Attack Graph Generation for Microservice Architecture

Anonymized author(s)

ABSTRACT

Microservices, which are typically technologically heterogenous and can be deployed automatically, are increasingly dominating ~~the field of~~ service systems~~, among their many characteristics are technology heterogeneity, communicating small services, and automated deployment. Therefore~~. However, with ~~the increase~~increased utilization of ~~utilizing~~ third-party components distributed as images, the potential vulnerabilities ~~existing~~ in ~~a~~ microservice-based ~~system~~systems increase. Based on ~~components~~component dependency, ~~these~~such vulnerabilities ~~may~~can lead to exposing a system's critical assets ~~of systems~~. Similar problems have been ~~tackled in~~ addressed by the computer networks ~~communities.~~community. In this paper, we propose ~~the utilization of~~ utilizing attack graphs ~~as a part of~~in the continuous delivery infrastructure ~~used in~~of 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~~in their microservice-based container networks. We present a complete solution that can be easily embedded ~~into the~~in continuous delivery systems~~,~~ and ~~show with real-world use cases~~demonstrate its efficiency and scalability based on real-world use cases.

Comment [Editor1]: Remark: Do you mean “micro”? Please clarify.

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~~managing the complexity of modern applications, are increasingly being adopted in real-world systems. ~~The new architectural style follows~~Microservice architectures follow the ~~foundational~~fundamental principle of Unix ~~that decomposes~~, i.e., systems are decomposed into small programs [33], each ~~fulfills only one~~performing a single cohesive task ~~and~~. However, such programs can work together ~~using~~via universal interfaces~~. Each~~, where each program is a microservice that is designed, developed, tested, deployed, and scaled independently [16]. ~~The smaller~~Smaller decoupled services have a positive impact on some system qualities ~~like~~, such as scalability, fault isolation, and technology heterogeneity [26~~]. However~~]; however, other qualities ~~like the~~ , such as network utilization and ~~the~~ security, can be affected negatively ~~affected~~ [3]. ~~Balancing the trade-off among these factors derive the~~ The decision ~~of using~~to use microservices in ~~industry.~~industrial applications must consider the tradeoffs among these factors. That said, a ~~none-exhaustive~~ list[1] ~~shows a significant shift by many enterprises across~~of companies from different domains ~~towards using~~that use microservice-based architecture~~.~~ indicates a significant shift towards their use. This shift is primarily motivated ~~mainly~~ by the demanding

requirements of scalability, time to market, and ~~better~~ improving development optimization ~~of development efforts. We see microservice~~. Microservice-based systems can be seen in various domains ~~of~~, such as video streaming, social networks, logistics, the Internet of ~~things~~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~~where new small services are shipped and deployed in containers [19]. As a result, ~~the~~such systems are deployed as networks of communicating microservices. ~~For~~Due to their lightweight and operating-system level virtualization [7], ~~the~~ containerization frameworks ~~like~~, such as *Docker* [10], ~~are~~have become a high-performance alternative to hypervisors [22]. The second ~~often-used~~ concept in ~~the domain of~~ microservices development is *DevOps* ~~[~~(developer operations) [10~~]. *DevOps*~~], which enable practices ~~in which full automation of~~ that can fully automate the deployment process ~~is achieved. In the course of this~~. Here, end-to-end automated packaging and deployment is a vital ~~part~~component of ~~microservices~~microservice development. In addition to the agility and optimization ~~brought~~realized by ~~the~~these two concepts, significant concerns ~~around~~have been raised about their ~~impact on~~ security [3~~] arise.~~]. These concerns are motivated by the increasing number of communication endpoints ~~points~~ among ~~the~~ microservices, the potentially ~~growing~~increasing number of vulnerabilities emerging from open-source DevOps tools and third-party frameworks distributed by Docker hub [17, 32], and ~~the~~ weaker isolation (~~than~~compared to hypervisor-based virtualization) between ~~the host~~hosts and ~~the container since~~containers because all containers share the same kernel [7, 8]. In this paper, we ~~tackle~~address 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~~to a system [21]. ~~Not only are they appealing to the~~ They appeal to practitioners ~~as~~because they provide ~~a~~ visual ~~presentation~~presentations of possible attack paths ~~on~~in a system~~, but~~. They also appeal to scientists~~, since~~ because 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.~~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 ~~to~~ carefully analyze system connections and detect the most vulnerable parts of ~~the~~a system. An attack graph depicts the actions ~~that~~ an attacker may use to reach their goal. Typically, experts (e.g., red teams) ~~manually~~

Comment [Editor2]: Remark: Please consider defining containers specific to this context.

