Attack Graph Generation for Microservice Architecture

Anonymized author(s)

ABSTRACT

Microservices, which are typically technologically heterogenous and can be deployed automatically, are increasingly dominating service systems. However, with increased utilization of third-party components distributed as images, the potential vulnerabilities in microservice-based systems increase. Based on component dependency, such vulnerabilities can lead to exposing a system's critical assets. Similar problems have been addressed by the computer networks community. In this paper, we propose utilizing attack graphs in the continuous delivery infrastructure of microservices-based systems. To that end, we relate microservices to network nodes and automatically generate attack graphs that help practitioners identify, analyze, and prevent plausible attack paths in their microservice-based container networks. We present a complete solution that can be easily embedded in continuous delivery systems and demonstrate its efficiency and scalability based on real-world use cases.

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, 9 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Microservices, a recent approach to managing the complexity of modern applications, are increasingly being adopted in real-world systems. Microservice architectures follow the fundamental principle of Unix, i.e., systems are decomposed into small programs [33], each performing a single cohesive task. However, such programs can work together via universal interfaces, where each program is a microservice that is designed, developed, tested, deployed, and scaled independently [16]. Smaller decoupled services have a positive impact on some system qualities, such as scalability, fault isolation, and technology heterogeneity [26]; however, other qualities, such as network utilization and security, can be affected negatively [3]. The decision to use microservices in industrial applications must consider the tradeoffs among these factors. That said, a list [1] of companies from different domains that use microservice-based architecture indicates a significant shift towards their use. This shift is primarily motivated by the demanding requirements of scalability, time to market, and improving development optimization. Microservice-based systems can be seen in various domains, such as video streaming, social networks, logistics, the 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 *container-based deployment*, where new small services are shipped and deployed in containers [19]. As a result, such systems are deployed as networks of communicating microservices. Due to their lightweight and operating-system level virtualization [7], containerization frameworks, such as *Docker* [10], have become a high-performance alternative to hypervisors [22]. The second concept in microservices development is *DevOps* [10], which enable practices that can fully automate the deployment process. Here, end-to-end automated packaging and deployment is a vital component of microservice development. In addition to the agility and optimization realized by these two concepts, significant concerns have been raised about their security [3]. These concerns are motivated by the increasing number of communication endpoints among microservices, the potentially increasing number of vulnerabilities emerging from open-source DevOps tools and third-party frameworks distributed by Docker hub [17, 32], and weaker isolation (compared to hypervisor-based virtualization) between hosts and containers because all containers share the same kernel [7, 8]. In this paper, we address the problem of analyzing the security of 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 to a system [21]. They appeal to practitioners because they provide visual presentations of possible attack paths in a system. They also appeal to scientists, because they are well formalized (syntax and semantics) [20, 24]. Such formalism enables quantitative and qualitative analysis of the risk, cost, and likelihood of attacks, which affect defense strategies. In computer networks, attack graphs [27, 31] are the dominant threat model used to inspect the security aspects of a network. They help analysts carefully analyze system connections and detect the most vulnerable parts of a system. An attack graph depicts the actions an attacker may use to reach their goal. Typically, experts (e.g., red teams) construct attack graphs manually; however, this manual process is time-consuming, error-prone, and does not address the complexity of modern infrastructures.

Previous studies have dealt with automatic attack graph generation for computer networks [18, 27, 31]. In these networks, an attacker performs multiple steps to achieve his goal, e.g., gaining the privileges of a specific host. Tools that scan the vulnerabilities of a particular host are available [14]; however, they are insufficient 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, automated attack graph generation for microservice architectures has not been examined by previous studies. To that end, we extend advancements made in the

[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

computer networks field to the microservice domain. The primary contributions of this paper are summarized as follows.

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

The remainder of this paper is organized as follows. We introduce preliminaries in Section 2. Then, we present the proposed approach in Section 3. An evaluation of the proposed approach is given in Section 4. We discuss related work in Section 5. Conclusions and suggestions for future work are discussed in Section 6.

2 BACKGROUND

We introduce 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 the security weaknesses of a given system comprising multiple hosts.

