A Cyber Risk Based Moving Target Defense Mechanism for Microservice Architectures

Preprint · July 2018

DOI: 10.13140/RG.2.2.27463.60323/1

Kennedy Torkura
Hasso Plattner Institute
38 PUBLICATIONS **211** CITATIONS
SEE PROFILE

Muhammad Ihsan Haikal Sukmana
Hasso Plattner Institute
31 PUBLICATIONS **181** CITATIONS
SEE PROFILE

Anne V. D. M. Kayem
83 PUBLICATIONS **305** CITATIONS
SEE PROFILE

Feng Cheng
Peking University
107 PUBLICATIONS **1,637** CITATIONS
SEE PROFILE

A Cyber Risk Based Moving Target Defense Mechanism for Microservice Architectures

Kennedy A. Torkura, Muhammad I.H. Sukmana, Anne V.D.M. Kayem, Feng Cheng, and Christoph Meinel

Hasso-Plattner-Institute for Digital Engineering, University of Potsdam, Potsdam,Germany
{kennedy.torkura,muhammad.sukmana,anne.kayem,feng.cheng,christoph.meinel}@hpi.de

Abstract. Microservice Architectures (MSA) structure applications as a collection of loosely coupled services that implement business capabilities. The key advantages of MSA include inherent support for continuous deployment of large complex applications, agility and enhanced productivity. However, studies indicate that most MSA are homogeneous, and thus vulnerable to *code reuse attacks*, and as such serves as an *economics-of-scale* incentive to attackers. In this paper, we address the issue of code reuse vulnerabilities with a novel solution based on the concept of *Moving Target Defenses* (MTD). Our mechanism works by performing risk analysis against microservices to detect and prioritize vulnerabilities. Thereafter, a risk-oriented *diversification index* is derived to define the depth of diversification to be implemented. Generative programming techniques e.g. OpenAPI code generators are then used to automatically transform the programming languages and container images of the microservices. Consequently, the attack surfaces are altered at *runtime* to introduce uncertainty for attackers and reduce the *attackability* of the microservices. The microservices attack surfaces are concurrently modelled and analyzed to detect the attack opportunities and correlate the dependence relationship of *shared* vulnerabilities. The results of this analysis is instrumental for objective validation of the *diversification efficiency* by comparing the previous attack surfaces with the corresponding diversified versions. Our experiments demonstrate the efficiency of our solution, with an average success rate of over 50 % attack surface randomization.

1 Introduction

Microservice Architectures (MSA) have become the *defacto* standard for agility and productivity in computing domains due to benefits such as speed and productivity. However, MSA introduce multiple security risks due to image vulnerabilites, embedded malware and configuration defects e.t.c [1]. While existing research indicate prevalence of vulnerabilities in container images [2, 3], risks in homogeneous MSA due to *shared code vulnerabilities* (i.e. *code reuse attacks*) are not yet investigated. Most MSA are *homogenous* i.e. composed of microservice instances built from identical *base images* thus vulnerable to *code reuse* *attacks* [4, 5]. Therefore, correlated attacks can be launched against microservices with common attack vectors and high success rates [6]. The problem is worsened by the high number of severe vulnerabilities infecting official and community container images [3, 7] on public image repositories e.g. DockerHub. Novel *container-aware*, risk analysis methodologies are imperative to counter these security challenges.

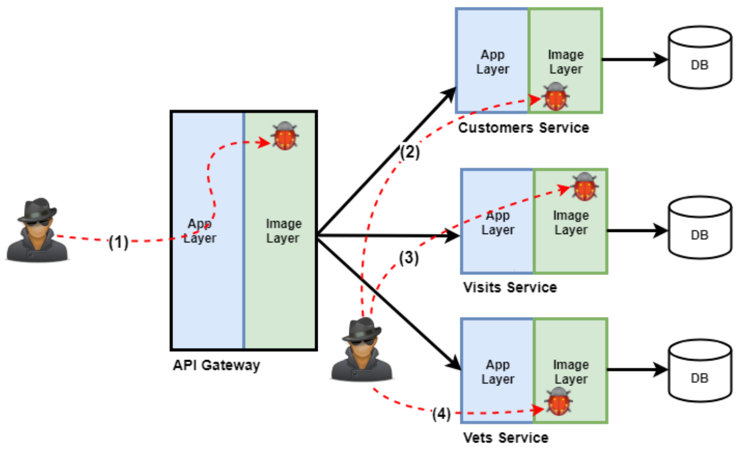

Fig. 1: Successful Code Reuse Attack Against PetClinic Due to Homogeneity.

