Attack Graph Generation for Microservice Architecture

Anonymized author(s)

ABSTRACT

Microservices are increasingly dominating the field of service systems, among their many characteristics are technology heterogeneity, communicating small services, and automated deployment. Therefore, with the increase of utilizing third-party components distributed as images, the potential vulnerabilities existing in a microservice-based system increase. Based on components dependency, these vulnerabilities may lead to exposing critical assets of systems. Similar problems have been tackled in computer networks communities. In this paper, we propose the utilization of attack graphs as a part of the continuous delivery infrastructure used in microservices-based systems. To that end, we relate microservices to network nodes and automatically generate attack graphs that help practitioners to identify, analyze, and prevent plausible attack paths on their microservice-based container networks. We present a complete solution that can be easily embedded into the continuous delivery systems, and show with real-world use cases its efficiency and scalability.

KEYWORDS

Attack Graph Generation, Microservices, Containers

ACM Reference Format:
Anonymized author(s). 2019. Attack Graph Generation for Microservice Architecture. In *Proceedings of ACM SAC Conference (SAC'19).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Microservices, a recent approach to manage the complexity of modern applications, are increasingly adopted in real-world systems. The new architectural style follows the foundational principle of Unix that decomposes systems into small programs [33], each fulfills only one cohesive task and can work together using universal interfaces. Each program is a microservice that is designed, developed, tested, deployed, and scaled independently [16]. The smaller decoupled services have a positive impact on some system qualities like scalability, fault isolation, and technology heterogeneity [26]. However, other qualities like the network utilization and the security can be negatively affected [3]. Balancing the trade-off among these factors derive the decision of using microservices in industry. That said, a none-exhaustive list [1] shows a significant shift by many enterprises across different domains towards using microservice-based architecture. This shift is motivated mainly by the demanding requirements of scalability, time to market, and better optimization of development efforts. We see microservice-based systems in domains of video streaming, social networks, logistics, Internet of things [9], smart cities [23], and security-critical systems [15].

The utilization of microservices has popularized two main concepts in the software engineering community. The first is the *container-based deployment*, in which the new small services are shipped and deployed in containers [19]. As a result, the systems are deployed as networks of communicating microservices. For their lightweight and operating-system level virtualization [7], the containerization frameworks like *Docker* [10], are a high-performance alternative to hypervisors [22]. The second often-used concept in the domain of microservices development is *DevOps* [10]. *DevOps* enable practices in which full automation of the deployment process is achieved. In the course of this, end-to-end automated packaging and deployment is a vital part of microservices development. In addition to the agility and optimization brought by the two concepts, significant concerns around their impact on security [3] arise. These concerns are motivated by the increasing communication end-points among the microservices, the potentially growing number of vulnerabilities emerging from open-source DevOps tools and third-party frameworks distributed by Docker hub [17, 32], and the weaker isolation (than hypervisor-based virtualization) between the host and the container since all containers share the same kernel [7, 8]. In this paper, we tackle the problem of analyzing the security of the container networks using threat models [21]. Following the DevOps mentality, we propose an automated method that can be integrated into continuous delivery systems to generate attack graphs.

Security threat models are widely used to assess threats facing a system [21]. Not only are they appealing to the practitioners as they provide a visual presentation of possible attack paths on a system, but also to scientists, since they are well formalized (syntax and semantics [20, 24]). Such formalism enables quantitative and qualitative analysis of the risk, cost, and likelihood of the attacks, which affect the defense strategy. In computer networks, attack graphs [27, 31] are the dominant threat model to inspect the security aspects of a network. They help analysts to carefully analyze system connections and detect the most vulnerable parts of the system. An attack graph depicts the actions that an attacker may use to reach their goal. Typically, experts (e.g., red teams) manually construct attack graphs. The manual process is time-consuming, error-prone and does not address the complexity of modern infrastructure.

Previous work has dealt with automatic attack graph generation, exclusively in computer networks [18, 27, 31]. In these networks, an attacker performs multiple steps to achieve his goal, e.g., gaining privileges of a specific host. Tools that scan the vulnerabilities of a particular host are available [14], but they are not sufficient to analyze the security of an entire network, and the possible composition of various vulnerability exploitation as an attack path [31].

To the best of our knowledge, an automated attack graph generation for microservice architectures was not tackled by any previous work. To that end, in this paper, we extend the advancement made

[1] https://microservices.io/articles/whoisusingmicroservices.html

SAC'19, April 8-12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/xx.xxx/xxx_x

in the computer networks field to the domain of microservices. Therefore the contribution of this paper is as follows.

- We propose attack graphs as a new artifact of the continuous delivery systems. We present an approach, based on methods from computer networks, to automatically generate attack graphs for microservice-based architectures that are deployed as containers.
- We present the technical details of an extensible tool that implements our approach. The tool is available online [2].
- An empirical evaluation of the efficiency of our tool in generating attack graphs of real-world systems.

