2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications

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*, Christoph Meinel*
*Hasso-Plattner-Institute, University of Potsdam, Potsdam, Germany
Email: *firstname.lastname@hpi.de, ‡firstname.lastname@mkayem.org

***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 introduce shared vulnerabilites, thus vulnerable to *multi-step attacks*, which are *economics-of-scale* incentives to attackers. In this paper, we address the issue of shared vulnerabilities in microservices 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, security risk-oriented software diversification is employed, guided by a defined *diversification index*. The diversification is performed at runtime, leveraging both model and template based automatic code generation techniques to automatically transform programming languages and container images of the microservices. Consequently, the microservices attack surfaces are altered thereby introducing uncertainty for attackers while reducing the *attackability* of the microservices. Our experiments demonstrate the efficiency of our solution, with an average success rate of over 70% attack surface randomization.**

***Index Terms*—Security Risk Assessment, Security Metrics, Moving Target Defense, Microservices Security, Application Container Security**

I. Introduction

Microservice Architectures (MSA) have become the *defacto* standard for agility and productivity in computing domains due to benefits such as speed and rapid scalability . 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 vulnerabilities* are not yet investigated. Most MSA are *homogenous*, composed of microservice instances built from identical *base images* thus vulnerable to *multi-step attacks* [4, 5] that exploit the re-occurrence of vulnerabilities in multiple microservices. This problem is worsened by the high number of severe vulnerabilities infecting official and community images [3, 6] on public image repositories e.g. DockerHub. Novel *container-aware*, risk analysis methodologies are imperative to address these security challenges.

A possible counter-measure against the aforementioned challenge is employment 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* [7, 8]. 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 *step 1* of Figure 1. Assuming the exploit affords him control over the API Gateway, the attacker might thereafter attack the Customer service (*step 2*), and subsequently replay the same attack against Visits and Vets services (steps 3 and 4) with the same success rates. These attacks succeed due to use of the same image for building PetClinic microservices, which amounts to inheritance of vulnerabilities across the entire application. A similar scenario of shared vulnerabilities was found in Mozilla software (Firefox and Thunderbird) in [4]. MTD tackle this challenge by deploying *security-by-diversity* tactics (*diversification*) e.g. Address Space Layout Randomization (ASLR) [9], instruction-level [10] and basic block-level [11] transformations.

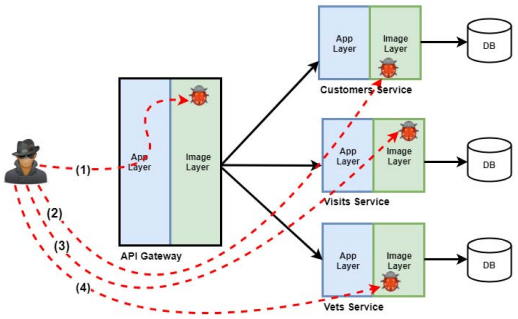

Figure 1: Successful *Multi-Step* Attacks Against PetClinic Application Due to Shared Vulnerabilities.

Contribution Therefore, we propose *MTD* mechanisms to overcome the security implications of homogeneous microservices. Our approach consists of multi-layered, risk analysis techniques that evaluate MSA and compute a *Security Risk* metric per microservice. To compute the *Security Risk*, we leverage our container-aware concept [12, 13] for vulnerability detection and prioritization. The *Security Risk* is computed using two risk models: OWASP Risk Rating Methodology (ORRM) and Common Vulnerability Scoring System (CVSS) based model. The *Security Risk* metrics are then used to derive the *diversification index* which determines the depth of

978-1-7281-1141-4/18/$31.00 ©2018 IEEE
DOI 10.1109/BDCloud.2018.00137

diversification to be implemented. Thereafter, microservices programming languages and container images are automatically diversified at runtime, using model-based [14] and template-based [15] code generation techniques. Consequently, the microservices attack surfaces are altered, introducing uncertainty for attackers while reducing the *attackability* of the microservices. The main benefit of our approach is randomization of the microservices attack surfaces to defeat anticipated attacks. To the best of our knowledge, this work is the first applying MTD techniques to MSA.

The rest of this paper is structured as follows, the next section presents related works. Section III briefly highlights the security implications of homogeneous MSA and introduces a running example. In Section IV, we present our risk models and our risk analysis methods, highlight a characterization of microservice attack surfaces and microservices vulnerability correlation matrix. The implementation details of our proposal is presented in Section V. In Section VI, we evaluate our work, while Section VII concludes the paper and provides insights on future work.