2.1 Microservices

As real-world software increases in size, there is an increasing need to decompose software into an organized structure to promote scalability, reusability, and readability. A software application with modules that cannot be executed independently is referred to as a monolith. Monolithic systems are characterized by tight coupling, vertical scaling and strong dependence [16]. The Service Oriented Architecture (SOA) addresses these issues by restructuring its elements into components that provide services that are used by other entities via a networking protocol [29]. However, in a typical SOA, the services are monolithic which gives rise to the concept of microservices to provide even more fine-grained task separation [3]. The term "microservices" was first introduced in 2011 at an architectural workshop as a common term to describe the work of multiple researchers [13, 16]. In the microservices paradigm, multiple services are split into very basic task-oriented units. 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 have 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. Container technology enables microservices to be packaged and orchestrated through the Cloud [28].

2.2 Vulnerability Scanners

A vulnerability is a system weakness that can be exploited by a malicious actor with the help of an appropriate suite of tools. Many vulnerabilities are publicly known (such as those in the Common Vulnerabilities and Exposures (CVE) list) and organized in databases, such as the National Vulnerability Database (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, which enables automation of vulnerability management, security measurement, and compliance [6]. Vulnerability scanners attempt to detect weaknesses by scanning a single host and generating a list of exploitable vulnerabilities [12, 14]. However, more sophisticated approaches are required because many attacks are network-based and performed in multiple steps throughout a network. Therefore, combinations of vulnerability scanners and topologies are considered promising solutions to this problem[18, 31].

2.3 Attack Graphs

Attack graphs [31] are a popular way to examine network security weaknesses. They facilitate careful analysis of a given system and detection its vulnerable components. The definition of an attack graph may vary, however, it is essentially a directed graph comprising nodes and edges with various representations.

Seyner et al. defined an attack graph as a tuple of states, transitions between the states, an initial state and success states. An initial state represents the state from which the attacker begins an attack and through a chain of atomic attacks attempts to reach one of the success states [31]. Ou et al. introduced the notion of a logical attack graph, which is a bipartite directed graph comprising fact 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 used in the derivation step. The edges in a logical attack graph represent a "depends on" relation [27]. Ingols et al. made a distinction between full, predictive, and multiple-prerequisite (MP) attack graphs. A full graph is a directed acyclic graph comprising 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. Note that predictive graphs are generally smaller than full graphs. An MP attack is an attack graph with contentless edges, state nodes, vulnerability instance nodes, and prerequisite nodes [18].

In this paper, we define an attack graph as a directed acyclic graph with a set of nodes and edges similar to the full graph representation proposed by Ingols et al. [18]. As an expansion to this model, a node represents the state of a host with its current privilege, and an edge represents a successful transition between two

[2]The link is omitted for anonymization purpose

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

such hosts. We can consider an edge as a successful vulnerability exploitation initiated from a host with a required privilege to another or the same host with a newly gained privilege as a result of the vulnerability exploitation. To the best of our knowledge, attack graphs have been used for networks but not microservices, potentially because there is currently no existing tool support.

3 METHOD

In Subsection 3.1, we discuss how the existing components of attack graph generation for a computer network are mapped to a microservice environment, and the concepts are illustrated using a small example. Then, in Subsection 3.2, we present the tools we use to achieve this mapping and provide an overview of the proposed system and its components, i.e., the Topology Parser (Subsection 3.2.1), the Vulnerability Parser (Subsection 3.2.2), and the Attack Graph Generator (Subsection 3.2.3). In addition, the Breath-first Search (BFS) graph traversal algorithm is discussed in Subsection 3.2.3.

3.1 From Network Nodes to Microservices

In this study, we adapt existing attack graph generation methods from the computer networks field to the microservices ecosystem. To accomplish this, we identify the corresponding components and identify an equivalent replacement that can be used in a microservice architecture. In this subsection, we begin 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 overall concept by demonstrating a small example.

Docker is one of the most popular and used containerization frameworks currently available. In Docker, a distinction is made between the terms *image*, *container*, and *service*. Here, an *image* is an executable package that includes everything required 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 a single image, however, it codifies the way that image runs, what ports it should use, and how many replicas of the container should run so the service has the capacity it requires [25]. We construct attack graphs by statically analyzing the topology of the containers; therefore, 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 access scope (virtual machine VOS, host machine OS). The privileges 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. The *OS* keyword means that a user who has this privilege can control the host machine. Since *VOSs* are isolated from host machines and their exploitation does not imply exploitation of the host machine, they are at the lower level of the hierarchy [4]. *None* means that no privilege is obtained, *User* means that only a subset of user level privileges is granted, and *Admin* grants control over the whole system.