The structure of this paper is as follows. We introduce the preliminaries needed for this paper in Section 2. We, then, present our approach in Section 3, and its evaluation in Section 4. We discuss related work in Section 5. Lastly, conclusions and future work are discussed in Section 6.

2 BACKGROUND

We start by introducing the concept of microservices, their benefits and security implications in Subsection 2.1. In Subsection 2.2, we look into vulnerability scanners as tools to scan a single host for vulnerabilities. In Subsection 2.3, we introduce and formally describe attack graphs as methods to diagnose security weaknesses of a given system composed of multiple hosts.

2.1 Microservices

As real-world software grows in size, there is an ever-increasing need to decompose it into an organized structure to promote scaling, reuse, and readability. A software application whose modules cannot be executed independently is called a monolith. Monolithic systems are characterized by tight coupling, vertical scaling and strong dependence [16]. Service Oriented Architecture (SOA) addresses these issues by restructuring its elements into components that provide services which are used by other entities through a networking protocol [29]. However, in a typical SOA, the services are monolithic which gives rise to the concept of microservices to provide an even more fine-grained task separation [3]. The novel term "microservices" was first introduced in 2011 at an architectural workshop to propose a common term for the explorations of multiple researchers [13, 16]. In the microservices paradigm, multiple services are split into very basic units which are task oriented. According to Dragoni et al. a microservice is a cohesive, independent process interacting via messages. These microservices constitute a distributed architecture called a microservice architecture [13]. Microservice architectures benefit us with the advantage of having more heterogeneous technologies, cheaper scaling, resilience, organizational alignment, and composability [26]. However, they add additional complexity and have a wider attack surface as the need for many services to communicate with each other and third-party software increases [11, 13]. While microservices are an architectural principle, container technology has emerged in cloud computing to provide a lightweight virtualization mechanism. This technology enables microservices to be packaged and orchestrated through the Cloud [28].

[2]The link is omitted for anonymization purpose

2.2 Vulnerability Scanners

A vulnerability is a system weakness that could be exploited by a malicious actor with the help of an appropriate suite of tools. Many vulnerabilities are publicly known (CVE) and organized in databases like (NVD). CVE[3] is a list of publicly known cybersecurity vulnerabilities where each entry contains an identification number, a description, and at least one public reference. This list of publicly known vulnerabilities is organized in the NVD[4] repository that enables automation of vulnerability management, security measurement, and compliance [6]. Vulnerability scanners try to detect weaknesses by scanning a single host and generating a list of exploitable vulnerabilities [12, 14]. However, since many attacks are network-based and performed in multiple steps through a network, more sophisticated approaches are required. Therefore a combination of a vulnerability scanner and topology is seen as a promising solution to this problem in previous work [18, 31].

2.3 Attack Graphs

Attack graphs [31] are a popular way of examining network security weaknesses. They help analysts to analyze a given system and detect its vulnerable parts carefully. The definition of attack graphs may vary, but it is essentially a directed graph that consists of nodes and edges with various representations.

Seyner et al. define an attack graph as a tuple of states, transitions between the states, initial state and success states. An initial state represents the state from where the attacker starts the attack and through a chain of atomic attacks tries to reach one of the success states [31]. Ou et al. introduce the notion of a logical attack graph. A logical attack graph is a bipartite directed graph that consists of two kinds of nodes: fact nodes and derivation nodes. Each fact node is labeled with a logical statement in the form of a predicate applied to its arguments, while each derivation node is labeled with an interaction rule that is used for the derivation step. The edges in the graph represent a "depends on" relation [27]. Ingols et al. make a distinction between full, predictive and multiple-prerequisite (MP) attack graphs. Full graph is a directed acyclic graph that consists of nodes that represent hosts and edges that represent vulnerability instances. Predictive attack graphs use the same representation as full attack graphs with the only difference lying in the constraint of when the edges are added to the attack graph. These graphs are generally smaller than full graphs. MP attack is an attack graph with as contentless edges and three node type: state nodes, vulnerability instance nodes and prerequisite nodes [18].

In this paper, we define an attack graph to be a directed acyclic graph with a set of nodes and edges similar to the full graph representation of Ingols et al. [18]. As an expansion to this model, a node represents a state of a host with its current privilege. An edge represents a successful transition between two such hosts. We can think of an edge as a successful vulnerability exploitation which is initiated from a host with a required privilege to another or the same host with the newly gained privilege as a result of the vulnerability exploitation. To the best of our knowledge, attack graphs are used for networks but not microservices, potentially because there is no tool support.

[3]https://cve.mitre.org/
[4]https://nvd.nist.gov/

3 METHOD

We already defined an attack graph. Now, we look at how the existing components of attack graph generation for a computer network map into a microservice environment, we illustrate the concepts using a small example in Subsection 3.1. Then, in Subsection 3.2, we present the tools that we use to achieve this mapping and give an overview of our proposed system and its components: Topology Parser in Subsection 3.2.1, Vulnerability Parser in Subsection 3.2.2 and Attack Graph Generator Subsection 3.2.3 with the Breath-first Search graph traversal algorithm in Subsection 3.2.3.