A possible counter-measure against the aforementioned challenges is deployment of Moving Target Defenses (MTD). MTD are techniques that transform specified system components to create uncertainty for attackers, thereby reducing the probability of successful attacks i.e. *attackability* [8, 9]. The main goal is to prevent attackers from exploiting knowledge acquired about target systems due to homogenous composition and software monoculture. For example, an attacker might exploit an XSS vulnerability in the API Gateway of the PetClinic microservice application illustrated in Figure 1 (step 1). Assuming the exploit affords him control of the API Gateway, the attacker might thereafter attack the Customer service with a vulnerability (step 2). Given the attack is successful, the attacker might replay the same attack (steps 3 and 4) against Visits and Vets services and achieve the same success as in previous attacks, leading to *correlated failures*. The success of this correlated attacks is due to use of the same image for building PetClinic microservices, which amounts to sharing of vulnerabilities. MTD generally deploy *security-by-diversity* tactics e.g. Address Space Layout Randomization (ASLR) [10], instruction-level [11] and basic block-level [12] transformations. Similarly, software homogeneity is common in MSA and diversification techniques could be employed as mitigative measures.

Contribution Consequently, we propose *MTD* mechanisms to overcome the security implications of homogeneous microservices. Our approach consists of multi-layered, risk analysis techniques that evaluate target MSA and propose diversification options. We leverage our container-aware concept [13, 14] through which vulnerabilities are detected using the *Security Gateway* and ranked with a *vulnerability-correlation* based risk prioritization scheme. The notion of *diversification index* is introduced as a metric for expressing the depth of diversification to be implemented. Using the OWASP Risk Rating Methodology (ORRM) and Common Vulnerability Scoring System (CVSS), a *Security Risk* is derived per microservice and vulnerability correlation matrices are generated to illustrate vulnerability dependencies between microservices. This approach makes it possible to alter programming languages of the microservices and corresponding attack surfaces. The direct effect of our approach is randomization of the microservices attack surfaces to defeat anticipated attacks. Hence, the attack surfaces are evaluated before and after diversification to verify the *diversification efficiency.*

The rest of this paper is structured as follows, the next section presents related works. Section 3 briefly highlights background information on container technologies and the security implications of homogeneous MSA. In Section 4, a *running example* of the aforementioned challenges is described, and our risk analysis techniques are discussed. The implementation details of our proposed prototype is presented in Section 5. In Section 6, we evaluate our work, while Section 7 concludes the paper and provides insights on future work.

2 Related Work

Baudry et al [15] introduced *sosiefication*, a diversification method which transforms software programs by generating corresponding replicas through statement deletion, addition or replacement operators. These variants still exhibit the same functionality but are computationally diverse. Williams et. al [16] introduced *Genesis*, a system that employed VMs for enabling dynamic diversity transforms. Through the use of the *Strata* VM, software components are distributed such that every version is unique, hence difficult to attack. In [6], the authors characterized models to represent correlated failures due to shared code vulnerabilities and presented diversification strategies as a mitigative solution. Larsen et al. [9] compared several automated diversification techniques and provided a systematization of these techniques.

Security Monitor is a microservice-aware system proposed in [17] to observe and discover anomalies and thereafter diversify the causative logic. However, this work assumes the MSA employs polyglot programming and details of the diversification requirements and techniques are not provided. Kratzke et 'al. [18] proposed a biology-inspired immune system that takes advantage of the agile properties of MSA to improve security. Our proposed mechanism can be combined with this work to improve resiliency.

3 Background and Problem Statement

In this Section, we briefly present background information on *container virtualization* concepts and ecosystems, and discuss the security implications of deploying homogeneous MSA.

3.1 An Overview of Application Containers

Application containerization is an OS-level virtualization technology designed to run and maintain multiple packaged applications in isolated processes. These packaged applications are executed from static *container images*, which are *templates* for producing application containers. A container image consists of several images arranged in layers with the underlying root layers being the *parent image* followed by the *base image* [1]. Custom applications are built atop base images in layers, each layer adding custom software artefacts as defined in *Dockerfiles*. Since each layer is a static component of several software binaries, these layers could be infected by vulnerabilities affecting their corresponding software packages. Unfortunately, unlike traditional software packages where software updates are rolled downstream, application images must be updated by end-users [1].

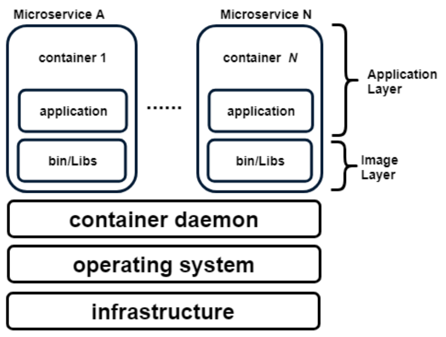

Fig. 2: Application Container Ecosystem

3.2 Problem Statement

MSA consists of several autonomous, loosely coupled, polyglot components (*microservices*) operating jointly as an application [19]. The microservice architectural style supports *polyglotness* at the persistence layer and use of diverse programming languages [20]. Polyglot persistence [21] is widely practiced since it affords flexible deployment of different database types according to workload requirements. Conversely, polyglot programming models are not favored