Comment [Editor3]: Tip: Word and phrase choice: In academic writing, information is presented with accuracy and conciseness. Formal language is a hallmark of academic English. One way to ensure conciseness in expression is converting phrasal verbs to formal words. Hence, we have changed "brought" to "realized."

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

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

construct attack graphs~~. The~~ manually; however, this manual process is time-consuming, error-prone, and does not address the complexity of modern ~~infrastructure~~infrastructures.

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

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

in the computer networks field to the microservice domain~~ of microservices. Therefore the contribution~~. The primary contributions of this paper ~~is~~are summarized as follows.

• We propose attack graphs as a new artifact ~~of the~~ for 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~~the proposed approach. ~~The~~ Note that this tool is available online[2].

• ~~An~~We provide an empirical evaluation of the efficiency of ~~our~~the proposed tool ~~in~~relative to generating attack graphs ~~of~~ for real-world systems.

Comment [Editor4]: Remark: Please consider revising this to “approach” here and at all such instances for consistency.

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

2 BACKGROUND

We ~~start by introducing~~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 ~~composed of~~comprising multiple hosts.

2.1 Microservices

As real-world software ~~grows~~increases in size, there is an ~~ever-~~increasing need to decompose ~~it~~ software into an organized structure to promote ~~scaling, reuse~~scalability, reusability, and readability. A software application ~~whose~~with modules that cannot be executed independently is ~~called~~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 ~~which~~that are used by other entities ~~through~~via 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~~as a common term ~~for~~to describe the ~~explorations~~work of multiple researchers [13, 16]. In the microservices paradigm, multiple services are split into very basic ~~units which are~~ 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 ~~benefit us with the advantage of having~~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. ~~This~~Container technology enables microservices to be packaged and orchestrated through the

Cloud [28].

2.2 Vulnerability Scanners

A vulnerability is a system weakness that ~~could~~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 ~~like~~, such as the National Vulnerability Database (NVD). CVE[2] 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[3] repository ~~that~~, which enables automation of vulnerability management, security measurement, and compliance [6]. Vulnerability scanners ~~try~~attempt to detect weaknesses by scanning a single host and generating a list of exploitable vulnerabilities [12, 14]. However, ~~since~~more sophisticated approaches are required because many attacks are network-based and performed in multiple steps ~~through~~throughout a network~~,~~ ~~more sophisticated approaches are required.~~. Therefore ~~a combination~~, combinations of ~~a~~ vulnerability ~~scanner~~scanners and ~~topology is seen as a~~topologies are considered promising ~~solution~~solutions to this problem ~~in previous work~~ [18, 31].

2.3 Attack Graphs

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

Seyner et al. ~~define~~defined an attack graph as a tuple of states, transitions between the states, an initial state, and a success states. An initial state represents the state from ~~where~~which the attacker ~~starts the~~begins an attack and through a chain of atomic attacks ~~tries~~attempts to reach one of the success states [31]. Ou et al. ~~introduce~~introduced the notion of a logical attack graph~~. A logical attack graph~~, which is a bipartite directed graph that ~~consists of two kinds of nodes:~~comprising 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~~in the derivation step. The edges in ~~the~~a logical attack graph represent a "depends on" relation [27]. Ingols et al. ~~make~~made a distinction between full, predictive, and multiple-prerequisite (MP) attack graphs. ~~Full~~A full graph is a directed acyclic graph ~~that consists of~~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

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

Comment [Editor5]: Remark: How many "states" are included in this definition? Please clarify.