3.1 From Network Nodes to Microservices

In our work, we adapt already existing attack graph generation methods from computer networks to the microservices ecosystem. In order to do this, we identify the corresponding components and find an equivalent replacement that can be used in a microservice architecture. In this subsection, we start first by introducing the Docker framework and its terminology. We then discuss the attack graph concepts mentioned in Subsection 2.3: that fit our use-case. We illustrate the whole idea by demonstrating a small example.

Docker is one of the most popular and used containerization frameworks currently available. In Docker, a distinction is being made between the terms *image*, *container*, and *service*. An *image* is an executable package that includes everything needed to run an application, a *container* is a runtime instance of an image, and a *service* represents a container in production. A service only runs one image, but it codifies the way that image runs, what ports it should use, how many replicas of the container should run so the service has the capacity it needs [25]. In our work, we construct attack graphs by statically analyzing the topology of the containers. Hence, we treat these terms equally.

Privileges play a central role in the generation of attack graphs. Traditionally, the privileges are modeled as a hierarchy that varies in the access level (*User, Admin*), and the access scope (virtual machine VOS, host machine OS). The exhaustive list of privileges, that are used in this paper are *None, VOS(User), VOS(Admin), OS(User) and OS(Admin)*. VOS means that the privilege is exclusive to a virtual machine, while not affecting the host machine. However in our case, unlike hosts in a network, these privileges refer to images and not virtual machines. On the other side, the keyword *OS* means that a user who has this privilege can control the host machine. Since *VOS* are isolated from host machines and their exploitation does not imply the exploitation of the host machine, they are in the lower level of the hierarchy [4]. *None* means that no privilege is obtained, *User* means only a subset of user level privileges are granted, and *Admin* grants control over the whole system.

As mentioned earlier, the *nodes* and the *edges* are the basic building blocks of an attack graph. A *node* represents a combination of a compromised Docker image and a certain privilege gained by the attacker after exploiting a vulnerability. A directed *edge* between two nodes represents an attack step from one node to another node (adjacent exploitable image with the gained privileges). Each edge is typed with the vulnerability (CVE) that could be exploited in the end node.

For attackers to exploit a given vulnerability, they need to have certain *preconditions*, i.e., the minimum required privileges to exploit [4]. Once an attacker meets these preconditions and exploits the vulnerability, he gains the privilege of the end node as a *post-condition*, and a directed edge is added between them. Both the pre- and postconditions in this work are transformed from pre- and postcondition rules manually selected and evaluated by experts in existing work [4]. The pre- and postcondition rules use the fields defined by NVD, as well as an occurrence of specific keywords from the CVEs descriptions [6].

3.1.1 Example. In order to show how the attack graph generation works in practice, we present a small example. The example is taken from the Netflix OSS Github repository. Netflix OSS example is a Spring Cloud-based microservices architecture that uses the following microservices: Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon). Displayed in Figure 1a is a subset of the example topology where each node denotes a container and each edge is a connection between two containers if one calls the other. The topology consists of an "Outside" node, "Docker daemon" node, Zuul, Eureka and other nodes. According to Netflix, Zuul is an edge service that provides dynamic routing, monitoring, resilience, and security functionalities. Eureka is a REST (Representational State Transfer) based service that is primarily used in the cloud for locating services for the purpose of load balancing and fail-over of middle-tier servers. In Figure 1b we can see a part of the corresponding attack graph, where a node is a pair of the image and its privilege, while an edge represents an atomic attack. Parts of both graphs have been intentionally omitted to reduce complexity. An example path that an attacker would take could be to first attack the Zuul container by exploiting the CVE-2016-10249 vulnerability by crafting an image file, which triggers a heap-based buffer overflow[5] and gains USER privilege. With this USER privilege, an attacker can exploit the CVE-2015-7554 vulnerability on the same container via crafted field data in an extension tag in a TIFF image[6] to gain ADMIN privilege. Once the ADMIN privilege has been obtained on Zuul, the attacker can attack the Eureka container by exploiting CVE-2017-7600 via another crafted image[7] and gain ADMIN privilege. It is important to note that this is not the only path that the attacker can take in order to have ADMIN privileges on Eureka. Another path would be to exploit the CVE-2018-1124 vulnerability via creating entries in the file-system (procfs) by starting processes, which could result in crashes or arbitrary code execution[8]. This vulnerability can be exploited by having only USER privilege on Zuul to gain ADMIN privileges of the Eureka container directly. Our attack graph generator shows both paths since it is of interest to see every possible route in which a container can be compromised.