As mentioned previously, *nodes* and *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 (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 must have certain *preconditions*, i.e., the minimum privileges required to exploit [4]. Once an attacker meets these preconditions and exploits the vulnerability, he gains the privilege of the end node as a *postcondition*, and a directed edge is added between the two nodes. Both the preconditions and postconditions in this study are transformed from precondition and postcondition rules manually selected and evaluated by experts [4]. The precondition and postcondition rules use the fields defined by the NVD, as well as an occurrence of specific keywords from CVE descriptions [6].

3.1.1 Example. Here, we present a small example to demonstrate how attack graph generation works in practice. The example is taken from the Netflix OSS GitHub repository. The Netflix OSS example is a Spring Cloud-based microservice architecture that uses the following microservices: Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul), and Client Side Load Balancing (Ribbon). Figure 1a shows 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 comprises an "Outside" node and a "Docker daemon" node, as well as 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 Representational State Transfer (REST) based service primarily used in the cloud for locating services for load balancing and fail-over of middle-tier servers. Figure 1b shows 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 omitted intentionally for simplicity. An example path 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 the 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 the ADMIN privilege. Once the ADMIN privilege has been obtained on the Zuul container, the attacker can attack the Eureka container by exploiting CVE-2017-7600 via another crafted image[7] and gain the ADMIN privilege. Note that this is not the only path the attacker can take to obtain ADMIN privileges on the Eureka container. Another path would be to exploit the CVE-2018-1124 vulnerability by 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 the USER privilege on Zuul to gain the ADMIN privileges of the Eureka container directly. Our attack graph generator shows both paths because it is of interest

[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 and (b) example resulting attack graph

to identify all possible routes through which a container can be compromised.

3.2 Attack Graph Generation for Docker Networks

Figure 2 shows an overview of our attack graph generator, where the rectangles denote the main system components, the arrows indicate the flow of the system and the files are intermediate products. The proposed attack graph generator comprises three primary components, i.e., the *Topology Parser*, the *Vulnerability Parser*, and the *Attack Graph Generator*. The Topology Parser reads the underlying topology of the system and converts it to a format required by the Attack Graph Generator. The Vulnerability Parser scans the vulnerabilities for each image, and the Attack Graph Generator generates the attack graph from the topology and vulnerabilities files. In the following, we first examine the system requirements, and then describe each component in greater detail.

The proposed generator was 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. Docker Compose provides a static configuration file that specifies the system containers, networks, and ports. Note that Clair and ClairCtl [10] were used for vulnerability scanning. The generator was written in Python 3.6. Although we used specific versions of these tools, the pipe and filter structure of the generator can be easily extended to other versions of Docker-Compose, vulnerability scanners, and microservice architectures.

3.2.1 Topology Parser. To generate an attack graph for a given system, we must arrange its components and connections as a system topology. The topology of Docker containers can be described at runtime or design time using Docker Compose. In our case, we are performing a static attack graph analysis; thus, we used Docker Compose to extract the topology. Docker Compose provides a file (docker-compose.yml) that is used to describe the orchestration of the services. In other words, this file already exists; therefore, no

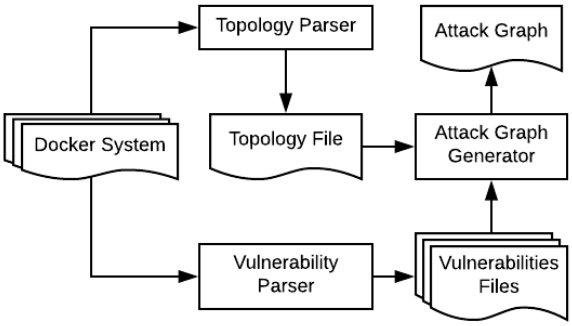

Figure 2: Overview of the proposed attach graph generator system

further input is required from a security analyst. However, different versions of the docker-compose.yml file use different syntax. For example, older versions use the deprecated keyword "*link*," while newer versions exclusively use "*networks*" to denote a connection between two images. Here, we use the keyword "*networks*" to indicate a connection between two images.

For an application to be useful in most cases, it communicates with the outside world, i.e., it has endpoints that can be used by an outer network. In Docker, this is typically accomplished by publishing ports. This is the case for both computer networks and microservice architectures.

