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

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

2.1 Attack Graphs

2.2 Microservices

3 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 runsâĂŤwhat 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 [9].

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 reach 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 gives access to the container's neighbors. After a successful exploitation of multiple intermediate containers, he finally has access to the database and its content. 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
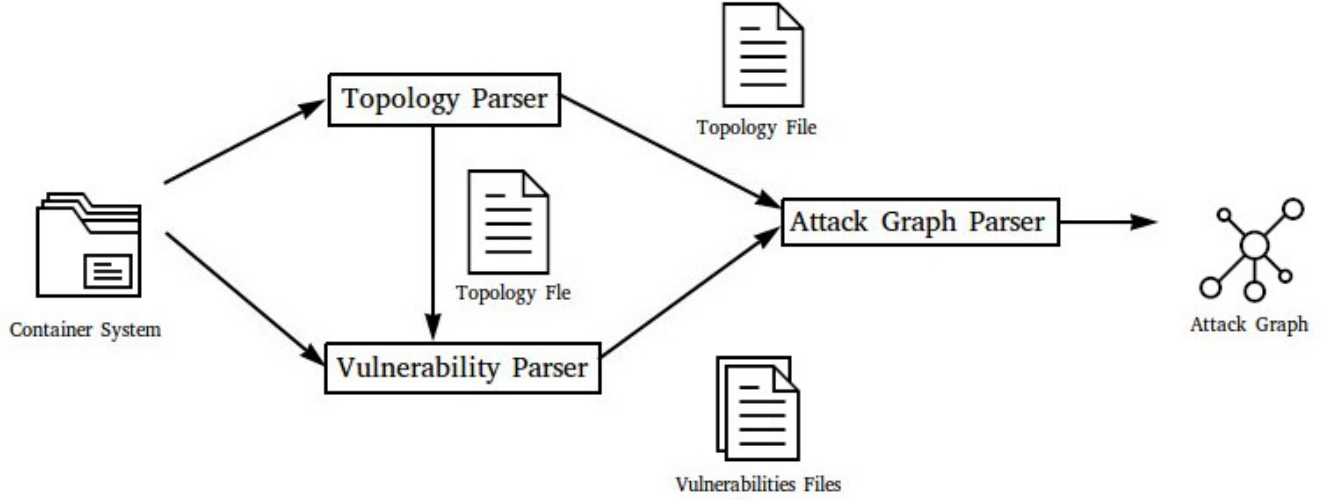
Figure 1: Our Attack Graph System. The rectangles denote the main components of the system: Topology Parser, Vulnerability Parser and Attack Graph Parser. The arrows describe the flow of the system and the files are the intermediate products.

machines, and their exploitation does not imply exploitation of the host machine, they are in the lower level of hierarchy [6].

Furthermore, in order for atomic attacks to be performed, certain preconditions have to be ensured. Preconditions are privilege levels that are required so that a vulnerability can be 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 manually selected by experts and evaluated [6].

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.[5] 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.
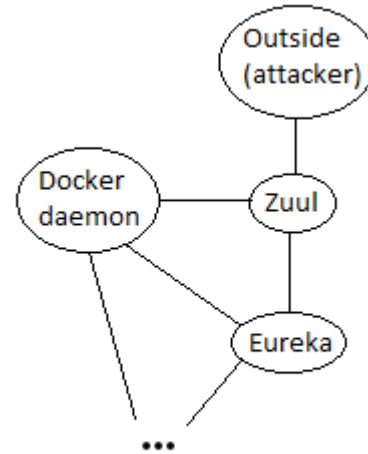
Figure 2: Example topology graph. The topology graph is a subset of a real topology graph from the Netflix OSS example. Each node denote container(plus Docker Deamon and Outside) and each edge denotes a connection between two containers.

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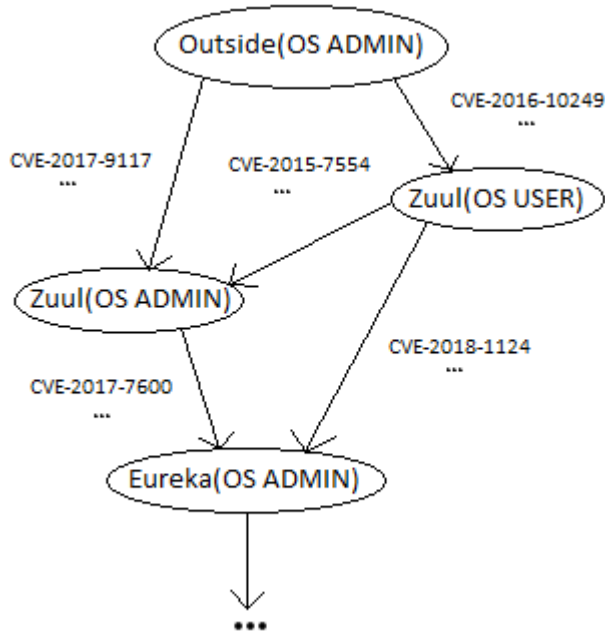
Figure 3: Example resulting attack graph. This attack graph is a subset of a real attack graph from the Netflix OSS example. Nodes correspond to a pair of container plus privilege, while edges are atomic attacks.