II. Related Work

Baudry et al [16] 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 [17] 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 [18], the authors characterized models to represent correlated failures due to shared code vulnerabilities and presented diversification strategies as a mitigative solution. Larsen et al. [8] compared several automated diversification techniques and provided a systematization of these techniques.

Gummaraju et al's [6] security analysis of DockerHub revealed that over 30% and 40% of official and community of Docker images respectively, are infected with highly severe vulnerabilities. Similarly, Shu et.al [3] asserted that each docker image contains an average of 180 vulnerabilities and updates are delayed for an average of hundred days. *Security Monitor* is a microservice-aware system proposed in [19] 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. [20] 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.

III. Problem Statement

This Section briefly discusses the security implications of homogeneous MSA and introduces an illustrative running example.

A. Security Risks in Homogeneous MSA

MSA consists of several autonomous, loosely coupled, polyglot components (*microservices*) operating jointly as an application [21]. The microservice architectural style supports *polyglotness* at the persistence layer and use of diverse programming languages [22] for the *business logic*. Polyglot persistence is widely practiced since it affords flexible deployment of different database types. Conversely, polyglot programming models are not favored due to complexities of managing multiple technologies. Instead, homogeneous microservices are more prevalent and prefered [23] basically due to simplicity. However, this simplicity introduces homogeneity and thus security issues, vulnerabilities that infect base images (shared libraries and packages), are directly inherited across the entire application. The security implications of shared vulnerabilities have been presented in several works e.g. Nappa et.al [4] detected 69 shared vulnerabilities between two Mozilla products: Firefox and Thunderbird, due to sharing of common codebases/libraries. The probability to compromise a microservices-based application depends on the number of microservice instances that are attacked concurrently. This possibility increases for homogenous microservices due to shared vulnerabilities, hence higher numbers of shared vulnerabilities imply higher probabilities of successful attacks i.e. *attackability* [8]. Furthermore, recent research revealed the challenge of high numbers of vulnerabilities infecting docker images [6] and late release of security patches for vulnerabilities [3]. Therefore, we aim at leveraging MTD mechanisms to reduce the attackability of MSA by minimizing shared vulnerabilities. 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 e.g. 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.

B. Running Example

We use our modified version of the Spring PetClinic microservices-based application [24] as a *running example* to discuss the security issues of homogeneous microservices. PetClinic is an application used by Pivotal [1] to demonstrate technologies and design patterns implemented with the Java Spring framework. [2] PetClinic consists of the *API Gateway*, *Visits*, *Vets* and *Customers* microservices (Figure 1). 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

[1]https://pivotal.io/
[2]https://spring.io/

vulnerabilites in closely related applications [4]. For example, an attacker might exploit an XSS vulnerability (e.g. CVE-2018-11039 [3]) in the API Gateway, to gain control of the microservice. Next, the attacker compromises the other 3 microservices using a single vulnerability e.g. *CVE-2016-7167* [4]. Lateral movement within microservices is easier once one of the *collaborating* microservices has been compromised since there is an established trust relationship. Lateral movement within microservices networks is easy once one microservice is compromised since microservices within an application trust themselves are inter-instance communications are rarely include authentication/authorization. Furthermore, the availability of most images on public image repositories e.g. DockerHub provides Open Source Intelligence (OSINT) and *economies of scale* [8] for attackers. These images can be analyzed for vulnerabilities and the knowledge gained used to orchestrate large scale attacks against MSA.

IV. 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.

A. Risk Analysis for Microservice Diversification

In Section III, we provided justification for microservice diversification i.e. to counter security issues due to homogeneous microservices. To provide a structured procedure to support security-biased diversification, we employ security metrics to design a cyber risk-based MTD mechanism. Security metrics are useful tools for risk assessments due to inherent encapsulation of risk quantification variables [25]. We aim at computing risks per microservice and thereafter employing *vulnerability prioritization* such that diversification is a function of microservice risk analysis. This approach is consistent with compliance security measures as defined by several standards e.g. ISO/IEC 27001/27002. To characterize this requirement, we introduce the notion of *Diversification Index* - D, 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 D as follows:

$$D = TM/n$$

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

[3]https://nvd.nist.gov/vuln/detail/CVE-2018-11039

[4]https://security-tracker.debian.org/tracker/CVE-2016-7167

1) Risk Analysis Using CVSS: The CVSS [26] 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 [25] as expressed below:

$$SR = \frac{\sum_{i=1}^{TV} S_i}{TV}$$

where S_i is the CVSS base score of vulnerabilities i, and TV is the total number of vulnerabilities detected in microservice m. However, averaging vulnerabilities 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* [27]. 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:

$$SR = (TV)/(TV + MV)).R + (MV/(TV + MV)).C$$

where, MV 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.

