:Service` and models a generic dependency relationship among services
*manages***	Links a node of type `ContainerOrchestrator` to a node of type `ContainerRuntime`
*runs***	Links a node of type `ContainerRuntime` to a node of type `Container`

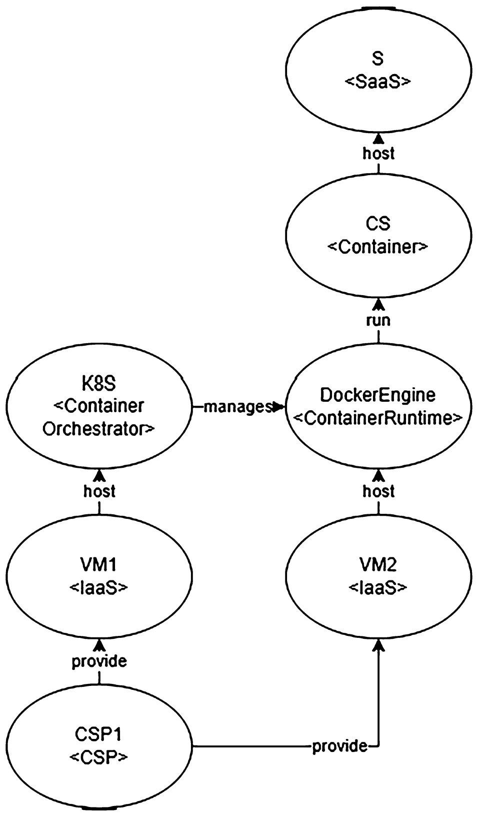

Fig. 5. An example of container-based application modeled with the MACM-CA formalism.

[6] Robail Yasrab, 2018. Mitigating Docker Security Issues. https://arxiv.org/abs/1804.05039.
[7] OWASP, Top 10 Kubernetes Risks - 2022. https://owasp.org/www-project-kubernetes-top-ten/.

the OWASP Kubernetes Security Cheat Sheet[8]) and are directly linked with component types, threats and controls in the catalogue.

It must be observed that, while some source code scanners exist that may help automate this task and that are quite good at identifying insecure code patterns and vulnerabilities, they have several limitations. In fact, analyzers are often designed for specific frameworks or languages and fail to address issues outside of this scope; moreover, they are not able to spot design flaws, which are not specific to the code structure, while these may most likely be identified by a human. Thanks to the mapping between checklists, assets, threats, and controls, the relevant checklists to present to developers are retrieved automatically depending on the results of the previous phases, making the analysis more focused and effective.

3.3. *The building stage*

In the build stage, the code that has been committed to the shared repository (typically by multiple developers) is built, and a series of end-to-end, integration and unit tests are automatically run. Besides typical activities done in this phase such as vulnerability scanning, which can be automated thanks to available tools, our methodology devises a static security assessment phase corresponding to the SSDE *per-application security assessment* step, which aims at verifying (statically) whether the application correctly implements the security controls that have been identified at the end of the planning stage, when taking into account not only the local behavior and characteristics of each developed component (as resulting from the code review performed in the previous phase) but also the impact of the interrelationships among components and of the deployment configuration. The application security assessment is carried out by means of the adoption of automated reasoning techniques based on the knowledge of the application model and of the characterization of security controls, as extensively illustrated in (Rak, 2017) and (Casola et al., 2020a).

3.4. *The testing stage*

The security testing stage usually takes place after completing the coding and building steps, and before deploying the application in the production environment. During this stage, dynamic testing techniques such as DAST and penetration testing are used. DAST alone is not able to comprehensively test an application, on the other hand, penetration testing is a very expensive and time-consuming activity performed by security experts. Although penetration testing is conceived to be performed in production, it is usually carried out in this stage too, to reduce the financial impact in case of security faults. However, small/medium enterprises may not have enough resources to hire penetration testing teams to evaluate the security of their products. Therefore, testing activities could be demanded from developers with limited security backgrounds who tend to perform such activities without defining a precise testing strategy, by "randomly" using the most popular penetration testing tools such as those offered by Kali Linux (OffSec Services Limited, 2023), a Linux distribution designed for digital forensics and penetration testing.

As discussed in detail in the next section, in order to provide stronger support to developers in the security testing stage, we propose a model-based security testing strategy that helps them define a specific security test plan for the application under test, according to the knowledge acquired about existing security risks during the design and analysis phases.

4. A model-based security testing strategy

According to our proposal, once the application has been modeled, a developer will be automatically provisioned with a security test plan that encompasses:

- a list of attacks associated with the assets that compose the application model;
- the prerequisites under which these attacks can be performed and their mitigation;
- the tests to replicate the retrieved attacks.

Our security testing strategy relies on two relevant open source projects promoted by the MITRE Corporation[9] and both aimed at organizing knowledge about adversary behavior, namely the *Common Attack Pattern Enumeration and Classification* (CAPEC) (The MITRE Corporation, 2020) and the *Adversarial Tactics, Techniques & Common Knowledge* (ATT&CK) framework.[10] CAPEC provides a publicly available catalogue of common *attack patterns*, in terms of attributes and techniques employed by adversaries to exploit known weaknesses in applications. It is focused on application security and is meant to be used for application threat modeling and penetration testing. The listed attack patterns are not directly linked with specific attack tools and scripts, hence they cannot be directly used for security testing automation. CAPEC attack patterns belong to three different abstraction levels:

- *Meta Attack Patterns*: represent a very high-level characterization of a specific methodology or technique adopted to perform an attack (e.g., "Parameter Injection"). Conventionally, they do not refer to any technology or implementation;
- *Standard Attack Patterns*: provide more information about the methodology or technique used in an attack. They supply enough details to understand a specific technique and how it allows for achieving the final goal (e.g., the "Parameter Injection" meta-attack pattern can be specialized in "Email Injection");
- *Detailed Attack Patterns*: provide a detailed description of an attack. They are characterized by techniques that target some specific technology (e.g., the "Email Injection" standard pattern can be further specified in "Using Meta-characters in E-mail Headers to Inject Malicious Payloads"). Typically, a detailed attack pattern also supplies a complete execution flow. The higher level of detail of these attack patterns results in more specific mechanisms to mitigate them.

On the other hand, the ATT&CK framework is focused on network defense and describes how an adversary interacts with a system during pre/post-exploit operations. In particular, the framework details the specific tactics, techniques, and procedures used while targeting, compromising, and operating inside a network:

- *Tactics*: represent high-level tactical objectives of an adversary such as persistence, information discovery, lateral movement, credential access, and data exfiltration;
- *Techniques*: represent how an adversary achieves a tactical objective by performing an action. Techniques may also represent "what" an adversary gains by performing an action. There may be many techniques associated with a tactical objective: for instance, to achieve credential access, adversaries may attempt to position themselves between two or more network devices using an adversary-in-the-middle technique or a brute force approach;
- *Sub-techniques*: provide a more detailed description of how techniques behave at a lower abstraction level. For example, a brute

[8] OWASP, Kubernetes Security Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes _Security_Cheat_Sheet.html.

[9] https://www.mitre.org/.

[10] https://attack.mitre.org/.

force technique used to gain access credentials may be implemented through password guessing, cracking, or spraying;

• *Procedures*: represent examples of how particular adversaries use specific techniques and can be used for the replication of an incident with adversarial emulation or, from a different perspective, for security testing to verify the effectiveness of the enforced security measures.

