Attack Graph Generation for Micro-service Architecture

Stevica Bozhinoski[1], Amjad Ibrahim[2] and Prof. Dr. Alexander Pretschner[3]

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.**

I. 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.

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).

II. RELATED WORK

Previous work has dealt with attack graph generation, mainly in computer networks, where multiple machines are connected to each other and the internet. There the attacker performs multiple steps to achieve his goal, i.e. gaining privileges of the goal container. Some works use model checkers with goal property(Sheyner reference). They However model checkers as a disadvantage have a state explosion.(Ingols reference)

Others use breath first search algorithm. (Ingols reference) They model the

Others have extended this work by generating attack graphs with using rule pre- and postcondtions in generating attacks.(reference) They define a specific test of rules and test their correctness.

Containers, despite their ever-growing popularity, have shown somewhat bigger security risks, mostly because of their bigger need of connectivity and lesser degree of encapsulation(Reference). To the best of our knowledge, there is no work that has been done for attack graph generation for docker containers. They dont use Clair as well. They do not offer any performance comparison between different topologies.

III. PROPOSED METHOD

In this section we first have a look at Docker and attack graph terminology. We then present an overview of our proposed system with its components, provide a small example and at the end describe each of the components in more detail.

In Docker, distinction is being made between the terms image, container and service. Image is an executable package that includes everything needed to run an application, 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 runswhat ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. In our work we treat these terms equally, since we are doing a static and not runtime attack graph analysis.

In this work we model an attack graph as a sequence of atomic attacks. Each atomic attack represents a transition from a component(with its privilege) to either the same component with a higher privilege or to a neighbor component with a new privilege. A goal of an attacker would be to perform multiple atomic attacks to get to the desired goal container. For example, let us suppose that an attacker wants to have access to a database of a certain website. In order to reach the database, he has to pass other containers between him and the database. He does that by finding a vulnerability to exploit and give access to the container's neighbors. After a successful exploitation of multiple intermediate containers, he finally has access to the database and its functions. Even though attack graphs model the attacker scenario from an attackers perspective, they are of crucial importance in computer security. For example, a system administrator would be interested to have an overview of the attack paths that an attacker could exploit, in order to harden the security of a given enterprise system.

Atomic attacks are fundamental building blocks in our attack graph. A single atomic attack represents a successful vulnerability exploitation. A consecutive sequence of atomic attacks represents a path in an attack graph and multiple attack paths constitute an attack graph. For an atomic attack to be executed, two constraints are imposed. First, the containers have to be connected between each other, so that physical access can be ensured. Second, the attacked container should contain some vulnerability that the attacker can exploit and gain a privilege level. We model the privileges into a hierarchy. The privileges in ascending order are: None, VOS(User), VOS(Admin), OS(User) and OS(Admin). VOS means that the privilege is connected to a virtual machine, while 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.

Furthermore, in order for atomic attacks to be performed, certain preconditions have to be met. Preconditions are privilege levels that are required so that a vulnerability can be