constraint of when the edges are added to the attack graph. ~~These~~Note that predictive graphs are generally smaller than full graphs. An 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~~as a directed acyclic graph with a set of nodes and edges similar to the full graph representation ~~of~~proposed by Ingols et al. [18]. As an expansion to this model, a node represents ~~a~~the state of a host with its current privilege~~. An~~, and an edge represents a successful transition between two such hosts. We can ~~think of~~consider 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~~a newly gained privilege as a result of the vulnerability exploitation. To the best of our knowledge, attack graphs ~~are~~have been used for networks but not microservices, potentially because there is ~~no~~currently no existing tool support.

[2]The link is omitted for anonymization purpose

3 METHOD

~~We already defined an attack graph. Now,~~In Subsection 3.1., we ~~look at~~discuss how the existing components of attack graph generation for a computer network ~~map into~~ are mapped to a microservice environment, ~~we illustrate~~ and the concepts are illustrated 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~~provide an overview of ~~our~~ the proposed system and its components~~:~~, i.e., the Topology Parser ~~in~~ (Subsection 3.2.1~~,~~), the Vulnerability Parser ~~in~~ (Subsection 3.2.2), and the Attack Graph Generator (Subsection 3.2.3). In addition, ~~with~~ the Breath-first Search (BFS) graph traversal algorithm is discussed in Subsection 3.2.3.

3.1 From Network Nodes to Microservices

In ~~our work~~this study, we adapt ~~already~~ existing attack graph generation methods from the computer networks field to the microservices ecosystem. ~~In order to do~~To accomplish this, we identify the corresponding components and ~~find~~identify an equivalent replacement that can be used in a microservice architecture. In this subsection, we ~~start first~~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 ~~whole idea~~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 ~~being~~ made between the terms *image, container,* and *service.* ~~An~~Here, an *image* is an executable package that includes everything ~~needed~~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 ~~one~~a single image~~, but~~; 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 ~~needs~~requires [25]. ~~In our work, we~~We construct attack graphs by statically analyzing the topology of the containers~~. Hence~~; 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 ~~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~~ The *OS* keyword ~~*OS*~~ means that a user who has this privilege can control the host machine. Since ~~*VOS*~~*VOSs* are isolated from host machines and their exploitation does not imply ~~the~~ exploitation of the host machine, they are ~~in~~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 ~~are~~is granted, and *Admin* grants control over the whole system.

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

3.1.1 Example. ~~In order~~Here, we present a small example to ~~show~~demonstrate how ~~the~~ attack graph generation works in practice~~, we present a small example.~~. The example is taken from the Netflix OSS ~~Github~~ GitHub repository. The Netflix OSS example is a Spring Cloud-based ~~microservices~~microservice 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~~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 ~~consists of~~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 ~~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~~shows a part of the ~~corresponding~~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 ~~omitted to reduce complexity.~~for simplicity. 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[4] 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[5] to gain the 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[6] and gain the ADMIN privilege. ~~It is important to note~~Note that this is not the only path ~~that~~ the attacker can take ~~in order~~ to ~~have~~obtain ADMIN privileges on Eureka. Another path would be to exploit the CVE-2018-1124 vulnerability ~~via~~by creating entries in the file~~-~~ system (procfs) by starting processes, which could result in crashes or arbitrary code execution[7]. 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 ~~since~~because it is of interest to ~~see every~~identify all possible ~~route in~~routes through which a container can be compromised.

3.2 Attack Graph Generation for Docker Networks

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

Comment [Editor6]: Remark: Please clarify what "them" refers to.

Comment [Editor7]: Remark: Please consider revising this to “the Zuul container”?

Comment [Editor8]: Remark: Please consider revising this to “the Eureka container”?

