Attack Graph Generation for Micro-service Architecture[*]

Extended Abstract[†]

Amjad Ibrahim
Technical University of Munich
amjad.ibrahim@tum.de

Stevica Bozhinoski
Technical University of Munich
stevica.bozhinoski@tum.de

Prof. Dr. Alexander Pretschner
Technical University of Munich
alexander.pretschner@tum.de

ABSTRACT

Microservices, in contrast to monolithic systems, provide an architecture that is modular and easily scalable. This advantage has resulted in a rapid increase in usage of microservices in recent years. Despite their rapid increase in popularity, there is a lack of work that focuses of their security aspect. Therefore, in the following paper, we present a novel Breath-First Search(BFS) based method for attack graph generation and security analysis of microservice architectures using Docker and Clair.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Attack Graph Generaton, Computer Security, Microservices

ACM Reference Format:
Amjad Ibrahim, Stevica Bozhinoski, and Prof. Dr. Alexander Pretschner. 2019. Attack Graph Generation for Micro-service Architecture: Extended Abstract. In *Proceedings of ACM SAC Conference (SAC'19).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Attack graphs are a popular way of examining security aspects of network. They help security analysts to carefully analyse a system connection and detect the most vulnerable parts of the system. An attack graph depicts the actions that an attacker uses in order to reach his goal.

Statistics. They do not offer any performance comparison between different topologies.

We first have a look into the work of other (Section 2), then present the architecure of our system (Section 3), test the scalability (Section 4) and at the end provide a conclusion (Section 5) and future work (Section 6).

[*]Produces the permission block, and copyright information
[†]The full version of the author's guide is available as `acmart.pdf` document

2 BACKGROUND

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

2.1 Microservices

As real-world software increases in size, there is an ever growing need to decompose it into an organized structure to promote scaling, reuse and readability. A software application whose modules cannot be executed independently is called a monolith. Monoliths are characterized by tight coupling, vertical scaling and strong dependence. Service Oriented Architecture(SOA) addresses these issues by restructuring its elements into components that provide services which are used by other entities through a networking protocol [20]. However, in a typical SOA, the services are monolithic which gives rise to microservices in order to provide an even more fine-grained task separation [8]. In the microservices paradigm, multiple services are split into very basic units which are task oriented. According to Dragoni et al. a microservice is a cohesive, independent process interacting via messages. These microservices constitute a distributed architecture called a microservice architecture [14]. Microservice architectures provide us with the advantage of having more heterogeneous technologies, cheaper scaling, resilience, organizational alignment, and composability among other benefits [18]. However, they add an additional complexity and have a wider attack surface as the need of many services to communicate with each other and third-party software increases [12, 14].

2.2 Vulnerability Scanners

This rise in communication makes it crucial data to be transferred and stored securely. Vulnerability scanners try to tackle this issue in computer networks by scanning a single host and generating a list of exploitable vulnerabilities [3, 13, 15]. However, since many attacks are network-based and performed in multiple steps through a network, more sophisticated approaches are required. Therefore a combination of vulnerability scanner and topology is seen as a promising solution to this problem in previous work [16, 22].

2.3 Attack Graphs

The definition of attack graphs may vary but it is essentially a directed graph that consists of nodes and edges with various representations. In this subsection, we first look at a few examples of how others define attacks graphs and at the end present the model of an attack graph that we use in this work.

Seyner et al. define an attack graph as a tuple of states, transitions between the states, initial state and success states. An initial state represents the state from where the attacker starts the attack and through a chain of atomic attacks tries to reach one of the success states [22].

Ou et al. introduce the notion of a logical attack graph. A logical attack graph is a bipartite directed graph that consists of two kinds of nodes: fact nodes and derivation nodes. Each fact node is labeled with a logical statement in the form of a predicate applied to its arguments, while each derivation node is labeled with an interaction rule that is used for the derivation step. The edges in the graph represent the "depends on" relation [19].

Ingols et al. make a distinction between full, predictive and multiple-prerequisite (MP) attack graphs. Full graph is a directed acyclic graph that consists of nodes that represent hosts and edges that represent vulnerability instances. Predictive attack graphs use the same representation as full attack graphs with the only difference lying in the constraint of when the edges are added to the attack graph. These graphs are generally smaller than full graphs. MP attack is an attack graph with as contentless edges and three node type: state nodes, vulnerability instance nodes and prerequisite nodes [16].