The ATT&CK framework is currently used by many different organizations and industry sectors, including financial, healthcare, retail, and technology, and is a fundamental standardized reference for adversary emulation. In this regard, there are several initiatives aimed at supporting adversary emulation based on the ATT&CK framework. The most popular ones are Red Canary's Atomic Red Team[11] and MITRE's CALDERA.[12]

Atomic Red Team has the broadest coverage of ATT&CK techniques so far, including more than 500 individual lightweight tests that refer to roughly 160 ATT&CK techniques. On the other hand, CALDERA is a browser-based application and requires an agent to be installed on all the machines to test. While it can be used to identify and replicate adversary behavior based on ATT&CK techniques, similarly to Atomic Red Team, CALDERA represents a complex cyber security framework that allows performing advanced adversary emulation actions and incident response. In case an attack is not already implemented by CALDERA or by the Atomic Red Team, the developer should implement the detailed attack by himself, as discussed later.

The proposed approach, as opposed to other model-based testing methodologies, does not force developers to learn any kind of formalization of the threat model, build threat trees, or implement security tests. In order to obtain a specific security test plan and corresponding security test to reproduce, developers are only required to model their application through the graph-based MACM formalism and provide some high-level information about each component.

4.1. The extended security data model

The aforementioned frameworks are tightly coupled: many attack patterns enumerated by CAPEC are carried out by adversaries through specific techniques described by ATT&CK. In such cases, CAPEC attack patterns and the related ATT&CK techniques are cross-referenced. This enables contextual understanding of the attack patterns within an adversary's operational lifecycle.

In our approach, we employ Atomic Red Team and CALDERA test suites to execute tailored security tests aimed at verifying whether the detailed attack patterns, bound to the threats identified during the planning stage, can be executed despite the enforcement of countermeasures. Security tests to execute are automatically identified based on the relationships existing among the concepts of *Threat*, *Attack Pattern* as proposed by CAPEC, ATT&CK *Technique* and *Sub-Technique*, and corresponding atomic *Test* in Red Atomic and CALDERA.

Fig. 6 models these relationships by illustrating the extended security data model. Due to their high level of abstraction, CAPEC *Meta Attack* patterns may be included in the model and associated with the threats provided by our Threat Catalogue. Moreover, a *Detailed Attack* pattern, which is associated with a threat, *is child of* a *Standard Attack* pattern, that in turn *is child of* a *Meta Attack* pattern. Note that the model is technology independent and other tools may also be adopted. We initially leveraged the cross-references defined by MITRE to directly link the detailed attacks to the ATT&CK techniques that are, in turn, implemented by a set of different atomic tests belonging to both Atomic Red Team and CALDERA. Thanks to the above association, identifying the attacks to launch for each asset is immediate, as it depends on the retrieved threats.

In Fig. 7 we illustrate the whole process that leads to the identification of the security test plan. We consider automated operations (yellow box) all the activities that do not require any kind of user interaction. Semi-automated operations (white box) are the actions where the user is partially involved such as *Test Execution*, where a developer has to launch collected security tests also providing configuration parameters. Firstly, for each asset, the set of applicable threats is obtained from the catalogue and, then, the attack patterns associated with the threats are identified. For each attack pattern, the related techniques are retrieved and the corresponding tests are selected. The selection is based on the asset type and the execution platform (e.g., Windows or Linux OS), which are easily obtained from the MACM model by means of the *hosts* relationship. In case of test failures, developers are provided with useful feedback that allows them to identify which security controls have failed and fix them.

Most of the available tests can be launched by simply configuring some parameters (e.g., platform or server URL). Furthermore, since each technique may have more than one implementation, developers should run all the tests that implement that technique, either belonging to Atomic Red Team or CALDERA. Finally, developers will evaluate the outcome of a test by verifying its post-conditions or through the user interface when available. If a test fails, thanks to our reference model that links threats to security controls, it is possible to identify the failing controls and provide feedback that will be applied in the planning and/or coding stage.

At the time of this work, the CAPEC provides 559 attack patterns of which 61 are classified as meta, 181 as standard, and 317 as detailed. Currently, MITRE has cross-referenced 112 attack patterns with 53 ATT&CK techniques and 70 sub-techniques. In particular, 14 meta attack patterns are mapped to 9 techniques and 5 sub-techniques, 47 standard attack patterns are matched with 26 techniques and 19 sub-techniques, and, finally, 51 detailed attack patterns are associated with 24 techniques and 47 sub-techniques. However, coverage can be improved by developing new tests from scratch or by integrating additional test suites belonging to different projects, such as the OWASP ZAP tool (OWASP Foundation, 2021a) for web applications.

In this regard, the proposed approach is highly flexible, as it relies upon a complex knowledge base that can be easily extended to include new assets, threats, attack patterns, and testing tools/scripts. In fact, as already mentioned in Section 3.1, we extended the Threat Catalogue with container-related threats and, furthermore, we conducted some research based on vulnerabilities databases and current literature aiming at identifying container-related detailed attack patterns that could be associated with the CAPEC meta attack patterns. In particular, we were able to identify 9 specific threats related to containerization technology, which were associated with a set of existing CAPEC meta attack patterns. Then, we defined 5 detailed attack patterns along with their execution flow, prerequisites, and mitigation. These detailed attack patterns were matched to available ATT&CK techniques. Finally, we implemented a security test for the detailed attack pattern *Gaining Full Access to The System*, which will be discussed in Section 5.4.2.

As a final remark, our catalogues are available to the community in a public repository.[13]

5. A case study

In this section, to better illustrate the proposed approach, we provide a complete example of the application of our secure development methodology to a fairly complex microservice-based e-commerce store application. We practically show how it helps identify potential software security threats and the associated controls to mitigate them. We

[11] https://atomicredteam.io/.

[12] https://www.mitre.org/research/technology-transfer/open-source-software/caldera.

[13] https://github.com/ci-ma/MMSDT.

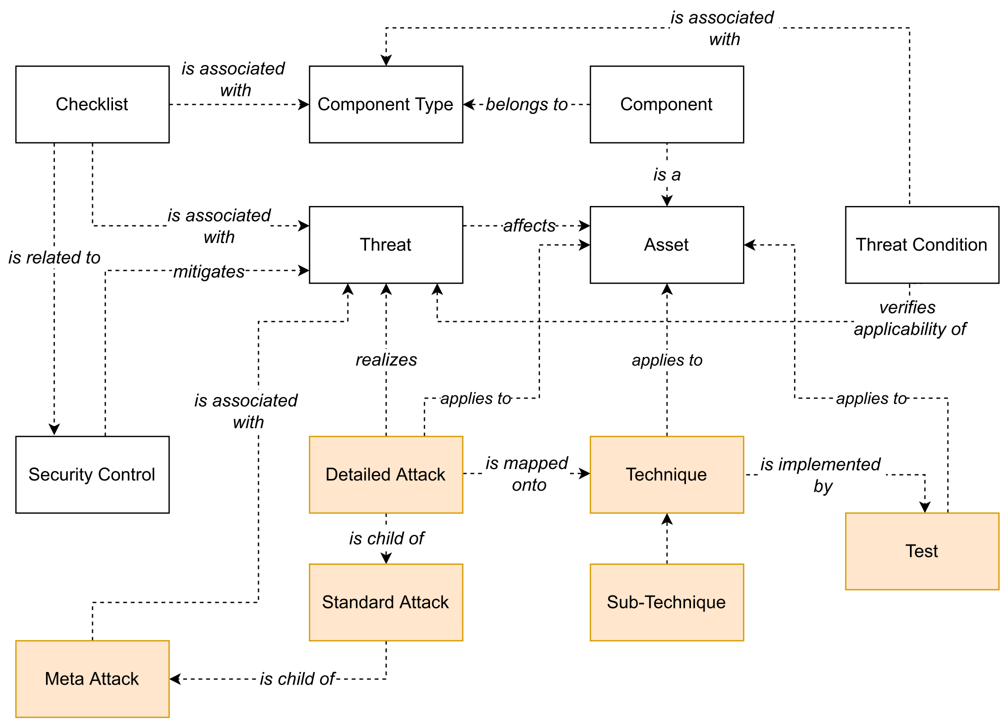

Fig. 6. Extended security data model.