3.2 Attack Graph Generation for Docker Networks

Figure 2 gives an overview of the attack graph generator. In the figure, the rectangles denote the main components of the system,

[5] https://nvd.nist.gov/vuln/detail/CVE-2016-10249
[6] https://nvd.nist.gov/vuln/detail/CVE-2015-7554
[7] https://nvd.nist.gov/vuln/detail/CVE-2017-7600
[8] https://nvd.nist.gov/vuln/detail/CVE-2018-1124

Figure 1: Reduced Netflix OSS example (a) Example topology graph (b) Example resulting attack graph

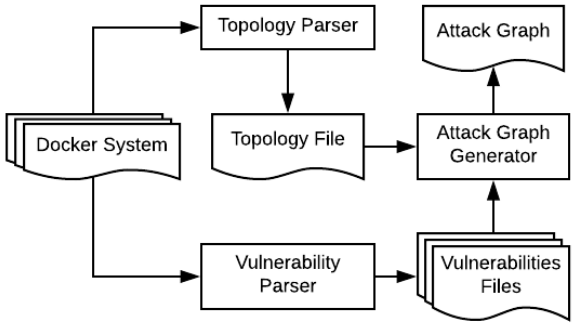

Figure 2: Overview of the Attack Graph Generator System

while the arrows describe the flow of the system and the files are the intermediate products. Our attack graph generator is composed of three main components: *Topology Parser, Vulnerability Parser, and Attack Graph Generator*. The Topology Parser reads the underlying topology of the system and converts it into to a format needed for our Attack Graph Generator, the Vulnerability Parser scans the vulnerabilities for each of the images and the Attack Graph Generator generates the attack graph from the topology and vulnerabilities files. In the following subsections, we first have a look into the system requirements, then describe each component in more details.

The generator is developed and tested for Docker 17.12.1-ce, and Docker Compose 1.19.0 [25]. Docker Compose[9] is a tool for defining the orchestration of multi-container applications. It provides a static configuration file that specifies the system containers, networks, and ports. Clair and ClairCtl [10] are used for vulnerabilities scanning. The generator is written in Python 3.6. Although we used specific versions of the tools, the pipe and filter structure of the generator can be easily extended to other versions of Docker-Compose, vulnerability scanners, and microservice architectures.

[9]https://docs.docker.com/compose/
[10]https://github.com/coreos/clair

3.2.1 Topology Parser. To generate an attack graph of a given system, we require an arrangement of its components and connections described as a system topology. The topology of Docker containers can be described either at runtime or at design time by using Docker Compose. In our case, since we are doing static attack graph analysis, we use Docker Compose to extract the topology. Docker Compose provides a file (docker-compose.yml) which is used to describe the orchestration of the services. That is to say, the file exists anyway; no further input from the security analyst is required. However different versions of docker-compose.yml, use different syntax. For example, older versions use the deprecated keyword "*link*", while newer ones use exclusively "*networks*", to denote a connection between two images. We use the keyword "*networks*" as an indicator that a connection between two images exists.

In the majority of cases, for an application to be useful, it communicates with the outside world, i.e., it has endpoints that can be used by the outer network. In Docker, this is usually done by publishing ports. This is the case in both computer networks, as well as in microservice architectures.

Another consideration that we take into account is the *privileged access* [11]. Some containers obtain certain privileges that grant them control over the Docker daemon in order to function properly. For example, a user may want to run some hardware (e.g., web-cam) or some applications that demand higher privilege levels from Docker. In Docker, this is usually done either by mounting the Docker socket or specifying the keyword "privileged" in the docker-compose.yml file. An attacker with access to these containers also has access to the Docker daemon. Once the attacker has access to the Docker daemon, he has potential access to the whole microservice system, since every container is controlled and hosted by the daemon.

3.2.2 Vulnerability Parser. In the preprocessing step, we use Clair to generate the vulnerabilities of a given image. Clair is a vulnerability scanner that inspects a Docker image and generates its vulnerabilities by providing *CVE-ID*, a description and attack vector for each vulnerability. An attack vector is an entity that describes which conditions and effects are connected to this vulnerability.

