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. ARCHITECTURE

In this section, we examine the architecture of our attack graph generator. We first present a brief overview of the main concepts. Next, we have a look at the preprocessing step. Lastly we have look into the main algorithm for generating the attack graph.

1) Main concepts: Preconditions are conditions that have to be fulfilled in order for the attack to take place. Postconditions are the result of a successful attack and the privileges acquired.

Atomic attack is one step or edge in the attack graph. A state is the container with its privilege level.

We model the privileges into a hierarchy. The priviliges 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 infected. Since VOS are on some hosts, they have lower level of hierarchy.

¡INSERT IMAGE PRIVILEGES¿

¡INSERT IMAGE EXAMPLE¿

¡INSERT IMAGE ATTACK GRAPH¿

2) Preprocessing: In our preprocessing step, we use Clair in order to generate the vulnerabilities for a given container. Clair provides the CVE-ID, description and attack vector. Attack vector is an entity that descriptes which conditions and effects are connected to this vulnerability. Clairctl is a wrapper that we use in order to analyze a docker image.

Additionaly, we parse the attack vectors from the vulnerability database to generate attack rules. We use rules annotated

docker-compose.yml is used for extraction of the topology of the system. The containers that have published ports are connected to the outside. Some containers have privileged access. That means that an attacker with access to these containers, has also access to the docker daemon. This can be done by us

This results in a list of container vulnerabilities with their preconditions and postconditions.

3) Breadth-First Search: After the preprocessing step is done, the vulnerabilties are parsed and their pre- and postconditions are extracted, together with the topology are feed into the Bradth-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.

Some properties of BFS:

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 - monotonicity properties - complexity time complexity O(—N— + —E—) where —N— is the number of nodes and —E— is the number of edges in the attack graph. memory complexity We use a modification of the breadth first search algorithm to find the nodes and the edges of the attack graph.

```
Data: topology, container_exploitability,
      privileged_access
Result: nodes, edges
nodes, edges, queue, passed_nodes = list(), dict{},
 Queue(), [];
queue.put(goal_container);
nodes = get_nodes();
while ! queue.isEmpty() do
    ending_node = queue.get();
    passed_nodes[ending_node] = True;
    cont_exp_end =
     container_exploitability[ending_node];
    neighbours = topology[ending_node];
    for neighbour in neighbours do
        if neighbour == "outside" then
            edges.append(create_edges());
            continue;
        end
        if ! passed_nodes[neighbour] then
            queue.put(neighbour);
        end
        if neighbour == goal_container then
            continue;
        end
        edges.append(create_edges())
    end
end
```

Algorithm 1: Breadth-first search algorithm for generating an attack graph.

The algorithm requires the 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 exploitable vulnerability. Then into a while loop we iterate through every node, check its neighbours and add the edges. If the neighbour was not passed, then we add it to the queue. The algorithm terminates when the queue is empty.

IV. EXPERIMENTS

In this section, we will conduct few experiments in order to test the scalability of our system with different number of containers and links between them.

Throughout the experiments, the biggest time bottleneck is the preprocessing step, and the graph drawing step. However these steps are with linear complexity because the container files are analyzied only once. The attack graph generation less time than the preprocessing step.

The following experiments were perfomed. We used the Samba(reference) and Phpmailer(reference) examples which were taken from . Where we artificially made clique of 1, 5, 20, 50, 100, 500 and 1000 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 the table(Table I) we can see the results of our experiments. In each of the experiments the number of Phpcontainer is constant, while the number of Samba containers is increasing in a clique fashion. There are also two artifitial containers("outside" that represents the outside world 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: 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$_5$, $the number of edges in the topology graph would be 32 : 1 outside edge, 6 docker host edges (n = 6, 1 Phpmailer and 5 Sambas) and 25 clic queedges (5 * 6/2 = 15)$.

We also performed testing on a real example- atsea sample shop app(reference). The system is composed of the containers app, database, payment$_g$ $ateway and reverse_p roxy$.

[t]

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.

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	4	22	232	1327	5152	125752	501502
No. nodes in attack graph	5	13	43	103	203	1003	2003
No. edges in attack graph	4	12	42	102	202	1002	2002
Topology parsing time	0.0052	0.0096	0.0257	0.0638	0.1185	0.8763	1.8723
Vulnerabilities preprocessing time	14.7317	15.1778	15.5593	17.1801	16.6612	23.8188	28.1352
Breadth-First Search time	0.0017	0.0106	0.1043	0.6128	2.1841	55.1850	193.2329
Total time	14.7387	15.1981	15.6895	17.8567	18.9638	79.8802	223.2404

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.