[1] https://docs.docker.com/develop/develop-images/baseimages/

given the introduction of complexities and maintenance requirements. Hence, homogeneous microservices are more prevalent and prefered [21]. This practice introduces insecurity since the vulnerabilities infecting base images are directly inherited across the entire application. The security implications of shared code vulernabilities were demonstrated in several works e.g. Nappa et.al [4] detected 69 shared vulnerabilities between two Mozilla products: Firefox and Thunderbird, given the sharing of common codebases. The probability to compromise a MSA depends on the number of microservice instances that are attacked concurrently. These correlated attacks are simpler for homogenous microservices due to shared code vulnerabilities such that higher numbers of shared vulnerabilities imply higher probabilities of successful attacks i.e. *attackability* [9]. Therefore, we aim at reducing the attackability of MSA by minimizing shared vulnerabilities leveraging MTD mechanisms. Software diversification aims at frustrating attackers by randomizing attack surfaces such that attackers are *blinded* thereby reducing the motivation to attack and overall attackability. However, employing MTD to microservices is challenging for various reasons. First, the basis for diversifying is unclear i.e how to achieve high entropy of composed microservices. Second, microservices are still novel and there are yet no standards, hence it is difficult to decide on implementations strategies. Third, given the ephemeral nature of MSA, it is challenging to keep apace with the current state of the deployed microservice e.g. due scaling requirements.

4 Design and System Model

In this Section, we illustrate the security implications of homogeneous MSA with a running example, and thereafter present our analysis of microservice attack surfaces and how MTD can mitigate risks due to these attack surfaces. The last sub-section briefly highlights on microservice vulnerability correlation matrix.

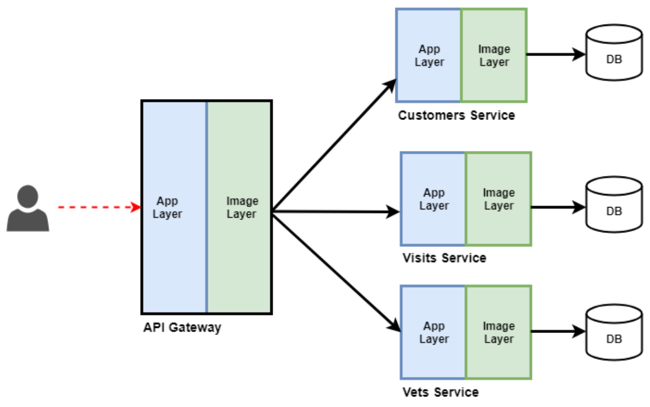

Fig. 3: Microservice-based PetClinic Application Comprising Four Microservices

4.1 Running Example - Security Risks in Homogeneous MSA

We use the Spring PetClinic microservices-based application [2] as a *running example* to discuss the security issues of homogeneous microservices. PetClinic is an application used by Pivotal [3] to demonstrate technologies and design patterns implemented with the Java Spring framework. [4] PetClinic consists of the *API Gateway*, *Visits*, *Vets* and *Customers* microservices. These microservices are derived from the same base image hence the vulnerabilities affecting these images are inherited. Though the vulnerabilities at the application layer might differ, there are various security implications for having almost identical security vulnerabilites in closely related applications [4]. For example, an attacker might discover the XSS vulnerability in the API Gateway, and exploit it to gain control of the API Gateway. Next, the attacker compromises the other 3 microservices using a single vulnerability e.g. *CVE-2016-7167* [5]. A consequence of shared vulnerabilities is that once an adversary is successful in exploiting one of the microservice instances, similar attacks can be propagated simply by repeating the same attack sequence. Furthermore, the availability of most images on public image repositories such as DockerHub provides Open Source Intelligence (OSINT) and *economies of scale* [9] for attackers. They can easily analyze all images on DockerHub for vulnerabilities and use this knowledge to orchestrate large scale attacks against MSA.

4.2 Risk Analysis for Microservice Diversification

In Section 3, we provided justification for microservice diversification i.e. to counter security issues due to homogeneous microservices. To provide a structured procedure to support diversification decision making, we employ security metrics to design a cyber risk-basedMTD mechanism. Security metrics are useful tools for risk assessment due to inherent encapsulation of risk quantification variables [22]. We aim at computing risks per microservice and thereafter employing *vulnerability prioritization* such that diversification is a function of microservice risk assessment (diversification is ordered by risk severity). To characterize this requirement, we introduce the notion of *Diversification Index* - D_i, as an expression of the depth of diversification to be implemented. Given a microservice-based application M consisting a set of microservice instances $\{m_1, m_2, m_3, \ldots, m_n\}$, we express Di as follows:

$$D_i = m_d/n$$

where, m_d is number of microservices to be diversified and n is the total number of microservices instances in the application. Therefore, D_i provides a flexible guidance on the number of microservices to be transformed. Due to specific

[2]https://github.com/spring-petclinic/spring-petclinic-microservices
[3]https://pivotal.io/
[4]https://spring.io/
[5]https://security-tracker.debian.org/tracker/CVE-2016-7167

requirements, developers might not want to diversify some microservices which can be excluded. A second metric is constructed, the *Security Risk* - SR, to provide a numeric expression of microservice's security state. We employ two approaches for determining SR:

Risk Analysis Using CVSS The CVSS [23] is a widely adopted vulnerability metrics standard. It provides vulnerability severity *base scores* which can extracted from vulnerability scan reports. In order to derive the SR, the base scores of all detected vulnerabilities can be summed and averaged [22] as expressed below:

$$SR = \frac{\sum_{i=1}^{N} V_i}{N}$$

where V_i is the CVSS base score of vulnerabilities i, and N is the total number of vulnerabilities detected in microservice m. However, averaging vulnerability to obtain single metrics for representing microservice security state is not optimal because the derived values are not sufficiently representative of other factors such as availability of exploits for vulnerabilities. Therefore, we employ another scoring technique namely, the *shrinkage estimator* [24]. This approach has been popularly used for online rating systems such as Internet Movie Database (IMDB), and is well suited to our case because it considers not just the average rating but also the number of vulnerabilities. The *shrinkage estimator* is expressed as:

$$SR = (v/(v+m)).R + (m/(v+m)).C$$

where, v is the total number of vulnerabilities detected in a microservice instance, m is minimum number of vulnerabilities required to be included in the assessment, and C is the mean vulnerability security score. Following, the Pearson's correlation coefficient is derived to determine the dependence relationship between the microservices.

Risk Analysis Using OWASP Risk Rating Methodology The risk assessment method described in the previous subsection is limited to vulnerabilites published in the Common Vulnerabilities and Exposures (CVE) dictionary. The CVE does not provide an exhaustive representation of web application vulnerabilities, thus we need to derive another risk assessment methodology for web vulnerabilities. We use the ORRM, which is specifically designed for web applications [25]. The ORRM expresses risks as a function of two parameters: the *likelihood of occurrence* of a vulnerability and the *potential impact* due to vulnerability exploitation. More formally this is expressed as:

$$Risk = Likelihood * Impact$$

These parameters are to be assigned by risk assessors considering threat vectors, possible attacks, and impacts of successful attacks.

4.3 An Anatomy of Microservice Attack Surfaces

A core aspect of our methodology is manipulation of microservice attack surfaces against attackers through random architectural transformations. We adopt Howard et. al's *attack opportunities* concept [26] for defining attack surfaces. The attack opportunities concept defines attack sufaces along three abstract dimensions: *targets and enablers, channels and protocols, and access rights* which are used for estimating attack likelihood [27]. Hence, to reduce attack probability for microservices, we identify and alter the attack surfaces e.g. entry and exit points. However, to enable better insights into these attack surfaces, we categorize them into: *horizontal* and *vertical* attack surfaces and thereafter employ *vulnerability correlation* to establish relationship between vulnerabilities.

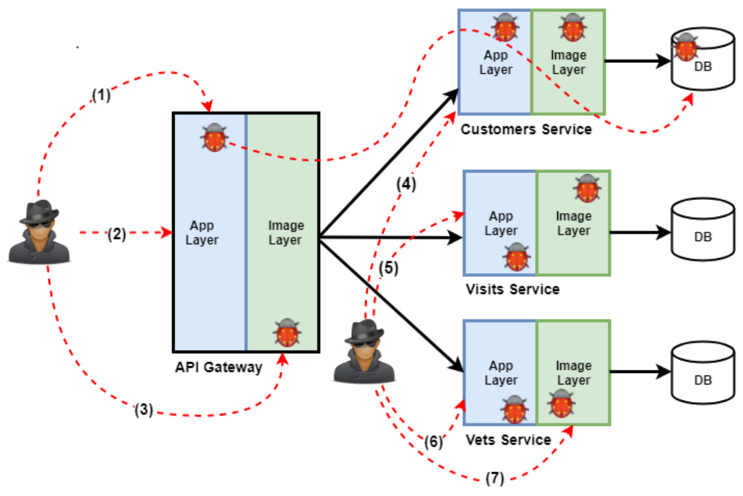

Fig. 4: Typical Microservice Attack Surfaces illustrated with the PetClinic

Horizontal Vulnerability Correlation The objective of correlating vulnerabilities horizontally is analyze the relationship of vulnerabilities along the horizontal attack surface i.e. the parts of the applications users directly interact with. Figure 4 illustrates the multi-layered attack surface of PetClinic. The application layer horizontal attack surface consists of interactions against the exit/entry points from the API gateway to the Vets,Visits and Customer services application layers. Requests and responses are transversed along this layer, providing attack opportunities for attackers. Our vulnerability correlation process is similar to *security event correlation* techniques [28], though rather than clustering similar attributes e.g. malicious IP addresses, we focus on Common Weakness Enumeration (CWE) Ids. CWE is a standardized classification system for application weaknesses [6]. For example, CWE 89 categorizes all vulnerabilities related to *Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)* [7] and can be mapped to several CVEs e.g. *CVE-2016-6652* [8], a SQL injection vulnerability in Spring Data JPA. If this vulnerability exists in all PetClinic's microservices, an attacker could easily conduct a correlated attack (*Attack Paths 2,4,5 and 6* of Figure 4) resulting to correlated failures and eventual application failure since each microservice works ultimately to the successful functioning of the PetClinic application.