In our work we define attack graph to be a directed acyclic graph with a set of nodes and edges similar to the full graph representation of Ingols et al. [16]. As an expansion to this model, a node represents a state of a host with its current privilege. An edge represents a successful transition between two such hosts. We can think of an edge as a successful vulnerability exploitation which is initiated from a host with a required privilege to another or the same host with the newly gained privilege as a result of the vulnerability exploitation.

3 METHOD

Up to this point, we formally defined what an attack graph is. In this section, we first look at how the existing components of attack graph generation for computer networks map into a microservice environment and provide a small example (Subsection 3.1). We then in Subsection 3.2 refer to the third party tools that we use to achieve this translation and present an overview of our proposed system with its components: Topology Parser in Subsection 3.3, Vulnerability Parser in Subsection 3.4 and Attack Graph Parser Subsection 3.5 with the Breath-first Search graph traversal algorithm in Subsection 3.5.1.

3.1 Mapping attack graphs from network to microservice perspective

In our work, we try to translate already existing attack graph generation work from computer networks into the microservices ecosystem. In order to do this, we identify the different components and find a compatible replacement that can be used in a microservice architecture. In this subsection, we start first by shortly introducing a famous microservice framework(Docker) and some of its terminology. We then modify the attack graph concepts mentioned above for our use-case(nodes, edges, privilege levels, pre- and postconditions). In the end, we see how this translation works in practice by demonstrating a small example.

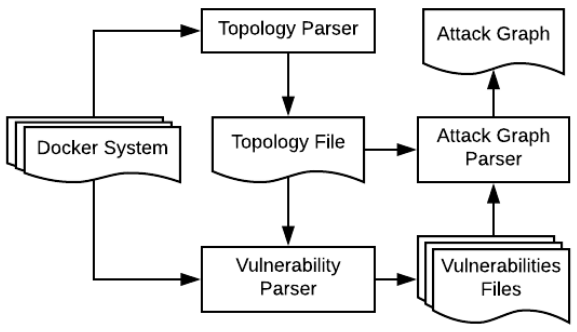

Figure 1: Our Attack Graph System. The rectangles denote the main components of the system: Topology Parser, Vulnerability Parser and Attack Graph Parser. The arrows describe the flow of the system and the files are the intermediate products.

Docker is one of the most popular and used microservice frameworks currently available. In Docker, a distinction is being made between the terms image, container and service. An image is an executable package that includes everything needed to run an application, a container is a runtime instance of an image and service represents a container in production. A service only runs one image, but it codifies the way that image runs, what ports it should use, how many replicas of the container should run so the service has the capacity it needs [17]. In our work, we treat these terms equally, since we are doing a static and not runtime attack graph analysis.

As mentioned above, nodes and edges are the basic building blocks of an attack graph. Nodes in this attack graph model are represented as a combination of docker images and their respective privilege levels, while edges are connections between node pairs accompanied by the vulnerabilities that are being exploited as descriptors. In order for an attacker to exploit a given vulnerability, certain preconditions have to be met. Once an attacker exploits this vulnerability, he gains the privilege of the target container as a postcondition and an edge is added to the attack graph. Both the pre- and postconditions in this work are transformed from pre- and postcondition rules manually selected and evaluated by experts in existing work [9]. The pre- and postcondition rules use the fields defined by NVD, as well as an occurrence of specific keywords from the CVEs descriptions [11].

Privileges play a central role in this attack graph generation. We model the privileges in a hierarchy. The privileges in ascending order 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. On the other side, The keyword OS means that the host machine has been compromised. Since VOS are hosted on some machines, and their exploitation does not imply exploitation of the host machine, they are in the lower level of hierarchy [9]. None means that no privilege is obtained, User that only a subset of user level privileges are available, while Admin grants control over the whole system.