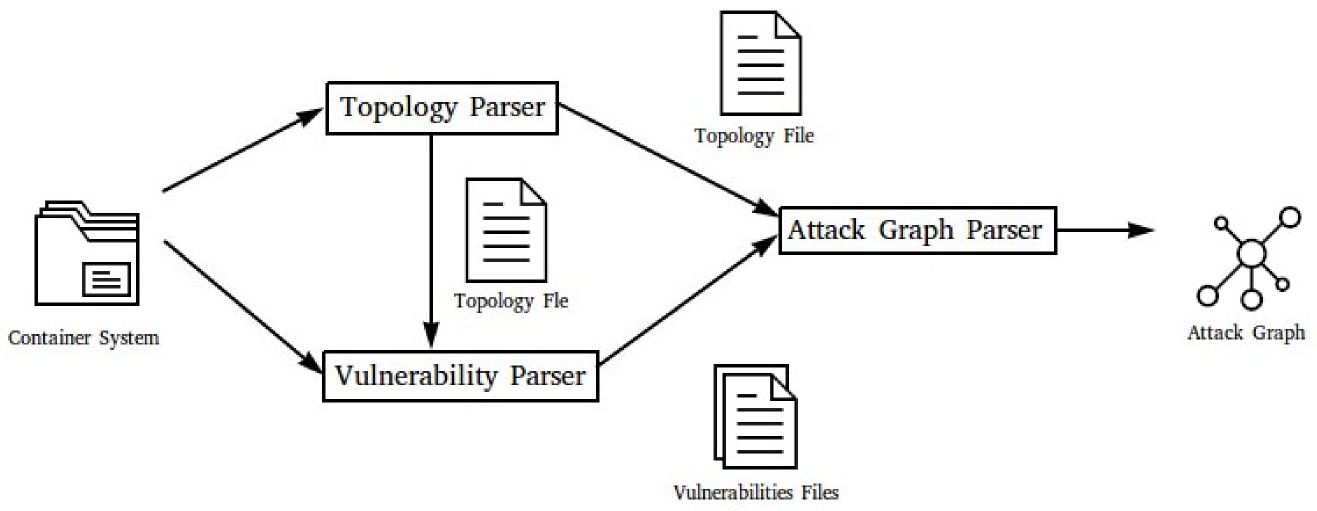

Fig. 1. Our Attack Graph System. The rectangles denote the main components of the system: Topology Parser, Vulnerability Parser and Attack Graph Parser. The arrows describe the flow of the system and the files are the intermediate products.

exploited. When an atomic attack is successfully executed, a postcondition is obtained. Postconditions are privileges acquired as a result of a successful attack. Both the Pre- and Postconditions are transformed from pre- and postcondition rules. We use in our attack graph generator the pre- and postconditions from (reference) which are manually selected by experts and evaluated.

Nodes in our attack graph model are represented as a combination of a docker image and its respective privilege level, while edges are a connection between node pairs with the vulnerability that is being exploited as a descriptor. Once an attacker exploits a given vulnerability, he gains the privilege of the new container and an edge is added to the attack graph.

In order to show how the attack graph generation works in practice, we present a small example. The example is taken from the Netflix OSS Github repository. Displayed on figure 2 is the topology of the example. The topology consists of ”Outside” node, ”Docker daemon” node and the a subset of other nodes. On figure 3, we can see a part of the resulting attack graph. An example path than an attacker to take would be to first attack the Zuul container and gain USER privilege. Then with this USER privilege it can exploit another vulnerability to gain ADMIN privilege. Once the ADMIN privilege has been obtained on Zuul, the attacker can attack the Eureka container 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 use another Zuul vulnerability to gain directly ADMIN privileges and then attack the Eureka container.

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

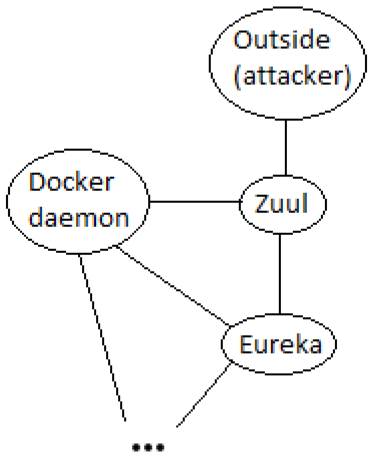

Fig. 2. Example topology graph. The topology graph is a subset of a real topology graph from the Netflix OSS example. Each node denote container(plus Docker Deamon and Outside) and each edge denotes a connection between two containers.

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.

1) Technical Details: Our system is developed for Docker 17.12.1-ce and Docker Compose 1.19.0. The code is written in Python 3.6, and we use Clair and Clairctl for vulnerabilities generation.