Vertical Vulnerability Correlation The vertical correlation technique is similar to the horizontal correlation, however the interactions across application-image layers are analyzed. In Figure 4, attack Path 1 illustrates the exploitation of a vulnerability across the vertical attack surface. The attacker initiates an attack against the PetClinic's API Gateway, and moves laterally through the application-image layer. From there, another attack is launched through the Customers service's application-image layer and finally the database is compromised. The same attack can be repeated against the other microservices hence the need to correlate casual relationships between vulnerabilities.

4.4 Microservices Vulnerability Correlation Matrix

Correlated vulnerabilities can be represented with correlation matrices, more specifically refered to as *microservices vulnerability correlation matrix*. Therefore, we are intuited by definition in [6] to define microservices vulnerability correlation matrix as *a mapping of vulnerabilities to microservice instances in a microservice-based application*. The *microservices vulnerability correlation matrix* presents a view of vulnerabilities that concurrently affect multiple microservices. An example of the microservice correlation matrix is Figure 5, where M_1 and M_2 will have a correlated failures under an attack that exploits vulnerability V_1 since they share the same vulnerability. However, an attack that exploits V_2 can only affect M_1, while M_2 remains unaffected.

$$\begin{array}{c} \\ M_1 \\ M_2 \\ \\ M_n \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & \dots & V_n \end{array} \\ \begin{bmatrix} 1 & 1 & \cdots & \dots \\ 1 & 0 & \cdots & \dots \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & \dots \end{bmatrix} \end{array}$$

Fig. 5: Microservice Vulnerability Correlation Matrix

[6]https://cwe.mitre.org/index.html

[7]https://cwe.mitre.org/data/definitions/89.html

[8]https://nvd.nist.gov/vuln/detail/CVE-2016-6652

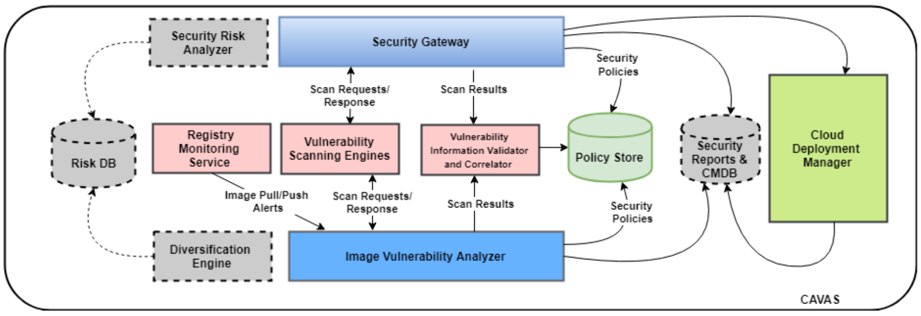

Fig. 6: CAVAS Architecture Showing New Components in Dashed Lines.

5 Implementation

Our MTD mechanism is implemented in software components which have been integrated into our previous work: Cloud Aware Vulnerability Assessment System (CAVAS) [13, 29]. In Figure 6, the new components are distinguished with ash colors and bordered with *dashed lines*. For the purposes of this paper, we discuss only the components related to our proposed solution.

5.1 Security Gateway

The Security Gateway leverages cloud design patterns for vulnerability assessment by modifying the behaviour of the *service discovery and registry* server to suit security testing requirements. This approach enables integration of SEPs for security policy enforcement. Implementation details of the Security Gateway are described in [13].

5.2 Image Vulnerability Analyzer

The Image Vulnerability Analyzer (IVA) leverages Anchore [9] to conduct static vulnerability analysis analysis of container images. Anchore is an open-source tool for analyzing Docker images. Anchore retrieves vulnerability information from several sources and supports 3rd party integration via an API. Vulnerability analysis is performed by extracting and inspecting underlying image layers for vulnerabilities defined in the retrieved vulnerability information. An accompanying image *metadata* may also contain information sufficient for vulnerability analysis. IVA is implemented in Java and Spotify Docker Java client library [10], it interacts the Anchore API.

[9]https://github.com/anchore/anchore-engine
[10]https://github.com/spotify/docker-client

5.3 Application Vulnerability Analyzer

OWASP ZAP [11]is deployed to perform vulnerability scanning against microservices application layer. OWASP ZAP is a popular open-source dynamic vulnerability scanner. It's API enables automation and integration with CAVAS, it also supports ingestion of OpenAPI documents.

5.4 Security Risk Analyzer

The *Security Risk* - *SR* is computed by the Security Risk Analyzer (SRA) using the risk analysis methodologies described in Sections 4.2 & 4.3. SRA retrieves vulnerability information from the Security Reports & CMDB for the target microservice to be analyzed. Similarly, to analyze the horizontal & vertical attack surfaces, queries are made against the DB for requisite vulnerability information, which is subsequently matched against the CWE of interest.