[11]http://obrown.io/2016/02/15/privileged-containers.html

We collect the fields in the attack vector which are, as defined by the National Vulnerability Database(NVD) [6], Access Vector (Local, Adjacent Network and Network), Access Complexity (Low, Medium, High), Authentication (None, Single, Multiple), Confidentiality Impact (None, Partial, Complete), Integrity Impact (None, Partial, Complete) and Availability Impact (None Partial, Complete). Since Clair does not provide a command line interface to analyze a Docker image, we use Clairctl to analyze a complete Docker image.

```
Data: topology, cont_expl, priv_acc
Result: nodes, edges
1  nodes, edges, passed_nodes = [], [], []
2  queue = Queue()
3  queue.put("outside" + "ADMIN")
4  while ! queue.isEmpty() do
5      curr_node = queue.get()
6      curr_cont = get_cont(curr_node)
7      curr_priv = get_priv(curr_node)
8      neighbours = topology[curr_cont]
9      for nb in neighbours do
10         if curr_cont == docker_host then
11             end = nb + "ADMIN"
12             create_edge(curr_node, end)
13         end
14         if nb == docker_host and priv_acc[curr_cont] then
15             end = nb + "ADMIN"
16             create_edge(curr_node, end)
17             queue.put(end)
18             passed_nodes.add(end)
19         end
20         if nb != outside and nb != docker_host then
21             precond = cont_expl[nb][precond]
22             postcond = cont_expl[nb][postcond]
23             for vul in vuls do
24                 if curr_priv > precond[vul] then
25                     end = nb + post_cond[vul]
26                     create_edge(curr_node, end_node)
27                     if end_node not in passed_nodes then
28                         queue.put(end_node)
29                         passed_nodes.add(end_node)
30                     end
31                 end
32             end
33         end
34     end
35     nodes = update_nodes()
36     edges = update_edges()
37 end
```

Algorithm 1: BFS algorithm for attack graph generation

3.2.3 Attack Graph Generator. After the topology is extracted and the vulnerabilities for each container are generated, we continue with the attack graph generation. Here, we first pre-process the vulnerabilities and convert them into sets of pre- and postconditions. In order to do this, we match the attack vectors acquired earlier from the vulnerability database and keywords of the descriptions of each vulnerability to generate attack rules. When a subset of attack vector fields and description keywords matches a given rule, we use the pre- or postcondition of that rule. An example of a precondition attack rule would be for a vulnerability to have either "gain root", "gain unrestricted, root shell access" or "obtain root" in its description and all of the impacts from the NVD attack vector [6] to be "COMPLETE" in order to gain the precondition OS(ADMIN) [4]. If more than one rule matches, we take the one with the highest privilege level for the preconditions and the lowest privilege level for the postconditions. If no rule matches, we take None as a precondition and ADMIN(OS) as a postcondition. This results in a list of container vulnerabilities with their preconditions and postconditions.

Breadth-first Search. After the preprocessing step is done, the vulnerabilities are parsed and their pre- and postconditions are extracted. Together with the topology, they are feed into a Breadth-first Search algorithm (BFS). Breadth-first Search is a popular search algorithm that traverses a graph by looking first at the neighbors of a given node, before diving deeper into the graph. Pseudo-code of our modified Breadth-first Search is given in Algorithm 1. The algorithm requires a topology, a dictionary of the exploitable vulnerabilities and a list of nodes with privileged access as an input and the output is made up of nodes and edges that make the attack graph. Topology (Subsection 3.2.1) provides the information about the connectivity between the containers, cont_expl (Subsections 3.2.2 and 3.2.3) contains information regarding which vulnerabilities can be attacked (with their pre- and postconditions), and priv_acc (Subsection 3.2.1) is the array of nodes that have high (ADMIN) permissions to the Docker daemon. The algorithm first initializes the nodes, edges, queue and the passed nodes (Lines 1, 2). Afterward, it generates the attacker node (line 3) from where the attack starts. The attacker node is a combination of the image name ("Outside") and the privilege level (ADMIN). Then into a while loop (Line 4), it iterates through every node (Line 5), checks its neighbors (Line 9) and adds the edges if the conditions are satisfied (Lines 12, 16, 26). If the neighbor was not passed, then it is added to the queue (Line 28). The algorithm terminates when the queue is empty (Line 4). Furthermore, BFS is characterized by the following properties.

- Completeness: Breadth-first Search is complete, i.e., if there is a solution, Breadth-first search will find it regardless of the kind of graph.
- Termination: This follows from the monotonicity property. Monotonicity is ensured if it is assumed that an attacker will never need to relinquish a state [5, 18, 27]. In this implementation, each edge is traversed only once, making sure that monotonicity is preserved.
- Complexity: is $O(|N|+|E|)$ where $|N|$ is the number of nodes and $|E|$ is the number of edges in the attack graph.

4 EVALUATION

Real-world microservice systems are composed of many containers that run different technologies with various degrees of connectivity among each other. This raises the need for a robust and scalable attack graph generator. In Subsection 4.1, we show our use-cases.

We then have a look at how others evaluate their systems. Finally, in Subsection 4.2, we conduct experiments to test the scalability of our system with a different number of containers and connectivity. All of the experiments were performed on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz with 8GB of RAM running Ubuntu 16.04.3 LTS.