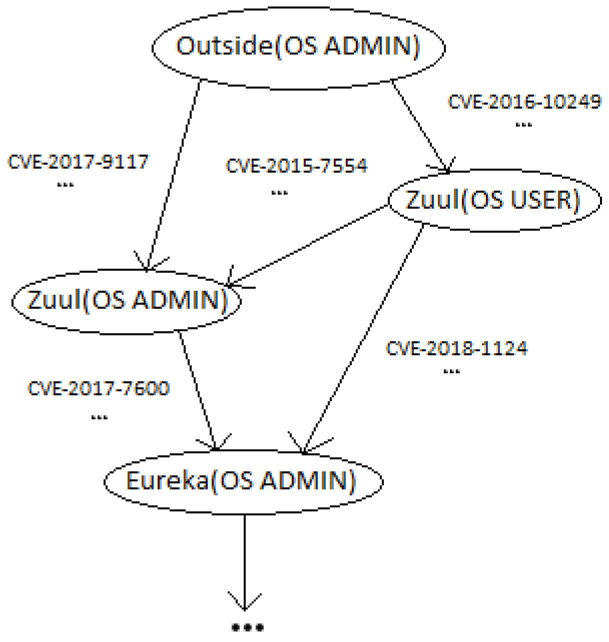

Fig. 3. Example resulting attack graph. This attack graph is a subset of a real attack graph from the Netflix OSS example. Nodes correspond to a pair of container plus privilege, while edges are atomic attacks.

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.

2) Topology Parser: The topology of Docker containers can be described at either runtime, 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 version 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 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 potentially 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) Vulnerability Parser: In the preprocessing step, we use Clair to generate the vulnerabilities of a given container. Clair is a tool that inspects a docker image, and generate its vulnerabilities by providing CVE-ID, description and attack vector for each vulnerability. 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) are: Access Vector(Local, Adjacent Network and Network), Access Com-

Statistics	example_1	example_5	example_20	example_50	example_100	example_500	example_1000
No. of Phpmailer containers	1	1	1	1	1	1	1
No. of Samba containers	1	5	20	50	100	500	1000
No. of nodes in topology	4	8	23	53	103	503	1003
No. of edges in topology	6	28	253	1378	5253	126253	502503
No. nodes in attack graph	5	13	43	103	203	1003	2003
No. edges in attack graph	8	68	863	5153	20303	501503	2003003
Topology parsing time	0.0082	0.0094	0.02879	0.0563	0.1241	0.7184	2.3664
Vulnerabilities preprocessing time	0.2551	0.2840	0.5377	0.9128	1.6648	6.9961	15.0639
Breadth-First Search time	0.0019	0.0209	0.2763	1.6524	6.5527	165.3634	767.5539
Total time	0.2654	0.3144	0.8429	2.6216	8.3417	173.0781	784.9843

TABLE I

Table with graph characteristics(no. of containers, nodes and edges in both the topology and attack graph) and executing times of the main attack graph generator components: Topology Parser, Vulerability Preprocessing Module and Breadth-first Search Module(the latter two parts of the main attack graph generation process). The examples are composed of two containers: Phpmailer and Samba. The Phpmailer container has 181, while the Samba container has 367 vulnerabilities. The topology time is the time required to generate the graph topology. The vulnerabilities preprocessing time is the time required to convert the vulnerabilities into sets of pre- and postconditions. The Breath-First Search is the main component that generates the attack graph. 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. The total time contains the topology parsing, the attack graph generation and some minor processes. 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.

plexity(Low, Medium, High), Authentication(None, Single, Multiple), Confidentiality Impact(None, Partial, Complete), Integrity Impact(None, Partial, Complete) and Availability Impact(None Partial Complete). Although it is very useful, Clair does not provide with an easy to use interface to analyze a docker image. As a result, we use Clairctl(Clair wrapper) in order to analyze a complete docker image.

4) Attack Graph Parser: After the topology file is extracted and the vulnerabilities for each container are generated, we continue with the attack graph generation.

We here 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 match a given rule, we use the pre- or postcondition of that rule. If more than one rule match, 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.

5) 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 going deeper in the graph. Pseudocode of our modified Breath 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 the nodes and the edges that make the attack graph. The algorithm first initializes the nodes, edges, queue and the passed nodes. Afterwards it generates the nodes which are a combination of the image name and the privilege level. Then into a while loop we iterate through every node, check its neighbors and add the edges if the conditions are satisfied. If the neighbor was not passed, then we add it 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.