2) 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 [28]. 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 assigned considering threat vectors, possible attacks, and impacts of successful attacks.

B. 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 [29] 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

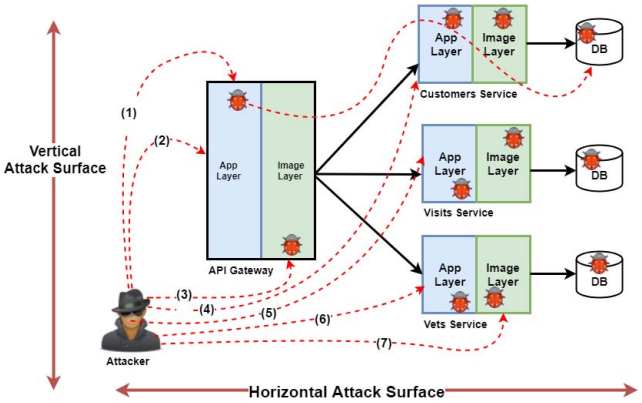

Figure 2: Horizontal and Vertical Microservice Attack Surfaces of the PetClinic Application Exploited via Multi-Step Attacks

application attackability [30]. According to OWASP [31], attack surfaces for web and REST applications include all entry and exit points e.g. APIs, HTTP headers and cookies, user interfaces e.t.c. Hence, to reduce microservices attackability, attack surfaces are identified and altered, these attack surfaces are further characterized as : *horizontal* and *vertical* attack surfaces. Furthermore, *vulnerability correlation* methods are applied to establish the relationship between vulnerabilities.

1) Horizontal Vulnerability Correlation: The objective of correlating vulnerabilities horizontally is analyze the relationship of vulnerabilities along the horizontal attack surface. Figure 2 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. Application requests and responses transversed along this layer, providing attack opportunities for attackers e.g XSS vulnerabilities. We leverage vulnerability correlation to understand the relationship between the vulnerabilities along this *plane*. In this context, vulnerability correlation process differs from the mathematical correlation, it is similar to *security event correlation* techniques [32], though rather than clustering similar attributes e.g. malicious IP addresses, we focus on Common Weakness Enumeration (CWE) Ids. CWE is a standard classification system for application weaknesses [5]. For example, CWE 89 categorizes all vulnerabilities related to *Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)* [6] and can be mapped to several CVEs e.g. *CVE-2016-6652* [7], an SQL injection vulnerability in Spring Data JPA. If this vulnerability exists in all PetClinic's microservices, an attacker could easily conduct a correlated attacks (*Attack Paths 2,4,5 and 6* of Figure 2) resulting to correlated failures and eventual application failure.

[5]https://cwe.mitre.org/index.html
[6]https://cwe.mitre.org/data/definitions/89.html
[7]https://nvd.nist.gov/vuln/detail/CVE-2016-6652

2) Vertical Vulnerability Correlation: The vertical correlation technique is similar to the horizontal correlation, however the interactions across application-image layers are analyzed. In Figure 2, 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.

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

Figure 3: Microservice Vulnerability Correlation Matrix

C. Microservices Vulnerability Correlation Matrix

Correlated vulnerabilities can be represented with correlation matrices, more specifically refered to as *microservices vulnerability correlation matrix*. Therefore, we are influenced by the definition in [18] 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 3, 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.

V. Implementation

Our MTD mechanism is implemented in software components which have been integrated into our previous work: Cloud Aware Vulnerability Assessment System (CAVAS) [12, 33]. In Figure 4, 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. The components shown in the workflow of our proposed approach (Figure 5) are explained in the following subsections.

A. 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 [12].

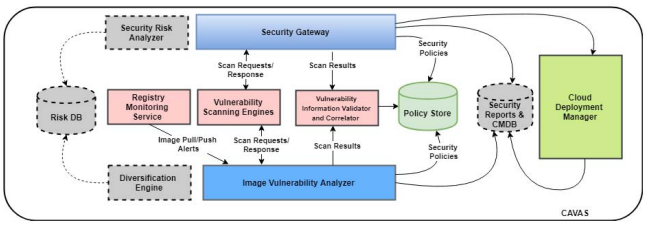

Figure 4: CAVAS Architecture Showing New Components in Dashed Lines.