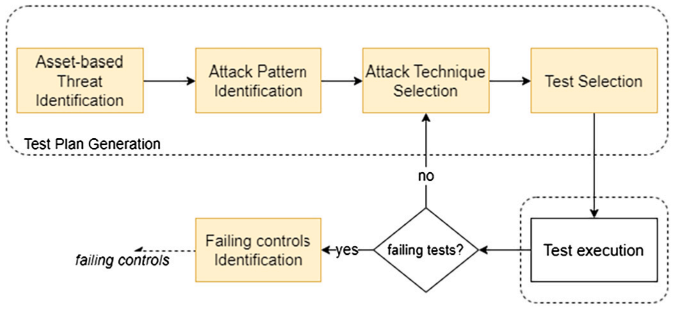
Asset-based Threat Identification
Attack Pattern Identification
Attack Technique Selection
Test Selection
Test Plan Generation
no
Test execution
failing tests?
yes
Failing controls Identification
failing controls

Fig. 7. Security testing process.

also show how the proposed security testing phase identifies a security test plan and corresponding security tests, which are tailored to the application under development.

5.1. Application description

Before illustrating all the steps of our methodology, we briefly introduce the main components of the application under study and its deployment configuration.

- *API-gateway*: single entry point for all clients. It receives all requests, which are then routed to the appropriate microservice;
- *Login*: receives the credentials required for authentication and verifies them through the interaction with the Vault service. In addition, it also generates, sends, and verifies OTPs;
- *Vault*: is responsible for authentication, authorization, encryption, and decryption. Two types of users are considered, i.e., *standard* and *administrator*;
- *Show*: through the interaction with MySQL, it obtains personal data, email, and product information. If a user has sufficient privileges, it allows adding new items. Furthermore, it also interacts with Vault to decrypt personal user data;
- *MySQL*: stores data related to users and products.

The application components have been deployed in a Kubernetes cluster, illustrated in Fig. 8, that comprises two modules: the *Kubernetes Control Plane* and the *Worker* with all nodes. The Control Plane is in charge of managing the cluster and hosts. Specifically, it comprises:

- *Kube-apiserver*: exposes the Kubernetes API to interact with the cluster itself;
- *Controller Processes*: a set of processes needed to manage the cluster (e.g. keep the right number of replicas);
- *etcd*: key-value database used to store all cluster information.

On the other hand, the Worker hosts the pods, a collection of Docker containers where each runs one of the aforementioned microservices that implement the application's business logic. Furthermore, it also runs the following additional components: (i) a *Kubelet*, to make sure that containers are running in a pod, (ii) a *Kube-proxy* to enforce network policies on the nodes (these policies allow network communication to pods from inside and outside the cluster), and (iii) a *Container Runtime* that is responsible for running containers. It is worth noting that Kubernetes may support several container runtime environments.

In Fig. 8, two types of connections are highlighted: the traffic between users and microservices (dotted lines), and the traffic among the microservices themselves. Although the API-gateway is the application entry point, all the traffic from inside and outside the cluster is routed through the Kube-proxy. All the requests sent by the administrator to the API server will be authenticated through a client certificate issued by a Certificate Authority.

In the following subsections, in compliance with the proposed methodology, we will illustrate the application of the proposed approach from planning to testing.

5.2. The planning stage

5.2.1. Application modeling

The application has been modeled, as shown in Fig. 9, using the MACM-CA formalism described in Section 3.1. The Kubernetes Control Plane and the Worker are two VMs, hence they were modeled by

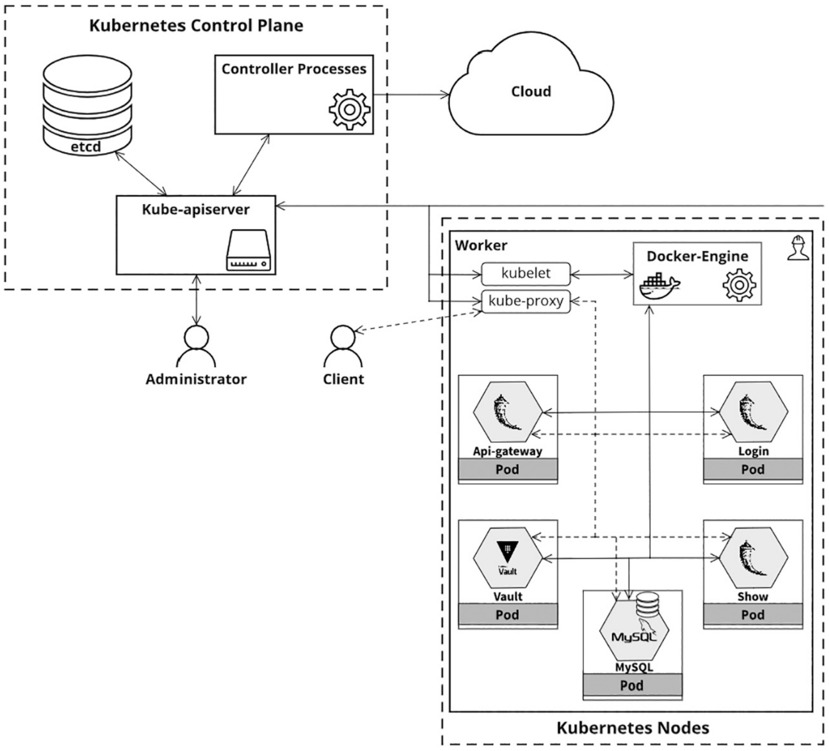

Fig. 8. Architecture of the case study.

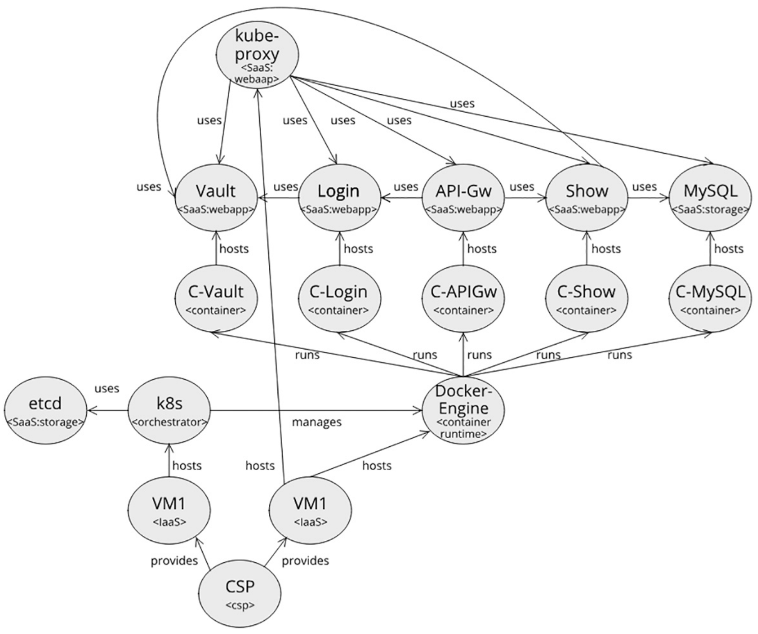

Fig. 9. MACM-CA representation of the application under study.

means of the `IaaS:Service` node type. All the microservices mentioned in the previous section were modeled as `SaaS:Service` node types belonging to the *web application* sub-type, except for MySQL which belongs to the *storage* sub-type. Since etcd is a key-value store, it was modeled as a storage service, too. Each microservice is *hosted* by a `Container` node, which in turn is *hosted* by a VM. Containers *run* under the control of the Docker Engine, represented by the `ContainerRuntime` node type. Finally, we modeled the Kubernetes components belonging to the Control Plane as `ContainerOrchestrator` nodes, responsible for *managing* the Docker Engine.