Another consideration is *privileged access*[11]. In order to function properly, some containers obtain certain privileges that grant them control over the Docker daemon. For example, a user may want to run hardware (e.g., a webcam) or applications that demand higher privilege levels from Docker. In Docker, this is typically achieved either by mounting the Docker socket or specifying the "privileged" keyword in the docker-compose.yml file. Here, 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

[9] https://docs.docker.com/compose/
[10] https://github.com/coreos/clair
[11] http://obrown.io/2016/02/15/privileged-containers.html

access to the entire microservice system because each container is controlled and hosted by the daemon.

3.2.2 Vulnerability Parser. In the preprocessing step, we use Clair to generate the vulnerabilities for a given image. Clair is a vulnerability scanner that inspects a Docker image and generates its vulnerabilities by providing a *CVE-ID*, a description and an attack vector for each vulnerability. An attack vector is an entity that describes which conditions and effects are connected to the given vulnerability. We collect the fields in the attack vector as defined by the NVD [6], i.e., Access Vector (Local, Adjacent Network, 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.

3.2.3 Attack Graph Generator. After the topology is extracted and the vulnerabilities for each container are generated, we proceed to attack graph generation. Here, we first preprocess the vulnerabilities and convert them to sets of preconditions and postconditions. To achieve this, we match the previously acquired attack vectors 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 precondition or postcondition of that rule. An example precondition attack rule would be for a vulnerability to have "gain root," "gain unrestricted, root shell access" or "obtain root" in its description and the impacts from the NVD attack vector [6] to be "COMPLETE" to obtain the OS(ADMIN) precondition [4]. If more than one rule matches, we take the rule with the highest privilege level for preconditions and the lowest privilege level for postconditions. If no rule matches, we take None as the precondition and ADMIN(OS) as the postcondition. This results in a list of container vulnerabilities with their preconditions and postconditions.