In order to show how the attack graph generation works in practice, we present a small example. The example is taken from

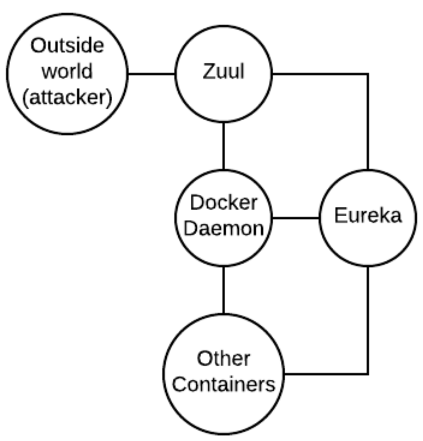

Figure 2: Example topology graph. The topology graph is a subset of a real topology graph from the Netflix OSS example. Each node denote container(plus Docker Deamon and Outside) and each edge denotes a connection between two containers.

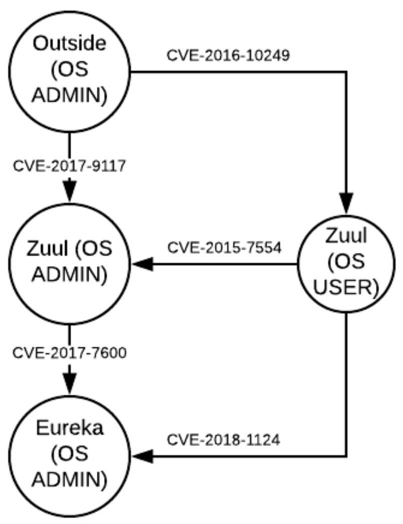

Figure 3: Example resulting attack graph. This attack graph is a subset of a real attack graph from the Netflix OSS example. Nodes correspond to a pair of container plus privilege, while edges are atomic attacks.

the Netflix OSS Github repository [7]. Displayed in Figure 2 is the topology of the example. The topology consists of "Outside" node, "Docker daemon" node and a subset of other nodes. In Figure 3 we can see a part of the resulting attack graph. Parts of both graphs have been intentionally omitted to reduce complexity. An example path that an attacker would take could be to first attack the Zuul container by exploiting the CVE-2016-10249 vulnerability and gain USER privilege. Then with this USER privilege, it can exploit the CVE-2015-7554 vulnerability on the same container to gain ADMIN privilege. Once the ADMIN privilege has been obtained on Zuul, the attacker can attack the Eureka container by exploiting CVE-2017-7600 and gain ADMIN privilege. It is important to note that this is not the only path that the attacker can take in order to have ADMIN privileges on Eureka. Another path would be to exploit the CVE-20108-1124 vulnerability while have only USER privilege on Zuul to gain directly ADMIN privileges and then attack the Eureka container. Our attack graph generator shows both paths since it is of an interest to see every possible route in which a container can be compromised.

Our attack graph generator is composed of three main components: Topology Parser, Vulnerability Parser and Attack Graph Parser(Fig. 1). The Topology Parser reads the underlying topology of the system and converts it into to a format needed for our Attack Graph Parser, the Vulnerability Parser generates the vulnerabilities for each of the images and the Attack Graph Parser generates the attack graph from the topology and vulnerabilities files.

In the following subsections, we first have a look into the system requirements, then describe each of the parsers in more detail and finally examine the characteristics of the Breath-first Search graph transversal algorithm.

3.2 Technical Details

Our system is developed for Docker 17.12.1-ce and Docker Compose 1.19.0 [17]. The code is written in Python 3.6, and we use Clair [3] and Clairctl [2] for vulnerabilities generation.

We developed this system to be used exclusively with a specific version of Docker and Docker-Compose. However, please note that the main algorithm is easily extendable to accommodate other microservice architectures if the appropriate Topology and Vulnerability parsers are provided and conform to the input of the attack graph generator.

3.3 Topology Parser

The topology of Docker containers can be described at either run-time or statically by using Docker Compose. In our case, since we are doing static attack graph analysis, we use Docker Compose as our main tool in the beginning. Docker Compose provides us with a docker-compose.yml file which is used for extraction of the topology of the system. However different versions of docker-compose.yml, use different syntax. For example, older versions use the deprecated keyword "link", while newer ones use exclusively "networks", to denote a connection between two containers. In this work, we use the keyword "networks" as an indicator that a connection between two containers exists.