5.2.2. *Threat modeling*

As discussed in Sections 3.1 and 4, we extended the Threat Catalogue with additional information related to the new architectural assets to protect. In Table 4, we report some of the threats specific for the case study along with their descriptions. Table 5 provides some examples of association between threats and assets, highlighting also the STRIDE category associated with that threat.

Given the application assets to protect, the threat modeling step led to the identification of 194 threats that resulted in 41 unique threats, due to the fact that a threat affects more than a single component. As

Table 4
Examples of threats from the catalogue.

Threat	Description
Injection Flaws	The Injection Flaws occurs when data not validated are sent as part of a command or query to their interpreter. The data can deceive the interpreter into running commands not provided or accessing data for which you have no authorization.
Online Guessing	An attacker may try to guess valid username/password combinations
Unauthorized Entry	An adversary may gain access to a server or account without authorization
Spoofing External IPv6	An attacker in a container can craft IPv6 router advertisements, and consequently, spoof external IPv6 hosts, obtain sensitive information or cause a denial of service
Container Escape	An attacker can perform full container escape, gaining control over the host system

Table 5
Extract of threats-assets associations for the case study application.

Threat	Asset	STRIDE
Injection Flaws	MySQL	Tampering
Online Guessing	Vault	Elevation of Privilege
Unauthorized Entry	Login	Elevation of Privilege
Spoofing External IPv6	ContainerAPI-gateway	Spoofing
Container Escape	Docker-Engine	Elevation of Privilege

Table 6
Extract of final threats analysis with the associated security controls.

Threat	Security Controls
Injection Flaws	SI-10, SI-15
Online Guessing	AC-7, AC-9, IA-5, IA-5(1, 2, 4, 6, 7, 11, 12, 13)
Unauthorized Entry	AC-3, AC-5, AC-6(1, 3, 5, 9)
Spoofing External IPv6	AC-4(4, 5, 12, 19, 21), CM-7, SC-8(1, 3), SC-13
Container Escape	AC-6(1, 5, 8, 9, 10), AU-2, AU-3, CM-7, SC-7, SC-39

anticipated, some of the retrieved threats may not be of interest to the application under study and may be filtered out with the help of a questionnaire. We do not report this step here for the sake of brevity.

5.2.3. *Security controls identification*

Once all the applicable threats have been identified, developers are automatically provided with the security controls deemed to mitigate them. In the current implementation of the catalogue, the associations between threats and security controls are accomplished taking into account all the NIST security controls and related control enhancements. For the application under study, we selected proper security controls to mitigate the threats highlighted by the threat modeling subphase. As threats may affect multiple assets, security controls may be required to protect more than one asset. For example, in Table 6, we report the security controls and the control enhancements (in brackets) required to mitigate the threats outlined in Table 4. From there, we decided to prioritize the most popular security control families: Access Control (AC), Identification and Authentication (IA), Audit and Accountability (AU), System and Communications Protection (SC), and System and Information Integrity (SI).

For the sake of clarity, we discuss the more relevant security controls that were added or enriched after this analysis. In particular, the enforcement of the AC family security controls led to the introduction of access control and network policies (AC-4) along with the limit of consecutive unsuccessful login attempts per user (AC-7). The IA security controls highlighted the need to introduce two-factor authentication for the Administrator (IA-2). Moreover, mutual authentication between the microservices was introduced using Istio[14] (IA-9), a completely open-source service mesh that allows connecting, securing, and controlling microservices. As far as the AU controls are concerned, we enforced audit policies to determine what happened in the cluster, when it happened, and who did it. The SC security controls implementation was covered by the use of Kubernetes and Vault facilities, the former to grant a minimum amount of resources to containers, namely resource quotas and limit ranges (SC-6), and the latter to protect information at rest using its transit secret engine (SC-28). Finally, the proper use of input validation was added in order to enforce the SI controls (SI-10, SI-15).

Fig. 10 shows the updated architecture of the application under study, resulting from the enforcement of the security controls. The assets affected by changes are highlighted in different colors and they have been modified as follows:

- *Kube-apiserver*: the configuration of the Kube-apiserver was modified to set up audit policies and log files;
- *Kube-proxy*: we enforced network policies to manage internal/external flows within the cluster;
- *Docker-Engine*: was configured with the live restore functionality which allows containers to run even when the daemon becomes unavailable;
- *Istio sidecar/microservices*: an additional Istio sidecar proxy mechanism was enforced to grant mutual TLS authentication among all the microservices.

5.3. *The coding and building stages*

As discussed in Section 3, the planning stage is followed by a coding stage, where a checklist-based code review is performed, and a building stage, where static security assessment activities are carried out. For the sake of brevity, we do not report here the details of these activities which have been extensively addressed in our previous work.

5.4. *The security testing stage*

In the testing stage, the application was deployed on a production-like environment built on top of Proxmox Virtual Environment,[15] a virtualization platform that allows managing virtual machines and containers. In particular, we deployed the Kubernetes Control Plane and Worker on two virtual machines both equipped with 4 CPU(s), 4.00 GiB of RAM, 20.00 GiB of boot disk, and Ubuntu 18.04 as the operating system. The Kubernetes cluster was created by means of *kubeadm*,[16] which enables creating a minimum cluster compliant with best practices. Moreover, in order to perform penetration testing activities, we locally set up a Kali Linux attacker machine and, by taking advantage of a virtual proxy network, the traffic generated by the attacker was suitably routed towards the target machines.

5.4.1. *Security test plan identification*

According to the security testing stage presented in Section 4, developers are provided with a specific security test plan for the application under test that we built according to the following steps:

[14] https://istio.io.

[15] https://www.proxmox.com/en/proxmox-ve.

[16] https://github.com/kubernetes/kubeadm.

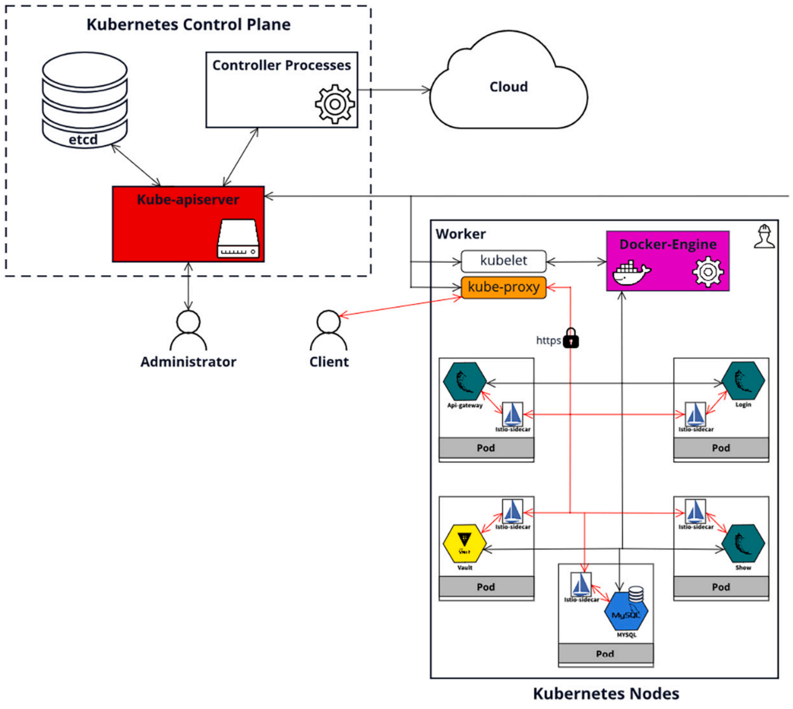

Fig. 10. Final architecture of the application under study.