Breadth-first Search. After preprocessing, the vulnerabilities are parsed and their preconditions and postconditions are extracted. Together with the topology, they are fed into a BFS algorithm. BFS is a popular search algorithm that traverses a graph by first looking at the neighbors of a given node before diving deeper into the graph. The pseudocode for our modified BFS algorithm is given in Algorithm 1. This algorithm requires a topology, a dictionary of the exploitable vulnerabilities, and a list of nodes with privileged access as input. The output comprises nodes and edges that form the attack graph. In Algorithm 1, "topology" (Subsection 3.2.1) provides information about the connectivity between containers, "cont_expl" (Subsections 3.2.2 and 3.2.3) contains information about which vulnerabilities can be attacked (with their preconditions and postconditions), and "priv_acc" (Subsection 3.2.1) is an array of nodes with high (i.e., ADMIN) permissions to the Docker daemon. First, the algorithm initializes the nodes, edges, queue, and passed nodes (lines 1 and 2). Then, it generates the attacker node (line 3) as the node where the attack begins. The attacker node is a combination of the image name ("outside") and the privilege level (ADMIN). Then, in a while loop (line 4), the algorithm iterates through each node (line 5), checks the given node's neighbors (line 9), and adds the edges if the conditions are satisfied (lines 12, 16 and 26). If a 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: BFS is complete, i.e., if there is a solution, BFS will find it regardless of the graph type.
- 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, which ensures that monotonicity is preserved.

```
   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

- Complexity: The algorithm's 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 comprise 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. We demonstrate use cases in Subsection 4.1. We then examine how others have evaluated their systems. In Subsection 4.2, we discuss experiments conducted to test the scalability of the proposed system with different numbers of containers and varying degrees of connectivity. Note that all experiments were performed on an Intel(R) Core(TM) i5-7200U CPU (2.50GHz) with 8 GB of RAM running Ubuntu 16.04.3 LTS.

4.1 Use Cases

Modern microservice architectures use many different technologies, different numbers of containers, various degrees of connectivity, and have different numbers of vulnerabilities. Therefore, it is critically important to demonstrate that an attack graph generator works well in such heterogeneous scenarios. Here, we tested the proposed system on real and slightly modified GitHub examples (Table 1). We employed test examples that are publicly available to facilitate potential future comparison characterized by different system properties (e.g., topologies, technologies and vulnerabilities) and different usage domains. We also had to consider the fact that an overwhelming majority of publicly-available examples are small, i.e., only one or a few containers, which made finding appropriate test examples challenging. The resulting examples are as follows: NetflixOSS, the Atsea Sample Shop App, and the JavaEE demo. NetflixOSS is a microservice system provided by Netflix comprising 10 containers and uses many tools, e.g., Spring Cloud, Netflix Ribbon, and Netflix Eureka. The Atsea Sample Shop App is an e-commerce sample web application comprising four containers and uses Spring Boot, React, NGINX, and PostgreSQL. The JavaEE demo is a sample application for browsing movies comprising only two containers and uses JavaEE, React, and Tomcat EE. We ran the attack graph generator and manually verified the resulting attack graphs for the small examples based on domain knowledge under the assumption that the output from Clair, the NVD attack vectors [6], and the preconditions and postconditions from Aksu et al. [4] are *correct*. After running the proposed attack graph generator, the attack graphs for the Atsea Sample Shop App and JavaEE demo were small as expected, containing only a few nodes and edges. The structure of the NetflixOSS attack graph demonstrated a nearly linear structure in which each node was connected to a small number of other nodes to form a chain of attacks. This linearity is due to the fact that each container is connected to only a few other containers to reduce unnecessary communication and increase encapsulation. Therefore, based on this degree of connectivity, an attacker needs to perform multiple intermediate steps to reach the target container. Note that all examples terminated, there were no directed edges from containers with higher privileges to lower privileges, no duplication of nodes, and no reflexive edges were observed, which is in line with the monotonicity property. In addition, the run time of the proposed system with each example was short, however, additional scalability tests were required. Therefore, the Phpmailer and Samba system was extended and employed as an artificial example to perform scalability tests. This is discussed in the following subsection.

4.2 Scalability evaluation

Extensive study of the scalability of attack graph generators is rare in the current literature, and many parameters contribute to the complexity of comprehensive analyses. Parameters that typically vary in this sort of evaluation include the number of nodes, their connectivity and the number of vulnerabilities per container, all of which contribute to the execution time of a given algorithm. Even though the definitions of an attack graph differ, we hope to achieve a comprehensive comparison with current methods. Here, we compared the proposed system to existing work in computer networks by treating each container as a host machine and any physical connection between two machines as a connection between two containers. In the following, we first examine three methods and their scalability evaluation results. We then present the scalability results of the proposed system.

Sheyner et al. [31] tested their system (NuSMV) using both small and extended examples. The attack graph in the larger example has 5948 nodes and 68364 edges. The time required for NuSMV to execute this configuration was two hours; however, the model checking component took four minutes. The authors claim that the performance bottleneck resides in the graph generation procedure. Ingols et al. [18] tested their system on a network of 250 hosts. They continued the study on a simulated network with 50000 hosts in under four minutes. Although their method yields better performance than NuSMV, their evaluation was based on a MP graph, which differs from our target graph. Ou et al. [27] provided a more extensive study, wherein they tested their system (MulVAL) using more examples. They state that the asymptotic CPU time was between $O(n^2)$ and $O(n^3)$, where n is the number of nodes (hosts). With 1000 fully-connected nodes, their system required more than 1000 seconds to execute.