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

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

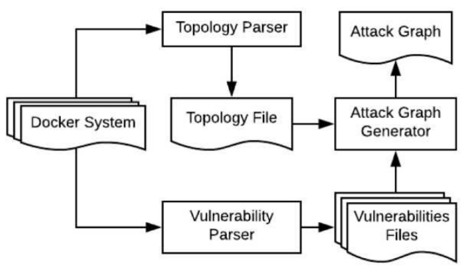

Figure 2: Overview of the ~~Attack Graph Generator System~~proposed attack graph generator system

~~while~~ the arrows ~~describe~~indicate the flow of the system, and the files are ~~the~~ intermediate products. ~~Our~~The proposed attack graph generator ~~is composed of~~comprises three ~~main~~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 ~~into~~ to a format ~~needed for our~~ required by the Attack Graph Generator~~, the~~. The Vulnerability Parser scans the vulnerabilities for each ~~of the images~~image, 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~~examine the system requirements, and then describe each component in ~~more details~~greater detail.

The proposed generator ~~is~~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. ~~It~~Docker Compose provides a static configuration file that specifies the system

Comment [Editor9]: Remark: Please verify our edit here.

Comment [Editor10]: Tip: Tense: In academic writing, the simple past tense is usually used in the Material and Methods/Experimental and Results sections of the research article.

containers, networks, and ports. Note that Clair and ClairCtl~~ ~~[10] ~~are~~were used for ~~vulnerabilities~~vulnerability scanning. The generator ~~is~~was written in Python 3.6. Although we used specific versions of ~~the~~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 ~~of~~for a given system, we ~~require an arrangement of~~must arrange 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~~ performing a static attack graph analysis; thus, we ~~use~~ used Docker-Compose to extract the topology. Docker-Compose provides a file (docker-compose.yml) ~~which~~that is used to describe the orchestration of the services. ~~That is to say, the~~ In other words, this file already exists~~ anyway;~~; therefore, no further input is required from ~~the~~ a security analyst~~ is required~~. However, different versions of the docker-compose.yml~~,~~ file use different syntax. For example, older versions use the deprecated keyword *"link~~",~~,"* while newer ~~ones use~~versions exclusively use *"networks~~",~~,"* to denote a connection between two images. ~~We~~Here, we use the keyword *"networks"* ~~as an indicator that~~to indicate a connection between two images~~ exists~~.

~~In the majority of cases, for~~ 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 ~~the~~an outer network. In Docker, this is ~~usually done~~typically accomplished by publishing ports. This is the case ~~in~~ for both computer networks~~, as well as in~~ and microservice architectures.

Another consideration ~~that we take into account~~ is ~~the~~ *privileged access*[8]~~. Some~~ In order to function properly, 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~~a webcam) or ~~some~~ applications that demand higher privilege levels from Docker. In Docker, this is ~~usually done either~~typically achieved by mounting the Docker socket or specifying the ~~keyword~~ "privileged" keyword in the docker-compose.yml file. ~~An~~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~~they have potential access to the ~~whole~~entire microservice system~~, since every~~ 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 ~~of~~for a given image. Clair is a vulnerability scanner that inspects a Docker image and generates its vulnerabilities by providing the *CVE-ID*~~, a description and~~, which describes the attack vector for each vulnerability. An attack vector is an entity that describes which conditions and effects are connected to ~~this~~the given vulnerability.

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

[10]https://github.com/coreos/clair

Comment [Editor11]: Remark: Please place footnotes appropriately here and at all such instances.

[8] 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], i.e., 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
nodes, edges, passed_nodes = [], [], []
queue = Queue()
queue.put("outside" + "ADMIN")
while ! queue.isEmpty() do
    curr_node = queue.get()
    curr_cont = get_cont(curr_node)
    curr_priv = get_priv(curr_node)
    neighbours = topology[curr_cont]
    for nb in neighbours do
        if curr_cont == docker_host then
            end = nb + "ADMIN"
            create_edge(curr_node, end)
        end
        if nb == docker_host and priv_acc[curr_cont] then
            end = nb + "ADMIN"
            create_edge(curr_node, end)
            queue.put(end)
            ~~passed_nodes.add(end)~~
        end
        if nb != outside and nb != docker_host then
            precond = cont_expl[nb][precond]
            postcond = cont_expl[nb][postcond]
            for vul in vuls do
                if curr_priv > precond[vul] then
                    end = nb + post_cond[vul]
                    create_edge(curr_node, end_node)
                    if end_node not in passed_nodes then
                        queue.put(end_node)
                        passed_nodes.add(end_node)
                    end
                end
            end
        end
    end
    nodes = update_nodes()
    edges = update_edges()
end
```

Algorithm 1: BFS algorithm for attack graph generation

Comment [Editor12]: Remark: Please check the position of Algorithm 1.