IV. EVALUATION

An attack graph system needs to be scalable because in a real world scenario, there are many containers interconnected to each other. (some references and description with statistics). In this section, we first have a look at how others evaluate their proposed systems. We will then conduct few experiments in order to test the scalability of our system with different number of containers and links between them in both synthetically-made and real networks.

Evaluation of a given system is a usually required part, when evaluating a given program. Extensive scalability study is not available. Characteristics that vary are the number of nodes, their interconnectedness and the amount of vulnerabilities per container. All of these components contribute to the execution time. Even though the notion of attack graph vary, we hope to reach a comprehensive comparison. In our case we treat every container as a computer, and treat any physical connection between two computers as a connection between two containers.

[t]

For each example describe their setup and their results.

Seyner, the attack graph 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. They claim that the performance bottleneck is inside the graph generation procedure.(reference) Therefore model checkers are not the most suitable graph representation.

Ingols tested on a network of 250 hosts. He also tested on a simulated network of 50000 hosts in under 4 minutes.(reference) However he does this evaluation on the Multiple Prerequisite graph, which is different from ours.

Amman does not provide with scalability methods.

Evaluation pre/postconditions where achieved, but they provide no evaluation for attack graph generation.

In our scalability experiments we use Samba(reference) and Phpmailer(reference) containers which were taken from their respective Github repositories. We then extended this example and artificially made cliques of 5, 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.

On Table I we can see the results of our experiments. In each of the experiments the number of Phpmailer containers stays constant, while the number of Samba containers is increasing. This increase is done in a clique fashion, a node of each additional container is connected to every existing 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 present). 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 a combination of: 1 edge(”outside”-”Phpmailer”), n edges(”docker host” to all of the containers) and 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 less connected ones, the biggest time bottleneck is the preprocessing step. However these steps are with the number of containers because the container files are analyzed only once. The attack graph generation less time than the preprocessing step. Starting from example_500, we can notice sharp increase in the number of the BDF time to 165 seconds. For the previous example with example_100, needed 6.5 seconds.

We also performed tests on some real examples as described on table. These examples are different from the synthetic ones presented above, since they contain less containers. Also they are in a more linear fashion. For example, in order for an attacker to reach the database, he needs to gain suitable privilege levels of multiple intermediate containers.

We tested our generator on few other randomly selected examples from Github: NetflixOSS, Atsea Sample Shop App, and JavaEE demo. These systems are composed of different kind of technologies, different number of containers and vulnerabilities. Short summary of them is given on the table II. We want to show that we can generalize our approach for different types of technologies.

Characteristics of the Attack Graph Generator. - stopping criteria (When all edges are traversed) - BFS is guaranteed to traverse all of the edges - monothonicity no edge is visited more than once. - all of the vulnerabilities per edge are checked - only the ones that fulfill a certain precondition are added. If precondition is not fulfilled, then the edge is not added.

V. Conclusion

VI. Future Work

Acknowledgment

References

[1] Merkel, Dirk. ”Docker: lightweight linux containers for consistent development and deployment.” Linux Journal 2014.239 (2014): 2.

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

[3] Clairctl. https://github.com/jgsqware/clairctl

[4] Computer Security Division of National Institute of Standards and Technology. National vulnerability database version 2.2 (2010), http://nvd.nist.gov/

[5] Ingols, Kyle, Richard Lippmann, and Keith Piwowarski. ”Practical attack graph generation for network defense.” Computer Security Applications Conference, 2006. ACSAC’06. 22nd Annual. IEEE, 2006.

[6] Aksu, M. Ugur, et al. ”Automated Generation Of Attack Graphs Using NVD.” Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. ACM, 2018.

[7] Sheyner, Oleg, et al. ”Automated generation and analysis of attack graphs.” Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on. IEEE, 2002.

[8] Artz, Michael Lyle. Netspa: A network security planning architecture. Diss. Massachusetts Institute of Technology, 2002.

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 II

LIST OF RANDOMLY SELECTED EXAMPLES THAT WERE ANALYZED WITH OUR ATTACK GRAPH GENERATION SYSTEM.