We used Samba [2] and Phpmailer [1] containers in our scalability experiments. 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 proposed system. As reported by Clair, the Phpmailer container has *181* vulnerabilities and the Samba container has *367* vulnerabilities. In our tests, we measured the total execution time and partial times, i.e., topology parsing time, vulnerability preprocessing time, and BFS time. The total time contains topology parsing, attack graph generation, and other utility processes. Here, the topology parsing time is the time required to generate the graph topology, the vulnerability preprocessing time is the time required to convert vulnerabilities into sets of preconditions and postconditions, and the BFS time is the time required for the BFS algorithm to traverse the topology and generate the attack graph after the previous steps are complete. All components were executed five times for each example and their final time was averaged. Note that the measured times are given in seconds. However, the total time does not include the Clair vulnerability analysis, because this evaluation is beyond the scope of this analysis.

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 proposed 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
BFS 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 graph characteristics and execution times (s)

Table 2 shows the experimental results. In each experiment, the number of Phpmailer containers was constant. In contrast, the number of Samba containers increased in a fully-connected manner, where a node of each container was connected to all other containers. In addition, there were also two additional artificial containers, i.e., "outside," which represents the environment from where the attacker can attack, and the "docker host," i.e., the Docker daemon where containers are hosted. Thus, the total number of nodes in the topology graph is the sum of "outside," "docker host," the number of Phpmailer containers, and the number of Samba containers. The number of edges in the topology graph is a combination of one edge ("outside"-"Phpmailer"), n edges ("docker host" to all containers) and n*(n+1)/2 edges between the Phpmailer and Samba containers. For example_20, the number of containers is 23 (one Phpmailer container, one "outside" container, one "docker host" container, and 20 Samba containers). Thus, the number of edges in this topology graph is 253, i.e., one outside edge, 21 Docker host edges (one toward Phpmailer and 20 toward the Samba containers), and 231 between-container edges (i.e., 21*22/2=231).

Throughout the experiments, the greatest time bottleneck was the preprocessing step for the smaller configurations. However, this time increased linearly because the container files are analyzed only once by Clair. Note that the attack graph generation time for the smaller examples was considerably less than the preprocessing time. For example_500, we note a sharp increase in execution time (165 seconds) compared to the previous example (i.e., example_100), where 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) was better than the results of Ou et al. [27], i.e., 1000 seconds. In Sheyners's extended example (four hosts, eight atomic attacks and multiple vulnerabilities), the attack graph took two hours to create. In contrast, even for a greater number of hosts (1000), the proposed attack graph procedure demonstrates faster attack graph generation time. However, the proposed system performs worse than the generator proposed by Ingols et al., but that is attributed to the usage of the MP attack graph, which differs from our target graph.

In summary, we found that the proposed algorithm generates attack graphs efficiently, i.e., it handles a system with 1000 containers in 13 minutes. Considering the strongly-connected system employed in the experiment and the high number of vulnerabilities in this system, we consider that the results demonstrate that the

proposed system is a practical solution that can be used as part of the continuous delivery processes of real-world systems.

5 RELATED WORK

Previous studies have examined attack graph generation, primarily relative to computer networks [18, 27, 30, 31], where multiple machines are connected to each other and the Internet. One early study of attack graph generation was conducted by Sheyner et al. using model checkers with a goal property [31]. Model checkers use computational logic to determine if a model is correct; otherwise, if the model is incorrect, the model checkers provide a counterexample. A collection of these counterexamples form an attack graph. Sheyner et al. stated that model checkers satisfy the monotonicity property to ensure termination. However, model checkers have a computational disadvantage. Amman et al. extended this work with some simplifications and more efficient storage [30]. Ou et al. used a logical attack graph [27] and Ingols et al. [18] used BFS algorithm to tackle the scalability issue. Ingols et al. discussed the redundancy of full and predictive graphs and modeled an attack graph as an MP graph with contentless edges and three node types. They used BFS technique to generate the attack graph. This approach provides faster results compared to using model checkers. For example, with this method, an MP graph with 8901 nodes and 23315 edges was constructed in 0.5 seconds. Aksu et al. conveyed a study on top of Ingols's system. They defined a specific set of precondition and postcondition rules and tested their correctness. Note that they used a machine learning approach in their evaluation [4].

Despite their increasing popularity, containers and microservice architectures have demonstrated serious security risks, primarily due to their connectivity requirements and lesser degree of encapsulation [11, 13]. To the best of our knowledge, no previous study has been conducted relative to attack graph generation for Docker containers. Similar to computer networks, microservice architectures have a container topology and tools for container analysis. Containers in our model correspond to hosts, and a connection between hosts translates to communication between containers.

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

6 CONCLUSIONS AND FUTURE WORK

Microservices are a promising architectural style that encourage practitioners to build systems as a group of small connected services. Although such architectures can realize better scalability and faster deployment, full container-based automation raises many security concerns. In this paper, we have proposed the use of automated attack graph generation relative to the development of microservice-based architectures. Attack graphs help developers identify attack paths that comprise exploitable vulnerabilities in deployed services. Manual construction of attack graphs is an error-prone, resource consuming activity; therefore, automating this process guarantees efficient construction and complies with the spirit of DevOps practices. By extending previous work in field of computer networks, we have demonstrated that such automation is efficient and scales to large and complex microservice-based systems.

In future work, we plan to extend this work to support more frameworks used in microservice systems. We also plan to study possible analysis of the resulting attack graphs for use in 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 Pièlre-Cambacédès, and Patrick Schweitzer. 2014. DAG-based attack and defense modeling: DonâĂŹt 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.