```
Data: topology, cont expl, priv acc
Result: nodes, edges
nodes, edges, passed nodes = [], [], []
queue = Queue()
queue.put("outside" + "ADMIN")
while ! queue.isEmpty() do
    curr node = queue.get()
    curr_cont = get_cont(curr_node)
    curr priv = get priv(curr node)
    neighbours   topology[curr_cont]
    for nb in neighbours do
        if curr_cont == docker_host then
            end   nb + "ADMIN"
            create_edge(curr_node, end)
        end
        if nb == docker_host and priv_acc[curr_cont] then
            end   nb + "ADMIN"
            create_edge(curr_node, end)
            queue.put(end)
            passed nodes.add(end)
        end
        if nb != outside and nb != docker host then
            precond = cont_expl[nb][precond]
            postcond = cont expl[nb][postcond]
            for vul in vuls do
                if curr_priv > precond[vul] then
                    end = nb + post cond[vul]
                    create_edge(curr_node, end_node)
                    if end_node not in passed_nodes then
                        queue.put(end_node)
                        passed nodes.add(end node)
                    end
                end
            end
        end
    end
    nodes   update_nodes()
    edges = update_edges()
end
```

3.2.3 Attack Graph Generator. After the topology is extracted and the vulnerabilities for each container are generated, we ~~continue with the~~proceed to attack graph generation. Here, we first ~~pre-process~~preprocess the vulnerabilities and convert them ~~into~~to sets of ~~pre-~~preconditions and postconditions.

~~In order to do~~To achieve this, we match the previously acquired 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-~~ precondition 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~~obtain the ~~precondition~~ OS(ADMIN) precondition [4]. If more than one rule matches, we take the ~~one~~ rule 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~~ the precondition and ADMIN(OS) as ~~a~~the postcondition. This results in a list of container vulnerabilities with their preconditions and postconditions.

Comment [Editor13]: Remark: Is this one item (separated by a comma) or two unique items? Please clarify.

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

Comment [Editor14]: Remark: This term is not capitalized in Algorithm 1. Please check.

- Completeness: ~~Breadth-first Search~~BFS is complete, i.e., if there is a solution, ~~Breadth-first search~~BFS will find it regardless of the ~~kind of~~ 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, ~~making sure~~which ensures that monotonicity is preserved.

- Complexity: The algorithm’s complexity is O~~(~~(|N | +1£ ~~|)~~|), where |N~~-~~| is the number of nodes and ~~|£ |~~|£| is the number of edges in the attack graph.

4 EVALUATION

Real-world microservice systems ~~are composed of~~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. ~~In~~We demonstrate use cases in Subsection 4.1~~, we show our use-cases.~~.

We then ~~have a look at~~examine how others ~~evaluate~~have evaluated their systems. ~~Finally, in~~In Subsection 4.2, we ~~conduct~~discuss experiments conducted to test the scalability of ~~our~~the proposed system with ~~a~~ different ~~number~~numbers of containers and varying degrees of connectivity~~. All of the~~. Note that all experiments were performed on an Intel(R) Core (TM) i5-7200U CPU ~~@~~ (2.~~50GHz~~50 GHz) with ~~8GB~~8 GB of RAM running Ubuntu 16.04.3 LTS.

Comment [Editor15]: Remark: Please check whether this edit retains your intended meaning. If not, please clarify.

4.1 Use Cases

Modern microservice architectures use ~~an abundance of~~ many different technologies, ~~number~~different numbers of containers, various degrees of connectivity, and ~~number~~have different numbers of vulnerabilities. Therefore, it is ~~of immense importance~~critically important to ~~show~~demonstrate that an attack graph generator works well in such heterogeneous scenarios. ~~To do this~~Here, we tested ~~our~~the proposed system on real and slightly modified ~~Github~~GitHub examples ~~as described in~~ (Table 1~~.