4.1 Use Cases

Modern microservice architectures use an abundance of different technologies, number of containers, various connectivity, and number of vulnerabilities. Therefore it is of immense importance to show that an attack graph generator works well in such heterogeneous scenarios. To do this, we tested our system on real and slightly modified Github examples as described in Table 1. We intended to find and test examples that are publicly available for possible future comparison characterized by different system properties (topologies, technologies, vulnerabilities) and coming from different usage domains. We also had to take into account that an overwhelming majority of the examples publicly available are small with only one or a few containers, which made this search challenging. The resulting examples are as follows: NetflixOSS, Atsea Sample Shop App, and JavaEE demo. NetflixOSS is a microservice system provided by Netflix that is composed of 10 containers and uses many tools like Spring Cloud, Netflix Ribbon, and Netflix Eureka. Atsea Sample Shop App is an e-commerce sample web application consisting of 4 containers and uses Spring Boot, React, NGINX and PostgreSQL. JavaEE demo is a sample application for browsing movies that is composed of only two containers and uses JavaEE, React and Tomcat EE. We ran the attack graph generator and verified the resulting attack graphs of the small examples manually based on domain knowledge and under the assumption that the output from Clair, NVD attack vectors [6] and the pre- and postconditions from the work of Aksu et al. [4] are *correct*. After running the attack graph generator, the attack graphs for the Atsea Sample Shop app and the JavaEE demo are small as expected with few nodes and edges. The structure of the resulting Netflix attack graph had a nearly linear structure in which each node is connected to a small number of other nodes that form a chain of attacks. This linearity is because each container is connected to a few other containers to reduce unnecessary communication and increase encapsulation. Therefore, based on this connectivity an attacker needs to perform multiple intermediate steps to reach the target container. All of the examples terminated, there are no directed edges from containers with higher privileges to lower privileges, no duplication of nodes and no reflexive edges, which is in line with the previously mentioned monotonicity property. Additionally, we noticed that the running time of our system for each of these examples was short, and additional scalability tests are needed. The Phpmailer and Samba system is an artificial example that we use and extend in the following subsection to perform these scalability tests.

4.2 Scalability evaluation

Extensive scalability study of attack graph generators is rare in current literature, and many parameters contribute to the complexity of a comprehensive analysis. Parameters that usually vary in this sort of evaluation are the number of nodes, their connectivity and the number of vulnerabilities per container. All of these components contribute to the execution time of a given algorithm. Even though the definitions of an attack graph differ, we hope to reach a comprehensive comparison with current methods. In this case, we compare our system to existing work in computer networks by treating every container as a host machine, and any physical connection between two machines as a connection between two containers. In the following, we first look at three works and their scalability evaluation results. After this comparison, we present the scalability results of our system.

Sheyner et al. [31] test their system in both small and extended examples. The attack graph in the larger example has 5948 nodes and 68364 edges. The time needed for NuSMV to execute this configuration is 2 hours, but the model checking part took 4 minutes. The authors claim that the performance bottleneck is inside the graph generation procedure. Ingols et al. [18] tested their system on a network of 250 hosts. They afterward continued the study on a simulated network of 50000 hosts in under 4 minutes. Although this method yields better performance than the aforementioned approach, this evaluation is based on the Multiple-Prerequisite graph, which is different from ours. In addition to this, missing an explanation of how the hosts are connected, does not make it directly comparable to our method. Ou et al. [27] provide some more extended study where they test their system (MulVAL) on more examples. They mention that the asymptotic CPU time is between $O(n^2)$ and $O(n^3)$, where n is the number of nodes (hosts). The performance of the system for 1000 fully connected nodes takes more than 1000 seconds to execute.

In our scalability experiments, we use Samba [2] and Phpmailer [1] containers. We extended this example and artificially created fully connected topologies of *20, 50, 100, 500 and 1000* Samba containers to test the scalability of the system. The Phpmailer container has *181* vulnerabilities, while the Samba container has *367* vulnerabilities reported by Clair. In our tests, we measure the total execution time as well as partial times: Topology parsing time, Vulnerability preprocessing time and Breath-first Search time. The total time contains the topology parsing, the attack graph generation and some minor utility processes. The Topology parsing time is the time required to generate the graph topology. The Vulnerability preprocessing time is the time needed to convert the vulnerabilities into sets of pre- and postconditions. The Breath-first Search time is the time needed for Breadth-first Search to traverse the topology and generate the attack graph after the previous steps are done. All of the components are executed five times for each of the examples and their final time is averaged. The times are given in seconds. However, the total time does not include the vulnerability analysis by Clair. Evaluation of Clair is not in the scope of this analysis.

Table 2 shows the results of our experiments. In each of these experiments, the number of Phpmailer containers stays constant, while the number of Samba containers is increasing. This increase is done in a fully connected fashion, where a node of each container is connected to every other container. In addition, there are also two additional artificial containers: "outside" that represents the environment from where the attacker can attack and the "docker host", i.e., the Docker daemon where the containers are hosted. Therefore the number of nodes in the topology graph is the sum of: "outside", "docker host", number of Phpmailer containers and number