5.5 Diversification Engine

The diversification engine receives the *diversification bleuprint* from the SRA and transforms the target microservice-based application using the OpenAPI code generation library [12]. This library generates client and server stubs in different languages/frameworks by ingesting representative OpenAPI documents. To achieve this, OpenAPI leverages *dynamic reflection* and Aspect Oriented Programming techniques to understand internal architectures of target microservices [30]. Based on this understanding and program models represented by OpenAPI documents, applications are transformed from one programming language to another. Also, Dockerfiles are generated for building container images which are subsequently executed into containerized microservices.

6 Evaluation

We conducted experiments to verify the efficiency of our diversification strategy. The PetClinic application was deployed in a local development environment on a Windows 10 laptop configured as follows: Intel (R) Core (TM) i5-5200U CPU, 2.20Ghz processor speed, 12GB RAM and 250 GB HDD. Two experiments are conducted: (1) Security risk comparison to verify the efficiency of our security-by-diversity tactics (2) Attack surface analysis to evaluate the improvement in the horizontal and vertical attack surfaces.

[11]https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[12]https://github.com/swagger-api/swagger-codegen

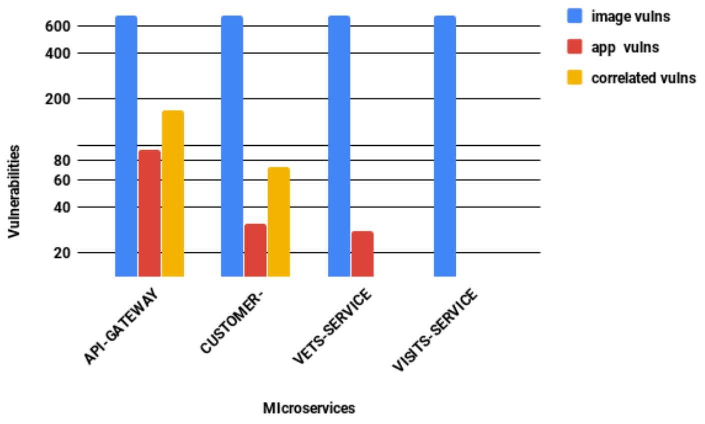

Fig. 7: Vulnerability scanning results of the Homogeneous PetClinic application

6.1 Security Risk Comparison

The vulnerability scanners integrated into CAVAS (Anchore and OWASP ZAP), were used for performing vulnerability scans against PetClininc images and microservice instances respectively. The detected vulnerabilities were persisted in the Security Reports and CMDB (Figure 6). First, the diversification index is derived by computing risks per PetClinic microservices to obtain the *Security Risk* - SR. Hence, we inspect the results for the image vulnerability scan and realize the vulnerabilities are too identical (Figure 7). Therefore, SR will be so similar that vulnerability prioritization would not be beneficial for our risk-based approach, which aims at arranging microservicies in order of risk severity. So, we compute SR for the application layer using the ORRM (Section 4.2). The app layer scan results are retrieved from the db and analyzed. Scores are assigned to the detected vulnerabilities based on the risk scores for OWASP top 10 2017 web vulnerabilities [31]. This is a reasonable approach given OWASP uses ORRM for deriving the top 10 scores. Also, this affords us objective assignment of scores which are publically verifiable [13]. Table 1 is the distribution of detected vulnerabilities, while a subset of the mapping between CWE-Ids and OWASP Top 10 is on Table 2. From Table 2, it is obvious that the API-Gateway has the most severe risks followed by the Visits, Vets and Customer microservices. Therefore, we apply diversification based on this result using a *diversification index of 3:4* i.e three out of four microservices. The diversification blueprint is passed to the diversification engine as follows: API-Gateway-Python, Customers-service:ruby, Vets-service:nodejs. The diversified PetClinic is *retested* and the results are illustrated in Figure 8. We observe that the diversified PetClinic application layer vulnerabilities are reduced with about 53.3 %. However, the image vulnerabilities

[13]https://www.owasp.org/index.php/Top_10-2017_Details_About_Risk_Factors

increased especially for the Customer and Vets service which are transformed to NodeJS and Ruby respectively. Importantly, the microservices are no longer homogeneous, and the possibilities for correlated attacks have been eliminated. Also, the vulnerabilities in the API Gateway's image are drastically reduced from 696 to 6, while the application layer vulnerabilities reduced from 94 to 24. The reduction is due to reduced code base size, a distinct characteristic of Python programming model. The API Gateway is the most important microservice, since it presents the most vulnerable and sensitive attack surface of the application, therefore consider the security of PetClinic improved, our results means that out of 94 opportunities for attacking the API Gateway, only 24 were left.