~~). We ~~intended to find and~~employed test examples that are publicly available ~~for possible~~to facilitate potential future comparison characterized by different system properties (e.g., topologies, technologies, and vulnerabilities) and ~~coming from~~ different usage domains. We also had to ~~take into account~~consider the fact that an overwhelming majority of ~~the examples~~ publicly~~-~~-available examples are small ~~with~~, i.e., only one or a few containers, which made ~~this search~~ 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 ~~that is composed of~~comprising 10 containers and uses many tools ~~like~~, e.g., Spring Cloud, Netflix Ribbon, and Netflix Eureka. The Atsea Sample Shop App is an e-commerce sample web application ~~consisting of 4~~comprising four containers and uses Spring Boot, React, NGINX, and PostgreSQL. The JavaEE demo is a sample application for browsing movies ~~that is composed of~~comprising only two containers and uses JavaEE, React, and Tomcat EE. We ran the attack graph generator and manually verified the resulting attack graphs ~~of~~ for the small examples ~~manually~~ based on domain knowledge ~~and~~ under the assumption that the output from Clair, the NVD attack vectors [6~~]~~], and the ~~pre-~~preconditions and postconditions from ~~the work of~~ Aksu et al. [4] are *correct.* After running the proposed attack graph generator, the attack graphs for the Atsea Sample Shop app and ~~the~~ JavaEE demo ~~are~~were small as expected ~~with~~, containing only a few nodes and edges. The structure of the ~~resulting~~ Netflix attack graph ~~had~~demonstrated a nearly linear structure in which each node ~~is~~was connected to a small number of other nodes ~~that~~to form a chain of attacks. This linearity is ~~because~~ 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. ~~All of the~~Note that all examples terminated, there ~~are~~were no directed edges from containers with higher privileges to lower privileges, and no duplication of nodes and no reflexive edges were observed, which is in line with the ~~previously mentioned~~ monotonicity property. ~~Additionally, we noticed that the running~~In addition, the run time of ~~our~~the proposed system ~~for~~ with each ~~of these examples~~ example was short~~, and~~; however, additional scalability tests ~~are needed. The~~ were required. Therefore, the Phpmailer and Samba system ~~is~~ was extended and employed as an artificial example ~~that we use and extend in the following subsection~~ to perform ~~these~~ scalability tests. This is discussed in the following subsection.

Comment [Editor16]: Remark: Please consider revising this to “non-specific.”

4.2 Scalability evaluation

Extensive study of the scalability ~~study~~ of attack graph generators is rare in the current literature, and many parameters contribute to the complexity of ~~a~~ comprehensive ~~analysis.~~analyses. Parameters that ~~usually~~typically vary in this sort of evaluation ~~are~~include the number of nodes, their connectivity, and the number of vulnerabilities per container~~. All~~, all of ~~these components~~which contribute to the execution time of a given algorithm. Even though the definitions of an attack graph differ, we hope to ~~reach~~achieve a comprehensive comparison with current methods. ~~In this case~~Here, we ~~compare our~~compared the proposed system to existing work in computer networks by treating ~~every~~each 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~~examine three works and their scalability evaluation results. ~~After this comparison, we~~We then present the scalability results of ~~our~~the proposed system.

Comment [Editor17]: Remark: Please consider revising this to “methods.”

Sheyner et al. [31] ~~test~~tested their system ~~in~~using both small and extended examples. The attack graph in the larger example has 5948 nodes and 68364 edges. The time ~~needed~~required for NuSMV to execute this configuration ~~is 2~~was two hours~~, but~~; however, the model checking ~~part~~component took ~~4~~four minutes. The authors claim that the performance bottleneck ~~is inside~~resides in 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~~with 50000 hosts in under ~~4~~four minutes. Although ~~this~~their method yields better performance than the aforementioned approach~~, this~~, their evaluation ~~is~~was based on ~~the Multiple Prerequisite~~a MP graph, which ~~is different~~differs from ~~ours.~~our target graph. 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~~provided a more ~~extended~~extensive study~~ where~~, wherein they ~~test~~tested their system (MulVAL~~) on~~) using more examples. They ~~mention~~state that the asymptotic CPU time ~~is~~was between $O(n^2)$ and $O(n^3)$, where n is the number of nodes (hosts). ~~The performance of the system for~~With 1000 fully~~-~~-connected nodes~~ takes~~, their system required more than 1000 seconds to execute.