1. Starting from the threats related to each asset, obtained as the result of the threat modeling, the corresponding CAPEC meta attack patterns are determined;
2. For each meta attack pattern matched, the standard attack patterns and then the corresponding detailed attack patterns are collected;
3. ATT&CK techniques associated with the identified detailed attack patterns and applicable to the specific assets of interest are retrieved;
4. Atomic tests implementing the retrieved techniques are selected.

For the application under test, we retrieved 219 detailed attack patterns, partitioned as follows:

- 83 provided with their detailed execution flow, prerequisites and mitigation;
- 136 without any detailed execution flow, but at least provided with a description of the attack. Some of them also present prerequisites, and/or mitigation;

Starting from this partitioning, we filtered out the attacks by platform and discarded *Windows-based attacks* and *Android/iOS-based attacks*. Furthermore, the process of attack filtering led to the identification of 30 detailed attack patterns which were not suitable to the application under study, and so excluded from the security test plan. In fact, we assumed having deployed our resources in a secure environment that cannot be accessed by unauthorized entities, besides discarding attacks depending on platforms that have not been adopted, we also filtered out attacks based on physical access to the machine/documentation. Finally, the plan only includes the detailed attack patterns that provide an execution flow, prerequisites, and mitigation. The resulting security test plan foresees 53 detailed attack patterns. For the sake of clarity, in Tables 7 and 8, we report an extract of the final plan with meta and detailed attack patterns and corresponding threats.

Note that, as discussed in Section 3.1 and 4, the threats and the detailed attack patterns associated with the assets introduced in the MACM-CA extension have been retrieved thanks to our preliminary work to enrich the catalogue and the methodology models. In this way, security expert knowledge is properly codified so as to automatically provide developers with the detailed attack patterns associated with the threats collected from their MACM model.

5.4.2. Security tests execution

Besides automatically providing developers with a security test plan specific to the application under test, our methodology also aims at automating test execution, as much as possible. As discussed in Section 4, this automation relies upon the association between detailed attack patterns and ATT&CK techniques, implemented by various adversarial emulation tools. In this case study, we focused on the implementation offered by CALDERA and Atomic Red Team. Of the 48 detailed attack patterns that compose the security test plan, only 9 are already implemented by at least one of the aforementioned tools. This implies that according to the identified detailed attack, a developer must implement all missing attacks.

Table 9 highlights the techniques retrieved from the ATT&CK framework associated with the detailed attack patterns of our security test plan. In particular, for each technique, we reported the number of available abilities/tests in both CALDERA and Atomic Red Team. When a detailed attack pattern is implemented by more than one technique, the number of tests that perform a technique is separated by a comma. For example, considering the *Log Injection-Tampering-Forging* attack, CALDERA provides 31 abilities to perform the attack, while Atomic Red Team offers 39 tests. Since these tests are independent of each other, a developer should execute all of them.

It is important to note that CALDERA and Atomic Red Team do not provide any abilities/tests for the new assets related to the MACM-CA introduced in Section 3.1. In this case, as already said, the developer needs to implement some specific test to perform the detailed attacks, to be included in the automatic test execution. For example, for the application under test, we implemented an automated test for the *Gaining Full Access to the System with Docker* attack shown in Table 8 in order to include it in the automatic test execution.

Finally, in the following paragraphs, we report the results of test executions of two attacks: the *Log Injection-Tampering-Forging* attack (using both CALDERA and Atomic Red Team) and the *Gaining Full Access to the System with Docker* attack implemented by ourselves, by also providing specific feedback to developers. In both cases, we assumed that the

Table 7
Extract of detailed attack patterns applicable to the application.

Threat	Meta Attack Pattern	Detailed Attack Pattern
Online Guessing	Brute Force	Password Spraying
Online Guessing	Brute Force	Try Common or Default Usernames and Passwords
Guessing Access Tokens based on acquired Knowledge	Excavation	Directory Indexing
Broken Authentication and Session Management	Identity Spoofing	Spear Phishing
Advanced Persistent Threats (APTs)	Infrastructure Manipulation	Log Injection-Tampering-Forging
Weak Identity, Credential & Access Management	Reverse Engineering	Retrieve Embedded Sensitive Data
Spoofing	Action Spoofing	Credential Prompt Impersonation
Spoofing	Action Spoofing	iFrame Overlay
Injection Flaws	Exploitation of Trusted Identifiers	Reusing Session IDs (Session Replay)

Table 8
Examples of detailed attack patterns associated with the container and container runtime threats.

Threat	Meta Attack Pattern	Detailed Attack Pattern
Container Escape	Privilege Escalation	Gaining Full Access to the System with Docker
Container Escape	Privilege Escalation	Container Breakout using runC
Container Escape	Code Injection	Full Container Escape using docker cp Command and libnss
Spoofing External IPv6	Identity Spoofing	Obtain Sensitive Information or Cause a Denial Service using CAP_NET_RAW Capability
Denial of Service	Denial of Service	Denial of Service via Large Integer

Table 9
Number of abilities/tests associated with the detailed attack patterns from our security test plan.

Detailed Attack Pattern	ATT&CK Technique(s)	# CALDERA Abilities	# Atomic Red Team Tests
Reusing Session IDs (Session Replay)	T1134.001, T1134.002, T1550.004	0, 0, 0	2, 1, 0
Password Spraying	T1110.003	4	4
Try Common or Default Usernames and Passwords	T1078.001	0	2
Directory Indexing	T1083	7	4
Spear Phishing	T1534, T1566.001, T1566.002, T1566.003	0, 2, 0, 0	0, 2, 0, 0
Log Injection-Tampering-Forging	T1070	31	39
iFrame Overlay	T1021	18	12
Credential Prompt Impersonation	T1021	18	12
Retrieve Embedded Sensitive Data	T1552.004	8	7

attacker already has a foothold inside the application by having compromised a user account on the machine where our services have been deployed.

Test 1: Log Injection-Tampering-Forging
Test description: *Log Injection-Tampering-Forging* is a tampering attack that modifies or deletes the content of one or more log files. Adversaries usually modify these files with the purpose of masking malicious behavior.
Test execution and result: Due to the significant number of available abilities/tests offered by CALDERA and Atomic Red Team, we considered the subset of such tests that belong to the *Indicator Removal on Host: Clear Command History* sub-technique. To reproduce these sub-techniques using CALDERA, we had to install its agent on the target machine. After having configured it by simply choosing the abilities to reproduce, the tests executed led to the discovery of some vulnerabilities related to access control of the file containing the history of the bash commands. In particular, the agent was able to tamper and delete it. As far as the Atomic Red Team's corresponding tests are concerned, their execution led to the same outcomes.
Feedback to developers: The failed security control is *AC-6 (Least Privilege)* and involves all the machines where the application has been deployed. Therefore, in order to mitigate the aforementioned vulnerabilities, proper authorization rules for machines must be applied, and only authorized users should be allowed to access the *bash_history* file.

Test 2: Gaining Full Access to the System with Docker
Test description: *Gaining Full Access to the System* is a privilege escalation attack that enables a malicious user to gain full access to the system. When an administrator allows an unprivileged user to access the docker group, she/he will be also allowed to make use of the Docker CLI to create containers. Since Docker runs with the SUID bit set, an attacker may exploit it to abuse the file system and gain more privileges on the target.
Test execution and result: Through our newly implemented test, a developer without particular security skills should be able to reproduce this attack. In particular, our test is a bash script that foresees the following steps:

1. *Verify access to the Docker group*: in order to perform the attack, a user should be already added to the Docker group. If the permission check fails, the test will be stopped;
2. *Try to run a container*: upon verifying the capability to use the docker CLI, the user will be able to run a container;
3. *Pull an image from the Docker Hub*: she/he will pull an image from the Docker Hub;
4. *Gaining full access to the system*: exploiting the SUID bit of Docker, a malicious user is able to mount the full file system directory into the container previously launched and then, through the `chroot` command, gain full privileges on the target machine.