However, in the majority of cases, in order for an application to be useful, it has to communicate with the outside world. This is usually done by using publishing ports. This is the case in both computer networks, as well as in microservice architectures.

Another consideration that we take into account is the privileged access. Some containers require certain privileges over the docker daemon in order to function properly. In docker, this is usually done either by mounting the docker socket or specifying the keyword "privileged" in the docker-compose.yml file. An attacker with access to these containers has also access to the Docker daemon. Once the attacker has access to the docker daemon, he has potential access to the whole microservice system, since every container is controlled and hosted by the daemon.

```
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 neigh in neighbours do
        if curr_cont == docker_host then
            end = neigh + "ADMIN"
            create_edge(curr_node, end)
        end
        if neigh == docker_host and priv_acc[curr_cont] then
            end = neigh + "ADMIN"
            create_edge(curr_node, end)
            queue.put(end)
            passed_nodes.add(end)
        end
        if neigh != outside and neigh != docker_host then
            precond = cont_expl[neigh][precond]
            postcond = cont_expl[neigh][postcond]
            for vul in vuls do
                if $curr_p riv > precond[vul]$ then
                    end = neigh + 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: Breadth-first Search algorithm for generating an attack graph.

3.4 Vulnerability Parser

In the preprocessing step, we use Clair to generate the vulnerabilities of a given container. Clair is a vulnerability scanner that inspects a docker image and generates its vulnerabilities by providing CVE-ID, description and attack vector for each vulnerability [3]. An attack vector is an entity that describes which conditions and effects are connected to this vulnerability. The fields in the attack vector as described by the National Vulnerability Database(NVD) [11] are: 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). Unfortunately, Clair does not provide with a ready to use interface to analyze a docker image. As a result, we use Clairctl [2] in order to analyze a complete docker image.

3.5 Attack Graph Parser

After the topology file is extracted and the vulnerabilities for each container are generated, we continue with the attack graph generation. Here, we first preprocess the vulnerabilities and convert them into sets of pre- and postconditions. In order to do this, we match the attack vectors acquired earlier from the vulnerability database and keywords of the descriptions of each vulnerability to generate attack rules. When a subset of attack vector fields and description keywords matches a given rule, we use the pre- or postcondition of that rule. If more than one rule matches, we take the one with the highest privilege level for the preconditions and the lowest privilege level for the postconditions. If no rule matches, we take None as a precondition and ADMIN(OS) as a postcondition. This results in a list of container vulnerabilities with their preconditions and postconditions.

3.5.1 Breadth-first Search. After the preprocessing step is done, the vulnerabilities are parsed and their pre- and postconditions are extracted. Together with the topology, they are feed into the Breadth-first Search algorithm (BFS). Breadth-first Search is a popular search algorithm that traverses a graph by looking first at the neighbors of a given node, before diving deeper into the graph. Pseudocode of our modified Breadth-first Search is given in Algorithm 1. The algorithm requires a topology and a dictionary of the exploitable vulnerabilities as an input and the output is made up of nodes and edges that make the attack graph. The algorithm first initializes the nodes, edges, queue and the passed nodes. Afterward, it generates the nodes which are a combination of the image name and the privilege level. Then into a while loop, it iterates through every node, checks its neighbors and adds the edges if the conditions are satisfied. If the neighbor was not passed, then it is added to the queue. The algorithm terminates when the queue is empty. Furthermore, BFS is characterized by the following properties:

- Completeness: Breadth-first Search is complete i.e. if there is a solution, Breadth-first search will find it regardless of the kind of graph.
- Termination: This follows from its monotonicity property. Each edge is traversed only once.
- Time 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 architectures are composed of many containers that run different technologies with various degrees of connectivity between each other. This raises the need for a robust and scalable attack graph system. In Subsection 4.1, we first show different microservice architectures on which our system was tested on. We then have a look at how others evaluate their systems. Finally in Subsection 4.2 we conduct a few experiments in order to test the

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

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