Name	Description	Technology Stack	No. Containers	No. vuln.	Github link
Netflix OSS	Combination of containers provided by Netflix.	Spring Cloud, Netflix Ribbon, Spring Cloud Netflix, Netflix's Eureka	10	4111	https://github.com/Oreste-Luci/netflix-oss-example
Atsea Sample Shop App	An example online store application.	Spring Boot, React, NGINX, PostgreSQL	4	120	https://github.com/dockersamples/atsea-sample-shop-app
JavaEE demo	An application for browsing movies along with other related functions.	Java EE application, React, Tomcat EE	2	149	https://github.com/dockersamples/javaee-demo
PHPMailer and Samba	An artificial example created from two separate containers. We use an augmented version for the scalability tests.	PHPMailer(email creation and transfer class for PHP), Samba(SMB/CIFS networking protocol)	2	548	https://github.com/opsxcq/exploit-CVE-2016-10033 https://github.com/opsxcq/exploit-CVE-2017-7494

Table 1: Microservice architecture examples analyzed by the attack graph generator

Statistics	example_20	example_50	example_100	example_500	example_1000
No. of Phpmailer containers	1	1	1	1	1
No. of Samba containers	20	50	100	500	1000
No. of nodes in topology	23	53	103	503	1003
No. of edges in topology	253	1378	5253	126253	502503
No. nodes in attack graph	43	103	203	1003	2003
No. edges in attack graph	863	5153	20303	501503	2003003
Topology parsing time	0.02879	0.0563	0.1241	0.7184	2.3664
Vulnerability preprocessing time	0.5377	0.9128	1.6648	6.9961	15.0639
Breadth-First Search time	0.2763	1.6524	6.5527	165.3634	767.5539
Total time	0.8429	2.6216	8.3417	173.0781	784.9843

Table 2: Scalability results with the graph characteristics and execution times in seconds.

of Samba containers. The number of edges of the topology graph is a combination of one edge ("outside"-"Phpmailer"), n edges ("docker host" to all of the containers) and n*(n+1)/2 edges of between Phpmailer and Samba containers. For example_20, the number of containers is 23 (one Phpmailer, one "outside", one "docker host" and 20 Samba containers) the number of edges in the topology graph would be 253: one outside edge, 21 Docker host edges (one toward Phpmailer and 20 toward the Samba containers) and 231 between-container edges (21*22/2=231).

Throughout the experiments, for the smaller configurations, the biggest time bottleneck is the preprocessing step. However, this step increases linearly because the container files are analyzed only once by Clair. The attack graph generation for the smaller examples is considerably less than the preprocessing time. Starting from example_500, we can notice a sharp increase in execution time to 165 seconds. For the previous example with example_100, the attack graph was generated in 6.5 seconds.

The total time of the attack graph generation procedure for 1000 fully connected hosts (784 seconds) outperforms results from OurOu et al. [27], i.e., 1000 seconds. In the Sheyners's extended example(4 hosts, 8 atomic attacks and multiple vulnerabilities) the attack graph took 2 hours to create. Our attack graph procedure even for the bigger number of hosts(1000) shows faster attack graph generation time. It, however, performs worse than the generator from Ingols et al., but that is attributed to the usage of MP attack graph which is different from ours.

In summary, we found out that our algorithm generates attack graphs efficiently, it handles a system with a *thousand* container in 13 minutes. Considering the strongly connected system in the experiment, and the high number of vulnerabilities in it, we think that the results of this experiment show a practical solution that can be used as part of the continuous delivery process of real-world systems.

5 RELATED WORK

Previous work has dealt with attack graph generation, mainly in computer networks [18, 27, 30, 31], where multiple machines are connected to each other and the Internet. One of the earlier works in attack graph generation was done by Sheyner et al. by using model checkers with goal property [31]. Model checkers use computational logic to check if a model is correct, and otherwise, they provide a counterexample. A collection of these counterexamples

form an attack graph. They state that model checkers satisfy a monotonicity property to ensure termination. However, model checkers have a computational disadvantage. Amman et al. extend this work with some simplifications and more efficient storage [30]. Ou et al. use a logical attack graph [27] and Ingols [18] et al. use a Breadth-first search algorithm in order to tackle the scalability issue. Ingols et al. discuss the redundancy Full and Predictive graphs and model an attack graph as an MP graph with content-less edges and 3 types of nodes. They use Breath-first search technique to generate the attack graph. This approach provides faster results in comparison to using model checkers. An MP graph of 8901 nodes and 23315 edges is constructed in 0.5 seconds. Aksu et al. build on top of Ingols's system and evaluate a set of rule pre- and postconditions in generating attacks. They define a specific test of pre- and postcondition rules and test their correctness. In their evaluation, they use a machine learning approach [4].

Containers and microservice architectures, despite their ever-growing popularity, have shown somewhat bigger security risks, mostly because of their need of connectivity and a lesser degree of encapsulation [11, 13]. To the best of our knowledge, no previous work that has been done in the area of attack graph generation for Docker containers. Similar to computer networks, microservice architectures have a container topology and tools for analysis of containers. Containers in our model correspond to hosts, and a connection between hosts translates to communication between containers.