For the application under test, the attack was successful. By executing the script, we were able to gain full access to the system.
Feedback to developers: The involved assets are the Docker Engine and the VM where the Kubernetes Control Plane is deployed. Also in this case, the failed security control is *AC-6 (Least Privilege)*, whose mitigation consists in properly configuring access control policies, too: only privileged users should be allowed to access the Docker group.

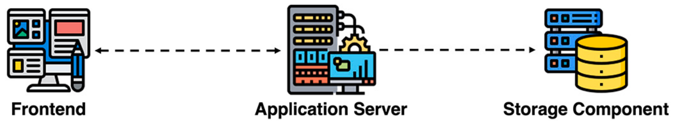

Fig. 11. Architecture of OWASP Juice Shop.

Table 10
Extract of threats-assets associations for OWASP Juice Shop.

Threat	Asset	STRIDE
Web User Account Hijacking	Frontend	Spoofing
Sensitive Data Exposure	Frontend	Information Disclosure
Sniffing Storage Traffic	Storage Component	Information Disclosure
Deletion of Data	Storage Component	Tampering
Web Session Hijacking	Application Server	Spoofing
Container Escape	Container	Elevation of Privilege

Table 11
Extract of security test plan for OWASP Juice Shop.

ATT&CK Technique	Asset	# Tests
Network Sniffing	Storage Component	8
Impair Defenses: Impair Command History Logging	Frontend	10
Valid Accounts	Application Server	6

5.5. *Assessing the effectiveness of our methodology with Juice Shop*

To better illustrate the usefulness and effectiveness of the proposed methodology, we demonstrate its application on a well-known software project widely used for security training, awareness demos, capture-the-flag events etc., namely the OWASP Juice Shop project.[17] Juice Shop is a vulnerable web application containing several hacking challenges of varying difficulty, developed by OWASP for vulnerability detection and security assessment purposes. OWASP provides information about the vulnerabilities affecting Juice Shop, useful to evaluate our methodology.

Since we are dealing with an already-developed application, we can only assess the threat modeling and security testing stages.

In the threat modeling phase, we modeled the target application using the MACM-CA formalism. This allowed us to identify the potential threats affecting each application component: 44 threats related to the three main components of Juice Shop, i.e., frontend, application server, and storage component, depicted in Fig. 11. Table 10 reports an extract of such findings. It is worth noting that Juice Shop is deployed as a containerized application, making the newly defined container-specific threats applicable.

Given the list of threats, we retrieved the corresponding ATT&CK techniques to define a security test plan for the target application. The security test plan encompassed 20 ATT&CK techniques in total, implemented by 148 tests from Atomic Red Team. Since we deployed the application in a Linux-based environment, 21 tests were not applicable because they targeted Windows-based systems, resulting in 127 applicable tests. Table 11 illustrates an extract of the security test plan for Juice Shop.

The threats identified during the threat modeling phase cover most vulnerabilities of Juice Shop, which can be identified leveraging the corresponding tests in the security test plan. Table 12 shows the coverage of Juice Shop's vulnerabilities for the proposed methodology. These vulnerabilities either match threats in our threat catalogue (e.g., Injection, Sensitive Data Exposure, Unvalidated Redirects) or map onto threats that further specify the vulnerability (in case the vulnerability has wide

Table 12
Coverage of Juice Shop's vulnerabilities for the proposed methodology.

Vulnerability Type	Juice Shop Threat Model	Proposed Methodology
Broken Access Control	✔	✔
Broken Anti-Automation	✔	
Broken Authentication	✔	✔
Cross Site Scripting (XSS)	✔	✔
Cryptographic Issues	✔	✔
Improper Input Validation	✔	✔
Injection	✔	✔
Insecure Deserialization	✔	✔
Security Misconfiguration	✔	✔
Security through Obscurity	✔	
Sensitive Data Exposure	✔	✔
Unvalidated Redirects	✔	✔
Vulnerable Components	✔	✔
XML External Entities (XXE)	✔	

scope, e.g., Broken Authentication, Security Misconfiguration, Vulnerable Components).

For the sake of brevity, for the security tests execution step, we refer to the tests associated with the *Impair Defenses: Impair Command History Logging* technique. As shown in Table 11, Atomic Red Team offers 8 tests to execute this technique, such as *Disable history collection*, *Setting the HISTFILE environment variable*, *Setting the HISTCONTROL environment variable*, and *Setting the HISTFILESIZE environment variable*. These tests aim to impair command history logging by modifying the *HISTFILE*, *HISTCONTROL*, and *HISTFILESIZE* environment variables. Editing such variables results in clearing command history, not logging commands already present in the history, or forcing the history size to zero respectively. To execute them, we employed the scripts provided by Atomic Red Team: all the tests succeeded in impairing logging history and clearing log files since there is no need to escalate privileges to perform such operations. The feedback to developers is the following: these tests violate the *AU-2 (Event Logging)* security control and affect all the components of Juice Shop (since the security test plan identified these tests for each of them). Consequently, all the machines hosting these components must be configured to grant access to such resources only to authorized users.

6. Related work

Several methodologies have been proposed in recent years to address software security (Kudriavtseva and Gadyatskaya, 2022). Such methodologies fall under the umbrella of Secure Software Development Life Cycle (SSDLC) models, which involve integrating security practices both at the organizational and technical levels into an existing development process. Examples of organizational practices include security role identification and security training, while main technical practices are threat modeling, security review, and security testing, which are the focus of this paper.

Microsoft Security Development Lifecycle (SDL)[18] is one of the most popular existing methodologies in industry. It embeds security practices in almost all development phases, including requirements definition and tracking, security quality metrics definition and monitoring, threat modeling, security design and implementation (mostly in terms of recommendations), static and dynamic testing and penetration testing. Although SDL is presented and promoted as platform-agnostic, it is mainly suited for Microsoft products and the underlying technologies and resources are proprietary, therefore they are not straightforward to understand (Kudriavtseva and Gadyatskaya, 2022). Similarly to our proposal, SDL automates threat modeling by leveraging a suitable application model and a threat knowledge base. While SDL adopts Data-Flow

[17] https://owasp.org/www-project-juice-shop/.

[18] https://www.microsoft.com/en-us/securityengineering/sdl.

Diagrams (DFDs) to model the system to analyze, with plenty of components and interconnection types (especially related to Microsoft products), we use a simpler graph-based model (MACM) with fewer (but extensible) components related to high-level software system architectural paradigms (e.g., cloud-based applications, micro-services applications, IoT applications), thus easier to build. Regarding the threat knowledge base, the SDL Threat Modeling Tool uses a proprietary (yet extensible) list of threats (some of which specifically related to Microsoft Azure products), whereas our Threat Catalogue natively integrates several open-source threat sources from industrial standards and best practices. The biggest difference with our approach is that in SDL the results of threat modeling are not linked to subsequent development phases related to countermeasure selection and testing. Concerning testing, apart from traditional SAST and DAST analyses, SDL prescribes a penetration testing activity but, contrary to our approach, there is no specific methodology/tool suggested for this activity.