scalability of our system with a different number of containers and connectivity. All of the experiments were performed on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz with 8GB of RAM running Ubuntu 16.04.3 LTS.

4.1 Heterogenious microservice systems evaluation

Modern microservice architectures use an abundance of different technologies, number of containers, various connectivity and number of vulnerabilities. Therefore it is of immense importance to show that an attack graph generator works well in such heterogeneous scenarios. In order to do this, we tested our system on some real and slightly modified Github examples as described in Table 1 composed of different types of technologies, number of containers and vulnerabilities. The examples are as follows: NetflixOSS [7], Atsea Sample Shop App [1] and JavaEE demo [4]. NetflixOSS is a microservice system provided by Netflix that is composed of 10 containers and uses Spring Cloud, Netflix Ribbon, Netflix Eureka etc. Atsea Sample Shop App is an e-commerce sample web application composed of 4 containers and that uses Spring Boot, React, NGINX and PostgreSQL. JavaEE demo is a sample application for browsing movies that is composed of only two containers and uses JavaEE, React and Tomcat EE. We ran the attack graph generator and checked for correctness of the resulting attack graph based on domain knowledge. After running the attack graph generator, the attack graphs for the Atsea Sample Shop app and the JavaEE demo was small as expected with few nodes and edges. However, the structure of the resulting attack graphs in the Netflix case was quite linear. This linearity is because each container is connected to a few other containers to reduce unnecessary communication and increase encapsulation. Therefore based on this connectivity an attacker needs to perform multiple intermediate steps in order to reach the target container. Additionally, we noticed that the running time of our system for each of these examples was short, and additional scalability tests are needed. The Phpmailer [5] and Samba [6] system is an artificial example that we use and extend in the following subsection to perform these scalability tests.

4.2 Scalability evaluation

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

Sheyner et al. test their system in both small and extended examples. The attack graph in the larger example has 5948 nodes and 68364 edges. The time needed for NuSMV to execute this configuration is 2 hours, but the model checking part took 4 minutes. The authors claim that the performance bottleneck is inside the graph generation procedure [22].

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

Table 2: Scalability experiments with the graph characteristics and execution times. The times are given in seconds.

Ingols et al. tested their system on a network of 250 hosts. He afterward continued his study on a simulated network of 50000 hosts in under 4 minutes [16]. Although this method yields better performance than the aforementioned approach, this evaluation is based on the Multiple Prerequisite graph, which is different from ours. In addition to this, missing an explanation of how the hosts are connected, does not make it directly comparable to our method.

Ou et al. provide some more extended study where they test their system(MulVAL) on more examples. They mention that the asymptotic CPU time is between $O(n^2)$ and $O(n^3)$, where n is the number of hosts. The performance of the system for 1000 fully connected nodes takes more than 1000 seconds to execute. The authors also provide an evaluation where he MulVAL clearly outperforms the Sheyner's system [19].

In our scalability experiments we use Samba [6] and Phpmailer [5] containers which were taken from their respective Github repositories. We extended this example and artificially made fully connected topologies of 20, 50, 100, 500 and 1000 Samba containers to test the scalability of the system. The Phpmailer container has 181 vulnerabilities, while the Samba container has 367 vulnerabilities detected by Clair. In our tests, we report the total execution time as well as its components times: Topology parsing time, Vulnerability preprocessing time and Breath-first Search time. The total time contains the topology parsing, the attack graph generation and some minor utility processes. The Topology parsing time is the time required to generate the graph topology. The Vulnerability preprocessing time is the time required to convert the vulnerabilities into sets of pre- and postconditions. The Breath-first Search time is the time needed for Breadth-first Search to traverse the topology and generate the attack graph after the previous steps are done. All of the components are executed five times for each of the examples and their final time is averaged. The times are given in seconds. However, the total time does not include the vulnerability analysis by Clair. Evaluation of Clair can depend on multiple factors and it is therefore not in the scope of this analysis.