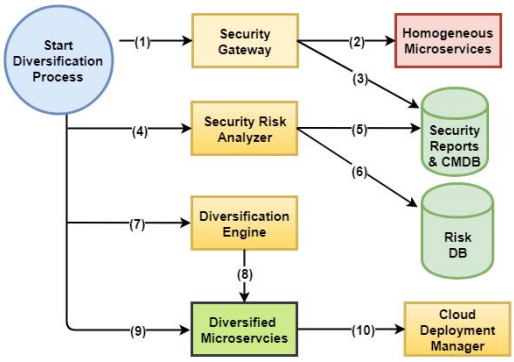

Figure 5: Workflow of the microservices MTD approach.

B. Image Vulnerability Analyzer

The Image Vulnerability Analyzer (IVA) leverages Anchore [8] to conduct static vulnerability 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 [9], it interacts with the Anchore API.

C. Application Vulnerability Analyzer

OWASP ZAP [10]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.

D. Security Risk Analyzer

The *Security Risk* - SR is computed by the Security Risk Analyzer (SRA) using our security risk models(Sections IV-A & IV-B). SRA retrieves vulnerability information from the Security Reports & CMDB for the target microservice to be analyzed (*steps 2 & 3, Figure 5*). 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.

[8]https://github.com/anchore/anchore-engine
[9]https://github.com/spotify/docker-client
[10]https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

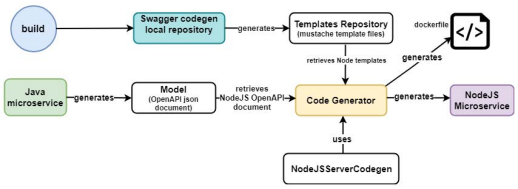

Figure 6: Swagger codegen workflow showing the transformation of a Java microservice to a NodeJS microservice.

E. Diversification Engine

The diversification engine receives the *diversification blueprint* (*step 7, Figure 5*) and transforms the target microservice-based application into the specified language (*step 8*). The transformation is done following a combination of model-driven engineering [14] and template driven automatic code generation [15] using the Swagger codegen library [11]. OpenAPI documents are used to formally model applications in json or yaml formats. In order to support a wide variety of languages, templates are used as placeholders for *boilerplate code*. Essentially, Swagger codegen generates client and server stubs in different languages/frameworks by ingesting the corresponding OpenAPI documents. Mustache templating engine [12] is used together with *dynamic reflection*, and corresponding model classes are derived from parsed Swagger documents. For example, to transform a Java microservice to NodeJS, three arguments are passed to Swagger codegen: the target language (from the list of supported languages), OpenAPI document endpoint or file location and an output directory. As illustrated in Figure 6, the Swagger codegen library then retrieves the OpenAPI documents, parses it to discover the model of the application. Similarly, the library generates mustache templates for all the supporting languages/frameworks and retains them in the local project directory. Based on the derived models, corresponding templates are retrieved from templates directory and the NodeJS configuration mapping class - *NodeJSServerCodegen* [34] employs the template for generating the NodeJS microservice. Also, Dockerfiles are generated by extracting the dependencies from maven file and constructing corresponding

[11]https://github.com/swagger-api/swagger-codegen
[12]http://mustache.github.io/

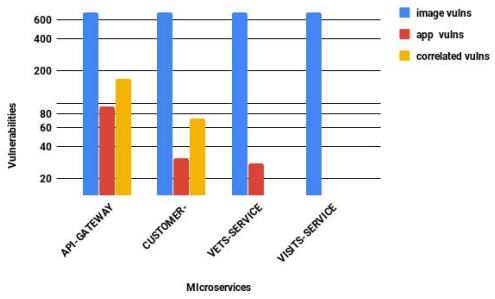

Figure 7: Vulnerability scanning results of the Homogeneous PetClinic application

NodeJS dependency file (*package.json*). Thereafter the appropriate Dockerfile is generated, and the appropriate libraries and dependencies listed in *package.json* are appended in the generated Dockerfile.

VI. Evaluation

The PetClinic application was used for our experiments, we modified the original PetClinic by adding OpenAPI support, this version is at [24]. PetClinic was deployed on a Windows 10 PC 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.

A. 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 4). 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 IV-A2). 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 [35]. This is a reasonable approach given OWASP uses ORRM for deriving the top 10 web application vulnerability scores. Also, this affords objective assignment of scores [13], which are publically verifiable. Table I is the distribution of detected vulnerabilities, while a subset of the mapping between CWE-Ids and OWASP Top 10 is on Table II. From Table II, it is obvious

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

Table I: Vulnerabilities Detected in PetClinic App-Layer

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 II: 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

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 (available at our GitHub repository [36]) is *retested* and the results are shown in Figure 8. We observe that the diversified PetClinic application layer vulnerabilities are reduced with about 53.3 %. However, the image vulnerabilities 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.