Comment [Editor18]: Remark: Please clarify what this is. Are you referring to the method proposed by Sheyner et al.? If so, please consider stating this.

Comment [Editor19]: Remark: Please consider naming the “aforementioned approach” for clarity.

Comment [Editor20]: Remark: This text is unclear. Please clarify.

~~In our scalability experiments, we use~~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. ~~The~~ As reported by Clair, the Phpmailer container has *181* vulnerabilities~~, while~~ and the Samba container has *367* vulnerabilities~~ reported by Clair.~~. In our tests, we ~~measure~~measured the total execution time ~~as well as~~ and partial times~~: Topology~~, i.e., topology parsing time, ~~Vulnerability~~ vulnerability preprocessing time, and ~~Breath-first Search~~BFS time. The total time contains ~~the~~ topology parsing, ~~the~~ attack graph generation, and some minor utility processes~~. The Topology~~. Here, the topology parsing time is the time required to generate the graph topology~~. The Vulnerability~~, the vulnerability preprocessing time is the time ~~needed~~required to convert ~~the~~ vulnerabilities into sets of ~~pre-~~preconditions and postconditions~~. The Breath-first Search~~, and the BFS time is the time ~~needed~~required for ~~Breadth-first Search~~ the BFS algorithm to traverse the topology and generate the attack graph after the previous steps are ~~done.~~complete. All ~~of the~~ components ~~are~~were executed five times for each ~~of the examples~~example, and their final time ~~is~~ was averaged. ~~The~~ Note that the times are given in seconds. However, the total time does not include the Clair vulnerability analysis ~~by Clair. Evaluation of Clair is not in~~because this evaluation is beyond the scope of this analysis.

Comment [Editor21]: Remark: This text is unclear. Please clarify.

Comment [Editor22]: Remark: Please verify our edit here.

Comment [Editor23]: Remark: Please specify where this has been mentioned.

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

Comment [Editor24]: Remark: Please check whether this edit retains your intended meaning.

Name	Description	Technology Stack	No. Containers	No. vuln.	~~Github~~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~~ 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. ~~ofSamba~~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~~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 ~~the~~ graph characteristics and execution times ~~in seconds.~~(s)

Comment [Editor25]: Remark: Table titles are typically positioned above tables. Please check this here and at all such instances.

Comment [Editor26]: Remark: Please verify our edit here.

~~of Samba containers.~~ The number of edges ~~of~~in 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 ~~Ph-pmailer~~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 ~~the~~this topology graph ~~would be~~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, ~~for the smaller configurations, the biggest~~the greatest time bottleneck ~~is~~ was the preprocessing step~~.~~ for the smaller configurations. However, this step increases linearly because the container files are analyzed only once by Clair. ~~The~~ Note that the attack graph generation time for the smaller examples ~~is~~was considerably less than the preprocessing time. ~~Starting from~~ For example_500, we ~~can notice~~note a sharp increase in execution time ~~to~~ (165 seconds~~. For~~) compared to the previous example ~~with~~ (i.e., example_100~~,~~), where the attack graph was generated in 6.5 seconds.

Comment [Editor27]: Remark: Do you mean the time cost/complexity increases? Please clarify.

Comment [Editor28]: Remark: We have edited this text to improve clarity/readability. Check whether your intended meaning is conveyed.

The total time of the attack graph generation procedure for 1000 fully~~-~~-connected hosts (784 seconds) ~~outperforms~~ was better than the results ~~from OurOu~~of Ou et al. [27], i.e., 1000 seconds. In ~~the~~ Sheyners's extended example~~(4~~ (four hosts, ~~8~~eight atomic attacks, and multiple vulnerabilities~~)~~), the attack graph took ~~2~~two hours to create. ~~Our attack graph procedure~~ In contrast, even for ~~the bigger~~a greater number of hosts (1000~~) shows~~), the proposed attack graph procedure demonstrates faster attack graph generation time. ~~It, however,~~ However, the proposed system performs worse than the generator ~~from~~proposed by Ingols et al., but that is attributed to the usage of the MP attack graph, which ~~is different~~differs from ~~ours~~our target graph.