Table 2 shows the results of our experiments. In each of these experiments, the number of Phpmailer containers stays constant, while the number of Samba containers is increasing. This increase is done in a fully connected fashion, where a node of each container is connected to every other container. In addition, there are also two additional artificial containers("outside" that represents the environment from where the attacker can attack and the "docker host", i.e. the docker daemon where the containers are hosted). Therefore the number of nodes in the topology graph is the sum of: "outside", "docker host", number of Phpmailer containers and number of Samba containers. The number of edges of the topology graph is a combination of 1 edge("outside"-"Phpmailer"), n edges("docker host" to all of the containers) and a clique of the Phpmailer and Samba containers n*(n+1)/2. For example_5, the number of containers would be 8(1 Phpmailer, 1 outside, 1 docker host and 5 Samba containers) the number of edges in the topology graph would be 32: 1 outside edge, 6 docker host edges(n=6, 1 Phpmailer and 5 Sambas) and 25 clique edges(5*6/2=15).

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

5 RELATED WORK

Previous work has dealt with attack graph generation, mainly in computer networks [16, 19, 21, 22], where multiple machines are connected to each other and the Internet. In these networks, an attacker performs multiple steps to achieve his goal, i.e., gaining privileges of a specific host. Attack graphs help in analyzing this behavior. Although attack graphs are useful, constructing them manually can be a cumbersome and time-consuming process. Tools that generate vulnerabilities of a specific host are available [3, 10, 15]. However previous works state that these tools alone are not enough to analyze the vulnerability of an entire network and that these tools in addition to network topology could solve this issue. This outcome is because different hosts are connected together and influence the outcome of an attack. Therefore, some teams have been working on developing systems that generate attack graphs automatically by using different approaches.

One of the earlier works in attack graph generation was done by Sheyner et al. by using model checkers with goal property [22]. Model checkers use computational logic to check if a model is correct, and otherwise, they provide a counterexample. A collection of these counterexamples form an attack graph. They state that model checkers satisfy a monotonicity property in order to ensure termination. However, model checkers have a computational disadvantage. In the example provided, NuSMV takes 2 hours to construct the attack graph with 5948 nodes and 68364 edges [22]. As a result of this, more scalable approach was needed. Amman et al. extend this work with some simplifications and more efficient storage [21]. Ou et al. use logical attack graph [19] and Ingols [16] et al. use Breadth-first search algorithm in order to tackle the scalability issue. Ingols et al. discuss the redundancy Full and Predictive graphs and model an attack graph as an MP graph with contentless edges and 3 types of nodes. They use Breath-first search technique for generating the attack graph. This approach provides faster results in comparison to using model checkers. An MP graph of 8901 nodes and 23315 edges is constructed in 0.5 seconds. Aksu et al. build on top of Ingols's system and evaluate a set of rule pre- and postconditions in generating attacks. They define a specific test of pre- and postcondition rules and test their correctness. In their evaluation, they use a machine learning approach [9].

Containers and microservice architectures, despite their ever-growing popularity, have shown somewhat bigger security risks, mostly because of their bigger need of connectivity and a lesser degree of encapsulation [12, 14]. To the best of our knowledge, there is no work that has been done so far in the area of attack graph generation for Docker containers. Similar to computer networks, microservice architectures have a container topology and tools for analysis of containers. Containers in our model correspond to hosts, and a connection between hosts translates to a communication between containers. Therefore we extended the work from Ingols [16] and Aksu [9] in conjunction to Clair OS [3] to generate attack graphs for microservice architectures.

6 CONCLUSION

7 FUTURE WORK

REFERENCES

[1] 2018. AtSea Shop Demonstration Application. https://github.com/dockersamples/atsea-sample-shop-app.

[2] 2018. Clairctl. https://github.com/jgsqware/clairctl.

[3] 2018. CoreOS Clair. https://github.com/coreos/clair.

[4] 2018. Java EE application migration. https://github.com/dockersamples/javaee-demo.

[5] 2018. PHPMailer < 5.2.18 Remote Code Execution. https://github.com/opsxcq/exploit-CVE-2016-10033.

[6] 2018. SambaCry RCE exploit for Samba 4.5.9. https://github.com/opsxcq/exploit-CVE-2017-7494.

[7] 2018. Spring Cloud - Netflix OSS Example. https://github.com/Oreste-Luci/netflix-oss-example.

[8] 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.

[9] 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.

[10] Michael Lyle Artz. 2002. *Netspa: A network security planning architecture.* Ph.D. Dissertation. Massachusetts Institute of Technology.

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

[12] 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.

[13] Renaud Deraison. 1999. Nessus scanner.

[14] 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.

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

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

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