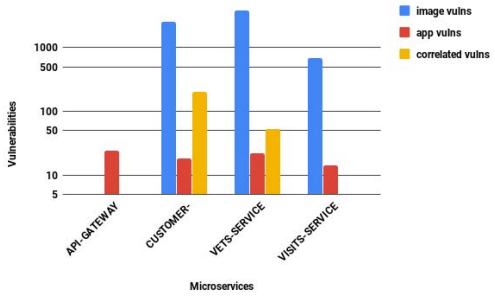

Figure 8: Vulnerabilities detected in the Diversified PetClinic Application

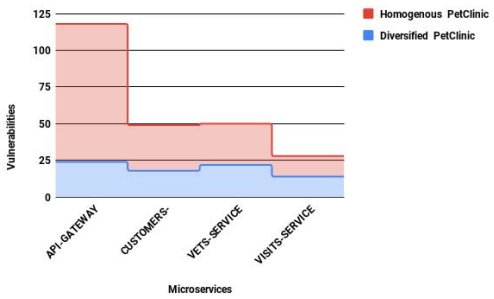

Figure 9: Horizontal Attack Surface Analysis

B. 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* [29, 31]). Figure 9 compares the horizontal app layer attack surface for both PetClinic apps. Notice a reduced attack surface in the diversified version, showing better security. Essentially, the attackability of PetClinic has been reduced, however the results for the vertical attack surface are different. This attack surface portrays attacks transversing the app-image layer (Figure 2). While there are fewer correlated vulnerabilities in the diversified API-Gateway, correlated vulnerabilities in the Customers and Vets Services have increased. This increment is due to the corresponding increase of image vulnerabilities, but the attackability due to homogeneity is reduced.

C. Discussion and Limitations

This results obtained above can be improved by combining secure coding practices in development pipelines with continuous vulnerability assessments. In our previous paper [13], we presented how CAVAS automates these processes. Detected web vulnerabilities e.g. *X-Content-Type-Options Header Missing*, can be resolved by appending appropriate headers, these suggestions are contained in CAVAS reports. Also, image vulnerabilities can be reduced e.g. *Alpine Linux* images can replace Ubuntu images as base images due to smaller footprint which equals smaller attack surfaces[37]. Furthermore, over 150 companies/projects use Swagger CodeGen in production, [14], hence the library is mature and capable of transforming huge microservice applications. Nevertheless, in this work a basic application has been used to introduce the concepts, more complex applications will be tested in future. However, our approach also has some limitations, our techniques can be applied only to OpenAPI compatible microservices. Also, Swagger Codegen currently supports transformation to about 30 programming languages/frameworks which might be a limitation in terms for possible combinations (entropy), although more languages can be added via customizations.

[14]https://github.com/swagger-api/swagger-codegen

VII. 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 an microservice-based application. Consequently, these microservices share similar vulnerabilities and are therefore vulnerable to *multi-step* 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 are employed to implement our *security-by-diversity* tactics by dynamically transforming the programming languages of the microservices to less vulnerable equivalents, consequently the container images are also transformed. This efforts effectively alters the entire composition of the applicaition, as well as the attack surfaces, thereby reducing the microservices *attackability*. Our practical evaluation demonstrates the efficiency of our security tactics, over 70 % 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. [20] 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*, 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*.

[5] O. H. Alhazmi and Y. K. Malaiya, "Application of vulnerability discovery models to major operating systems", *IEEE Transactions on Reliability*, 2008.

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

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

[8] 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.

[9] 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.

[10] 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, 2013 IEEE Symposium on*.

[11] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind", in *Security and Privacy, 2014 IEEE Symposium on*.

[12] 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.

[13] k. A. Torkura, M. I. Sukmana, and C. Meinel, "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.

[14] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry", in *Proceedings of the 33rd International Conference on Software Engineering*, ACM, 2011, pp. 633–642.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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*.

[24] K. A. Torkura, *Swaggerized-spring-petclinic*, https://github.com/maineffort/Swaggerized-Spring-PetClinic, 2017.

[25] 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.

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

[27] 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.

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

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

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

[31] OWASP, *Attack surface analysis cheat sheet*, https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet.

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

[33] 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*.

[34] Swagger, *Swagger codegen*, online. [Online]. Available: https://github.com/swagger-api/swagger-codegen/blob/master/modules/swagger-codegen/src/main/java/io/swagger/codegen/languages/NodeJSServerCodegen.java.

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

[36] K. A. Torkura, *Spring-petclinic-microservices-polyglot*, https://github.com/maineffort/Spring-petclinic-microservices-polyglot, 2018.

[37] 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.