Besides Microsoft, it is worth mentioning OWASP and its multiple initiatives related to secure software development. Among these, it is possible to cite CLASP (Comprehensive, Lightweight Application Security Process),[19] a process targeted to small organizations that formalizes a set of security activities to carry out during the development lifecycle, and assigns them to specific project roles. The scope of CLASP is wider than that of SDL, but no supporting tools have been provided and, besides, the project seems not to be supported anymore. Moreover, OWASP has recently introduced its DevSecOps Maturity Model (DSOMM) (OWASP Foundation, 2017b), built upon the well-known Software Assurance Maturity Model (OWASP SAMM) (OWASP Foundation, 2021b), which provides an overview of almost 140 low-level security practices applicable to a DevOps environment.

With regard to security automation, OWASP has released the Threat Dragon[20] open-source tool for threat modeling. The tool supports STRIDE, LINDDUN (for privacy), and CIA threat taxonomies, but currently includes only a few high-level threats. Even in this case, threat modeling results are not directly linked to design and testing activities. As for testing, OWASP Zed Attack Proxy (ZAP)[21] is an easy-to-use penetration testing tool for finding vulnerabilities in web applications, designed to be used by developers and functional testers who are new to penetration testing. Despite its efficacy, ZAP is not integrated into a wider development methodology, therefore there is no direct connection between the threats identified during threat modeling and the actual tests that are launched during the assessment.

Among the scientific community, Kumar and Goyal (2020) proposed the adoption of a framework to use open-source software in order to simplify SecDevOps implementation. This framework extends the main principles of DevOps by introducing a conceptual model, called ADOC, which codifies security controls into an automated workflow that can be applied to all DevOps phases to grant a specific level of security assurance. The security controls are then implemented using open-source software, which enables the integration of continuous security in the DevOps paradigm. Kumar and Goyal further extended their work by defining a continuous security model for cloud applications (Kumar and Goyal, 2021). Their model introduces new principles, practices, and stages to be integrated into the DevOps application lifecycle and also identifies the automation ecosystem to use. Although this work is very relevant to the subject of security automation, again there is no connection between the different phases of the secure development process, more specifically between threat modeling and testing.

As a general remark, it must be noted that existing security automation tools and methods, especially related to testing such as SAST, DAST and penetration testing, often require significant security experience by operators, which is rarely available in small enterprises. Moreover, human intervention is still heavily required in order to define and prioritize tests, configure the tools, and interpret results. Without careful test planning, penetration testing risks losing both efficacy and efficiency. In addition, regardless of the adopted testing strategy, most of the existing approaches do not support developers in identifying a *targeted* test plan but rely upon the security expertise of developers to select (or write from scratch) atomic tests to execute. These strategies often do not help in the identification of the assets that expose security issues and the security controls that should be improved.

Our proposal aims to overcome the aforementioned limits by leveraging model-based testing techniques (Felderer et al., 2016). In model-based testing, suites of test cases are automatically generated from a set of models of the system under test, created during the analysis and design phases. This approach focuses on the data model and architecture, instead of hand-crafting individual tests (Dalal et al., 1999). Felderer et al. (2011) argue that the use of security models raises the level of abstraction of test design. This enables the reduction of the level of expertise required to design security tests, thus allowing more people to perform this task. Moreover, the use of models helps to partially/fully automate the generation of security tests. The adoption of model-based security testing offers several benefits, as highlighted by the authors. However, in order to leverage its advantages, a complete approach to security modeling and testing is needed.

An example of model-based testing is provided by (Xu et al., 2012), where an automated technique for software vulnerability detection was introduced. The authors presented an approach based on the formalization of threat models through Predicate/Transaction nets and on the mapping between the individual elements of a threat model to their implementation constructs. This method enables converting all the attack paths, identified by the threat model, into executable code. Although the effectiveness of this approach was supported by two case studies, its main limitation lies in the need for a formal specification of the threat model to automate security test code generation. Similarly, in (Marback et al., 2013) the authors proposed a threat model-based security testing approach that automatically generates security test cases from threat trees. The proposed technique requires developers to build threat trees and the accompanying tests, so its effectiveness heavily depends on the developers' skills.

Maciel et al. (2019) realized Robot, a framework capable of automatically generating test cases derived from the Requirement Specification Language (RSL) (da Silva, 2017). Tests are generated thanks to a mapping between the constructs of RSL and the Robot language. Almubairik and Wills (2016) presented an algorithm to systematically generate a penetration testing plan guided by a threat model. However, the algorithm is not discussed in detail so it can not be reused. Furthermore, it does not cover all the threats reported by the threat model, including common threats such as SQL injection and cross-site scripting. Another example of model-based penetration testing is provided by Xiong and Peyton (2010): in their work, they define a methodology that leverages the collaboration with developers to achieve quality penetration test campaigns. They also define a grey-box, model-driven test architecture to automate the main processes in penetration testing and a structured representation of Web security knowledge to be processed by test platform programs, to make test results more reliable, measurable, and assessable.

7. Conclusions

The integration of security activities in a DevOps pipeline requires a high level of automation, which can be only partly achieved through the multitude of development, deployment, and testing tools available on the market. The poor adoption of such tools and the lack of security models have been identified as the main open issues that hamper the definition of a comprehensive SecDevOps methodology.

[19] OWASP, CLASP. https://owasp.org/www-pdf-archive/Us_owasp-clasp-v12-for-print-lulu.pdf.

[20] https://www.threatdragon.com/.

[21] https://www.zaproxy.org/.

In this paper, we proposed a methodology that aims to support developers with limited security skills in the development of secure applications. In particular, we focused on enriching our previous development methodology with a novel semi-automated security testing phase. Our approach allows developers to automatically define a security test plan for their applications and provide them with the corresponding set of security tests to reproduce. This significantly reduces the amount of time usually spent on penetration testing activities and on the identification of the assets and mitigation that failed during the tests. These features make it particularly suited for securing modern development methodologies since potential changes to the software will not affect its delivery time. The proposed methodology relies upon a security data model and catalogues that are able to codify the security expert's knowledge. Thanks to our approach, any developer will be capable of making informed decisions about security design choices in each development phase and improving her/his awareness of security issues and controls. The catalogues leveraged by our methodology are publicly available at the address https://github.com/ci-ma/MMSDT.

As a final remark, the validity of our approach heavily depends on the completeness and correctness of the underlying knowledge base, the Threat Catalogue. Completeness is hard to evaluate and impossible to achieve in absolute terms, as the security landscape is dynamic by nature (new threats and attacks emerge over time), but the approach is extensible by design. On the flip side, correctness is partly pursued by the adoption of well-established and renowned international standards and projects from which all the key concepts and data are derived.

In future work, we plan to further enrich the catalogue by including threats specific to software technologies and refine the countermeasure selection step by considering more specific mitigation tailored to the different assets instead of generic, technology-agnostic security control. Moreover, we aim to involve industrial partners to validate our methodology.

CRediT authorship contribution statement

Valentina Casola: Conceptualization, Methodology, Project administration, Supervision, Writing – review & editing. **Alessandra De Benedictis:** Conceptualization, Methodology, Project administration, Supervision, Validation, Writing – original draft, Writing – review & editing. **Carlo Mazzocca:** Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Vittorio Orbinato:** Methodology, Software, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

Agrawal, P., Rawat, N., 2019. Devops, a new approach to cloud development testing. In: 2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), vol. 1, pp. 1–4.

Almubairik, N.A., Wills, G., 2016. Automated penetration testing based on a threat model. In: 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), pp. 413–414. https://ieeexplore.ieee.org/abstract/document/7856742.

Casola, V., De Benedictis, A., Rak, M., Villano, U., 2020a. A novel security-by-design methodology: modeling and assessing security by slas with a quantitative approach. J. Syst. Softw. 163, 110537.

Casola, V., De Benedictis, A., Rak, M., Villano, U., 2020b. A methodology for automated penetration testing of cloud applications. Int. J. Grid Util. Comput. 11 (2), 267–277.