Table 1: Distribution of Detected App-Layer Vulnerabilities in PetClinic

CWE-ID	**API-GATEWAY**	**CUSTOMERS-SERVICE**	**VETS-SERVICE**	**VISITS-SERVICE**
CWE-16	31	4	2	2
CWE-524	48	17	6	11
CWE-79	0	3	0	1
CWE-425	0	0	20	0
CWE-200	14	6	0	0
CWE-22	0	1	0	0
CWE-933	1	0	0	0
TOTAL	94	31	28	14

Table 2: Risk Scores By CWE

CWE-ID	**OWASP T10 Risk Category**	**Risk Score**
CWE-16	A6 - Security Misconfiguration	6.0
CWE-524	Not Listed	3.0
CWE-79	A6 - Security Misconfiguration	6.0
CWE-425	Not Listed	3.0
CWE-200	A3 - Sensitive Data Exposure	7.0
CWE-22	A5 - Broken Access Control	6.0
CWE-933	Not Listed	3.0

6.2 Attack Surface Analysis

Here we analyze the attack surfaces of the homogeneous and diversified PetClinic versions. We consider direct and indirect attack surfaces i.e. vulnerabilities that directly/ indirectly lead to attacks respectively. From the vulnerability scan reports, each detected vulnerability is counted as an attack surface *unit* (*attack opportunities concept* [26]). Figure 9 compares the horizontal app layer attack surface for both PetClinic apps. Notice a reduced attack surface in the

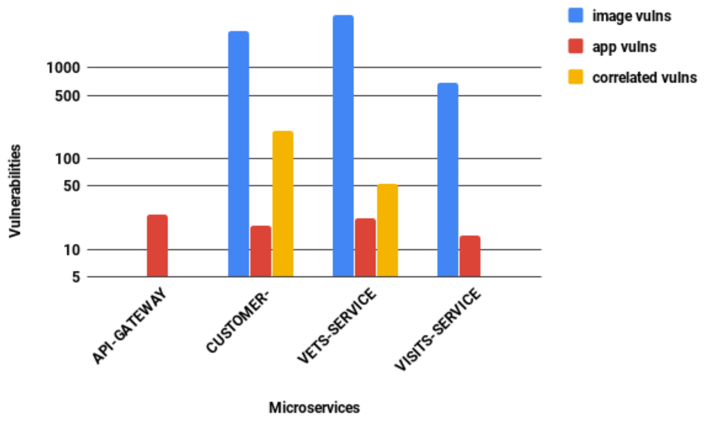

Fig. 8: Vulnerability scanning results of the Diversified PetClinic Application

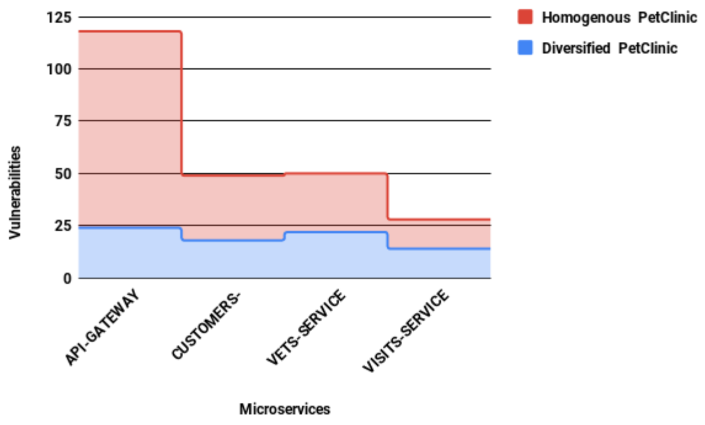

Fig. 9: Horizontal Application Attack Surface Analysis

diversified version, showing better security. Essentially, the attackability of Pet-Clinic has been reduced, however the results for the vertical attack surface are different. This attack surface portrays attacks transversing the app-image layers. While there are fewer correlated vulnerabilities in the diversified version of the API-Gateway, correlated vulnerabilities in the Customers and Vets Services have increased. This increment is due to the corresponding increase of image vulnerabilities. However, the attackability due to homogeneity is reduced. This result can be improved by combining secure coding practices in development pipelines e.g policy based risk assessments, continuous vulnerability scanning/-patching. In our previous paper [14], we presented how CAVAS automates these processes. Also, configuration-based vulnerabilities e.g. *X-Content-Type-Options Header Missing*, can be fixed by appending appropriate headers. Similarly, image vulnerabilities can be reduced e.g. *Alpine Linux* images can be used as base images instead of Ubuntu images since their smaller footprint equals smaller attack surface[32].

7 Conclusion and Future Work

Container technologies e.g. Docker and Kubernetes have become core components for agility and productivity. However, recent investigations indicate high rates of vulnerability infection among docker images. Furthermore, most containerized applications employ homogeneous architectures, i.e. identical container images are used for every microservice composing the application. Consequently, these microservices share similar vulnerabilities and are therefore vulnerable to *code reuse* attacks. In this paper, we have proposed a cyber risk based mechanism for employing MTD to counter attacks due to the aforementioned reasons. We extend our previous work on automated security assessments of microservices to identify and prioritize security risks. The notion of *diversification index* is introduced as a security metric for expressing the depth of desired diversification. *Automatic code generation* techniques and *security-by-diversity* tactics are then used to automatically transform the programming technologies of target microservices to less vulnerable equivalents. This effectively changes the attack surfaces, thereby reducing the microservices *attackability.* Our practical evaluation demonstrates the efficiency of our security tactics, over 50 % of attack surface are randomized, hence improving security. Microservices are suitable for employing MTD, therefore our work is preliminary, several aspects can be improved. For example, a synergy of the security potentials of MSA asserted by Kratzke et 'al. [18] and our mechanism can be explored. This approach uses microservices attributes e.g. *auto-scaling*, *resiliency* and *self-healing*, for security hardening and rapid recovery from cyber-attacks.