In summary, we found ~~out~~ that ~~our~~the proposed algorithm generates attack graphs efficiently, i.e., it handles a system with ~~a *thousand* container~~1000 containers in 13 minutes. Considering the strongly~~-~~-connected system employed in the experiment~~,~~ and the high number of vulnerabilities in ~~it~~this system, we ~~think~~consider that the results ~~of this experiment show~~demonstrate that the proposed system is a practical solution that can be used as part of the continuous delivery ~~process~~processes of real-world systems.

5 RELATED WORK

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

Comment [Editor29]: Remark: Please clarify what you are referring to.

Comment [Editor30]: Remark: Check whether your intended meaning is conveyed.

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

~~Containers~~Despite their increasing popularity, containers and microservice architectures~~, despite their evergrowing popularity,~~ have ~~shown~~demonstrated somewhat bigger security risks, ~~mostly because of~~primarily due to their ~~need of~~ connectivity requirements and ~~a~~ lesser degree of encapsulation [11, 13]. To the best of our knowledge, no previous ~~work that~~study has been ~~done in the area of~~conducted relative to attack graph generation for Docker containers. Similar to computer networks, microservice architectures have a container topology and tools for container 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~~using attack graph generation as part of ~~the~~ DevOps practices~~,~~ and providing ~~a~~ tool-support for this ~~idea.~~concept. To that end, we have extended the work ~~from~~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 ~~advocate~~encourage practitioners to build systems as a group of small connected services. Although ~~this style enables~~such architectures can realize better scalability and faster deployment, ~~the~~ full container-based automation ~~within this style~~ raises many security concerns. In this paper, we have proposed ~~to~~ the use of automated attack graph generation ~~as part of~~ relative to the ~~practices~~development of ~~developing~~ microservice-based architectures. Attack graphs ~~aid the~~help developers ~~in identifying~~identify attack paths that ~~consist of multiple vulnerability exploitation in the~~ comprise exploitable vulnerabilities in deployed services. ~~The manual~~Manual construction of attack graphs is an error-prone, resource consuming activity~~. Hence~~; therefore, automating this process ~~does not only guarantee~~guarantees efficient construction ~~but also~~and complies with the spirit of DevOps practices. ~~We have shown that such automation.~~By extending previous ~~works~~work in the field of computer networks ~~field,~~, we have demonstrated that such automation is efficient and scales to large and complex ~~and big~~ microservice-based systems.

Comment [Editor31]: Remark: Do you refer to the monotonicity property? Indefinite article "a" infers multiple properties.

Comment [Editor32]: Remark: Check whether your intended meaning is conveyed.

Comment [Editor33]: Remark: Check whether your intended meaning is conveyed.

Comment [Editor34]: Remark: Please complete the comparison here. In addition, "size" is an inappropriate way to represent security concepts. Please consider revising this to severe, serious, harmful, etc.

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

REFERENCES

Comment [Editor35]: Remark: We have checked your references for consistency and found that there were a few incomplete references. Please check and format your references as per your target journal guidelines.

[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] Bjorn 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).~~

Formatted: Style3

Formatted: Style19

Formatted: Style15

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

Formatted: Style15

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

Formatted: Style15

[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 Pietre-Cambacedes, 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] XinmingOu, Wayne FBoyer, andMiles AMcQueen. 2006. Ascalable approachto 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 ofthe 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 ofthe 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.