In summary, our contribution is proposing attack graph generation as part of the DevOps practices, and providing a tool-support for this idea. To that end, we extended the work from Ingols [18] and Aksu [4] in conjunction with Clair OS to generate attack graphs for microservice architectures.

6 CONCLUSIONS AND FUTURE WORK

Microservices are a promising architectural style that advocate practitioners to build systems as a group of small connected services. Although this style enables better scalability and faster deployment, the full container-based automation within this style raises many security concerns. In this paper, we proposed to use automated attack graph generation as part of the practices of developing microservice-based architectures. Attack graphs aid the developers in identifying attack paths that consist of multiple vulnerability exploitation in the deployed services. The manual construction of attack graphs is an error-prone, resource consuming activity. Hence, automating this process does not only guarantee efficient construction but also complies with the spirit of DevOps practices. We have shown that such automation, extending previous works in computer networks field, is efficient and scales to complex and big microservice-based systems.

As a future work, we plan to extend this work to support more frameworks that are used in microservices systems. We also plan to study the possible analysis of the resulting attack graphs for purposes of attack detection, and post-postmortem forensics investigations.

REFERENCES

[1] 2018. PHPMailer 5.2.18 Remote Code Execution. https://github.com/opsxcq/exploit-CVE-2016-10033. Retrieved September 4 2018.

[2] 2018. SambaCry RCE exploit for Samba 4.5.9. https://github.com/opsxcq/exploit-CVE-2017-7494. Retrieved September 4 2018.

[3] Mohsen Ahmadvand and Amjad Ibrahim. 2016. Requirements reconciliation for scalable and secure microservice (de) composition. In *Requirements Engineering Conference Workshops (REW), IEEE International*. IEEE, 68–73.

[4] M Ugur Aksu, Kemal Bicakci, M Hadi Dilek, A Murat Ozbayoglu, et al. 2018. Automated Generation Of Attack Graphs Using NVD. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, 135–142.

[5] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. 2002. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 217–224.

[6] Harold Booth, Doug Rike, and Gregory A Witte. 2013. *The National Vulnerability Database (NVD): Overview.* Technical Report.

[7] James Bottomley. [n. d.]. What is All the Container Hype?

[8] Thanh Bui. 2015. Analysis of docker security. *arXiv preprint* (2015).

[9] Björn Butzin, Frank Golatowski, and Dirk Timmermann. 2016. Microservices approach for the internet of things. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 1–6.

[10] Tomas Cerny, Michael J Donahoo, and Michal Trnka. 2018. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review* 17, 4 (2018), 29–45.

[11] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To Docker or not to Docker: A security perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62.

[12] Renaud Deraison. 1999. Nessus scanner.

[13] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.

[14] Daniel Farmer and Eugene H Spafford. 1990. The COPS security checker system. (1990).

[15] Christof Fetzer. 2016. Building critical applications using microservices. *IEEE Security & Privacy* 6 (2016), 86–89.

[16] Martin Fowler. 2015. Microservices resource guide. *Martinfowler. com. Web* 1 (2015).

[17] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. 2015. Over 30% of official images in docker hub contain high priority security vulnerabilities. In *Technical Report*. BanyanOps.

[18] Kyle Ingols, Richard Lippmann, and Keith Piwowarski. 2006. Practical attack graph generation for network defense. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 121–130.

[19] David Jaramillo, Duy V Nguyen, and Robert Smart. 2016. Leveraging microservices architecture by using Docker technology. In *SoutheastCon, 2016*. IEEE, 1–5.

[20] Somesh Jha, Oleg Sheyner, and Jeannette Wing. 2002. Two formal analyses of attack graphs. In *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*. IEEE, 49–63.

[21] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. 2014. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer science review* 13 (2014), 1–38.

[22] Nane Kratzke. 2017. About microservices, containers and their underestimated impact on network performance. *arXiv preprint arXiv:1710.04049* (2017).

[23] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. 2015. Designing a smart city internet of things platform with microservice architecture. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 25–30.

[24] Sjouke Mauw and Martijn Oostdijk. 2005. Foundations of attack trees. In *International Conference on Information Security and Cryptology*. Springer, 186–198.

[25] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

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

[27] Xinming Ou, Wayne F Boyer, and Miles A McQueen. 2006. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 336–345.

[28] Claus Pahl and Pooyan Jamshidi. 2016. Microservices: A Systematic Mapping Study. In *CLOSER (1)*. 137–146.

[29] Mike P Papazoglou. 2003. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. IEEE, 3–12.

[30] Ronald W Ritchey and Paul Ammann. 2000. Using model checking to analyze network vulnerabilities. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 156–165.

[31] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. 2002. Automated generation and analysis of attack graphs. In *null*. IEEE, 273.

[32] Rui Shu, Xiaohui Gu, and William Enck. 2017. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 269–280.

[33] Eberhard Wolff. 2016. *Microservices: flexible software architecture.* Addison-Wesley Professional.