References

[1] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide", 2017. [Online]. Available: `https://doi.org/10.6028/NIST.SP.800-190`.

[2] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective", *IEEE Cloud Computing*, 2016.

[3] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub", in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017.

[4] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching", in *Security and Privacy (SP), 2015 IEEE Symposium on*, IEEE.

[5] O. H. Alhazmi and Y. K. Malaiya, "Application of vulnerability discovery models to major operating systems", *IEEE Transactions on Reliability*, vol. 57, no. 1, pp. 14–22, 2008.

[6] P.-Y. Chen, G. Kataria, and R. Krishnan, "Correlated failures, diversification, and information security risk management", *MIS quarterly*, pp. 397–422, 2011.

[7] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in docker hub contain high priority security vulnerabilities", Technical report, BanyanOps, Tech. Rep., 2015.

[8] D. Evans, A. Nguyen-Tuong, and J. Knight, "Effectiveness of moving target defenses", in *Moving Target Defense*, Springer, 2011, pp. 29–48.

[9] P. Larsen, S. Brunthaler, L. Davi, A.-R. Sadeghi, and M. Franz, "Automated software diversity", *Synthesis Lectures on Information Security, Privacy, & Trust*, vol. 10, no. 2, pp. 1–88, 2015.

[10] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving target defense: creating asymmetric uncertainty for cyber threats.* Springer Science & Business Media, 2011, vol. 54.

[11] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization", in *Security and Privacy (SP), 2013 IEEE Symposium on*, IEEE, 2013, pp. 574–588.

[12] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind", in *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, 2014, pp. 227–242.

[13] K. A. Torkura, M. I. Sukmana, and C. Meinel, "Integrating continuous security assessments in microservices and cloud native applications", in *Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017.

[14] ——, "Cavas: Neutralizing application and container security vulnerabilities in the cloud native era (to appear)", in *14th EAI International Conference on Security and Privacy in Communication Networks*, Springer, 2018.

[15] B. Baudry, S. Allier, and M. Monperrus, "Tailored source code transformations to synthesize computationally diverse program variants", in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, 2014, pp. 149–159.

[16] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong, "Security through diversity: Leveraging virtual machine technology", *IEEE Security & Privacy*, 2009.

[17] C. Otterstad and T. Yarygina, "Low-level exploitation mitigation by diverse microservices", in *European Conference on Service-Oriented and Cloud Computing*, Springer, 2017, pp. 49–56.

[18] N. Kratzke, "About being the tortoise or the hare?-a position paper on making cloud applications too fast and furious for attackers", *arXiv preprint arXiv:1802.03565*, 2018.

[19] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow", in *Present and Ulterior Software Engineering*, Springer, 2017, pp. 195–216.

[20] S. Newman, *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[21] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce", in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 2017.

[22] J. A. Wang, H. Wang, M. Guo, and M. Xia, "Security metrics for software systems", in *Proceedings of the 47th Annual Southeast Regional Conference*, ACM, 2009.

[23] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system", *IEEE Security & Privacy*, 2006.

[24] R. Ranchal, A. Mohindra, N. Zhou, S. Kapoor, and B. Bhargava, "Hierarchical aggregation of consumer ratings for service ecosystem", in *2015 IEEE International Conference on Web Services (ICWS)*, IEEE, 2015.

[25] OWASP, *Owasp risk rating methodology*, online.

[26] M. Howard, J. Pincus, and J. M. Wing, "Measuring relative attack surfaces", in *Computer security in the 21st century*, Springer, 2005, pp. 109–137.

[27] L. Allodi and F. Massacci, "Security events and vulnerability data for cybersecurity risk estimation", *Risk Analysis*, 2017.

[28] M. Ficco, "Security event correlation approach for cloud computing", *International Journal of High Performance Computing and Networking 1*, 2013.

[29] K. A. Torkura, M. I. Sukmana, F. Cheng, and C. Meinel, "Leveraging cloud native design patterns for security-as-a-service applications", in *Smart Cloud (SmartCloud), 2017 IEEE International Conference on.*

[30] K. Czarnecki, K. Østerbye, and M. Völter, "Generative programming", in *European Conference on Object-Oriented Programming*, Springer, 2002.

[31] OWASP, *Application security risks-2017. open web application security project (owasp)*, 2017.

[32] H. Gantikow, C. Reich, M. Knahl, and N. Clarke, “Providing security in container-based hpc runtime environments”, in *International Conference on High Performance Computing*, Springer, 2016.