Cloud Security Alliance, 2011. Consensus Assessment Initiative Questionnaire. https://cloudsecurityalliance.org/group/consensus-assessments/.

da Silva, A.R., 2017. Linguistic patterns and linguistic styles for requirements specification (I): an application case with the rigorous RSL/business-level language. In: Proceedings of the 22nd European Conference on Pattern Languages of Programs, EuroPLoP '17. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3147704.3147728.

Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J., Lott, C.M., Patton, G.C., Horowitz, B.M., 1999. Model-based testing in practice. In: Proceedings of the 21st International Conference on Software Engineering, pp. 285–294.

Felderer, M., Agreiter, B., Zech, P., Breu, R., 2011. A classification for model-based security testing. In: Advances in System Testing and Validation Lifecycle (VALID 2011), pp. 109–114.

Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A., 2016. Chapter one - security testing: a survey. In: Advances in Computers, vol. 101. Elsevier, pp. 1–51. https://www.sciencedirect.com/science/article/pii/S0065245815000649.

Gartner, Inc., 2020. Forecast Analysis: Container Management (Software and Services), Worldwide. https://www.gartner.com/en/documents/3985796.

Govil, N., Saurakhia, M., Agnihotri, P., Shukla, S., Agarwal, S., 2020. Analyzing the behaviour of applying agile methodologies devops culture in e-commerce web application. In: 2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184), pp. 899–902.

Granata, D., Rak, M., 2021. Design and development of a technique for the automation of the risk analysis process in IT security. In: Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER. INSTICC, SciTePress, pp. 87–98.

Granata, D., Rak, M., Salzillo, G., 2022. Risk analysis automation process in it security for cloud applications. In: Ferguson, D., Helfert, M., Pahl, C. (Eds.), Cloud Computing and Services Science. Springer International Publishing, Cham, pp. 47–68.

Haindl, P., Plösch, R., 2019. Towards continuous quality: measuring and evaluating feature-dependent non-functional requirements in devops. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 91–94.

Kudriavtseva, A., Gadyatskaya, O., 2022. Secure software development methodologies: a multivocal literature review. arXiv:2211.16987.

Kumar, R., Goyal, R., 2021. When security meets velocity: modeling continuous security for cloud applications using devsecops. In: Innovative Data Communication Technologies and Application. Springer, pp. 415–432.

Kumar, Rakesh, Goyal, Rinkaj, 2020. Modeling continuous security: a conceptual model for automated DevSecOps using open-source software over cloud (ADOC). Comput. Secur. 97 (6), 101967. https://www.sciencedirect.com/science/article/pii/S0167404820302406.

Leite, L., Rocha, C., Kon, F., Milojicic, D., Meirelles, P., 2019. A survey of devops concepts and challenges. ACM Comput. Surv. 52 (6). https://doi.org/10.1145/3359981.

Maciel, D., Paiva, A.C., Da Silva, A.R., 2019. From requirements to automated acceptance tests of interactive apps: an integrated model-based testing approach. In: ENASE, pp. 265–272.

Marback, A., Do, H., He, K., Kondamarri, S., Xu, D., 2013. A threat model-based approach to security testing. Softw. Pract. Exp. 43 (2), 241–258. https://doi.org/10.1002/spe.2111. arXiv: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2111. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2111.

Microsoft, 2016. The STRIDE Threat Model. https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).asp.

National Institute of Standards and Technology, 2013. SP 800-53 Rev 4: Recommended Security and Privacy Controls for Federal Information Systems and Organizations. Tech. rep. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf.

OffSec Services Limited, 2023. Kali Linux. https://www.kali.org/.

OWASP Foundation, 2016. The OWASP Risk Rating Methodology Wiki Page. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

OWASP Foundation, 2017a. Code review guide v2. https://owasp.org/www-project-code-review-guide/assets/OWASP_Code_Review_Guide_v2.pdf.

OWASP Foundation, 2017b. DevSecOps Maturity Model. https://dsomm.timo-pagel.de/.

OWASP Foundation, 2021a. OWASP Zed Attack Proxy (ZAP). https://owasp.org/www-project-zap/.

OWASP Foundation, 2021b. Software Assurance Maturity Model (OpenSAMM). https://owaspsamm.org/.

Rak, M., 2017. Security assurance of (multi-)cloud application with security SLA composition. Lect. Notes Comput. Sci. 10232, 786–799.

RedHat Inc., 2020. What is container orchestration? https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

Sánchez-Gordón, M., Colomo-Palacios, R., 2020. Security as culture: a systematic literature review of devsecops. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20. Association for Computing Machinery, New York, NY, USA, pp. 266–269. https://doi.org/10.1145/3387940.3392233.

Šćekić, M., Gazivoda, M., Šćepanović, S., Nikolić, J. Application of devops approach in developing business intelligence system in bank. In: 2018 7th Mediterranean Conference on Embedded Computing (MECO), pp. 1–4.

Sultan, S., Ahmad, I., Dimitriou, T., 2019. Container security: issues, challenges, and the road ahead. IEEE Access 7, 52976–52996. https://doi.org/10.1109/ACCESS.2019.2911732.

The MITRE Corporation, 2020. Common Attack Pattern Enumeration and Classification (CAPEC). https://capec.mitre.org/.

Tomas, N., Li, J., Huang, H. An empirical study on culture, automation, measurement, and sharing of devsecops. In: 2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), pp. 1–8.

Waseem, M., Liang, P., 2017. Microservices architecture in devops. In: 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), pp. 13–14.

Xiong, P., Peyton, L., 2010. A model-driven penetration test framework for web applications. In: 2010 Eighth International Conference on Privacy, Security and Trust, pp. 173–180.

Xu, D., Tu, M., Sanford, M., Thomas, L., Woodraska, D., Xu, W., 2012. Automated security test generation with formal threat models. IEEE Trans. Dependable Secure Comput. 9 (4), 526. https://ieeexplore.ieee.org/document/6155723.

Valentina Casola, Dr. is an Associate Professor at the Department of Electrical Engineering and Information Technology of the University of Naples Federico II, Italy. She got a Ph.D. in Computer Engineering from the Second University of Naples in 2004. She has published more than 100 papers in journals, conference proceedings and books. Her research activities are both theoretical and experimental and focus on security methodologies to design and evaluate distributed systems, including cyber physical infrastructures, cloud systems and web services. These activities are led in cooperation with academic institutions and industrial partners within national and international projects.

Alessandra De Benedictis received her M.S. degree in Computer Engineering in 2009 and her Ph.D in Computer and Automation Engineering in 2013, both from the University of Naples Federico II, Naples, Italy. She is currently an assistant professor at the Department of Electrical Engineering and Information Technology of the University of Naples Federico II. Her research interests mainly involve the design and evaluation of secure architectures for the protection of distributed systems. She is particularly interested in the definition of methodologies for the development of applications able to offer well-defined security guarantees, both in the cloud environment and in presence of resource constraints. Other relevant research activities include the investigation on moving target defense mechanisms and on embedded security solutions based on reconfigurable hardware.

Carlo Mazzocca received his M.Sc. and B.Sc. degrees in Computer Engineering in 2018 and 2020, respectively, both from the University of Naples Federico II, Italy. He is currently a Ph.D. student in Computer Science and Engineering at the University of Bologna, Bologna, Italy. His research interests mainly include digital identity, security mechanisms based on distributed ledger technologies, and authentication and authorization solutions for the cloud-to-thing continuum.

Vittorio Orbinato is a PhD Student at Università degli Studi di Napoli Federico II, Italy. He received his M.Sc. and B.Sc. degrees in Computer Engineering in 2018 and 2020, respectively, both from the University of Naples Federico II, Italy. His research interests include threat emulation based on AI techniques and software security.