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.

3.0.1 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 [2] and Clairctl [1] 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.

3.0.2 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 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 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 $curr_priv > precond[vul]$ **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.

3.0.3 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 generate its

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 1: 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, Vulnerability 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.

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) [7] 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). 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.

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

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.

3.0.5 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 nodes and 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 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

In real-world microservice architectures, many containers are interconnected to each other. This raises the need for a scalable attack graph system. In this section, we first have a look at how others evaluate their systems. We then conduct few experiments in order to test the scalability of our system with different number of containers, and connections in both artificial and real networks. All of the experiments were performed on a Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz with 8GB of RAM running Ubuntu 16.04.3 LTS.

Extensive scalability study of attack graph generators is rare in current literature and many parameters contribute to the complexity of a comprehensive analysis. Characteristics that usually vary in this sort of evaluation are the number of nodes, their interconnectedness and the amount 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 our case we compare our system to existing work by treating every container as a computer, and any physical connection between two computers as a connection between two containers.

Seyner in his work tests the system in both a small and extended examples [12]. 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. Sheyner claims that the performance bottleneck is inside the graph generation procedure.

Ingols tested his system on a network of 250 hosts. He afterwards continued his study on a simulated network of 50000 hosts in under 4 minutes [8]. 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 explanation of how the hosts are connected, does not make it directly comparable to our method.

Ou provides some more extended study where he tests his system on more examples [10]. He mentions 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. He also provides an evaluation where he MulVAL clearly outperforms the Sheyner's system.

In our scalability experiments we use Samba [4] and Phpmailer [3] 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 1 we can see 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, 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 smaller configurations, the biggest time bottleneck is the preprocessing step. However this step increases in 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 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.

We also performed tests on some real examples as described on table 2 and extend our system to different types of technologies, number of containers and vulnerabilities. The examples are as follows: NetflixOSS, Atsea Sample Shop App, and JavaEE demo. These examples are different from the synthetic ones presented above, since they contain less and different containers. They are connected in a more linear fashion in contrast to the full connection provided in the previous example. For example, in order for an attacker to reach a database, he needs to gain suitable privilege levels of multiple intermediate containers in several steps.

5 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. 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 specific host are available. However previous works have 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. 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 with using model checkers with goal property.[8]

Model checkers use computational logic to check if a model is correct, and otherwise, they provide a counter example. A collection of these counter examples 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(Ingols reference) As a result of this, more scalable approach was needed. Amman et al. extends this work with some simplifications and more efficient storage [11].

Ou et al. use logical attack graph [10] and Ingols [8] et al. use breath first search algorithm in order to tackle the scalability issue. Ingols discusses the redundancy Full and Predictive graphs and model an attack graph as a 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. builds on top of Ingols's system and evaluate a set of rule pre- and postcondtions in generating attacks [6]. They define a specific test of pre- and postcondition rules and test their correctness. In their evaluation, they use machine learning approach.

Name	Description	Technology stack	No. Containers	No. vuln.	Attack Graph generation time	Github link
Netflix OSS	Combination of containers provided from Netflix.	Spring Cloud, Netflix Ribbon, Spring Cloud Netflix, Netflix's Eureka	10	4111	0.5120 sec.	https://github.com/Oreste-Luci/netflix-oss-example
Atsea Sample Shop App	An example online store application.	Spring Boot, React, NGINX, PostgreSQL	4	120	0.2761 sec.	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	0.2613 sec.	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	0.2570 sec.	https://github.com/opsxcq/exploit-CVE-2016-10033 https://github.com/opsxcq/exploit-CVE-2017-7494

Table 2: List of randomly selected examples that were analyzed with our attack graph generation system.

Containers and microservice architectures, 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 so far in the area of attack graph generation for docker containers. Therefore we extend the work from Ingols [8] and Aksu [6] in conjuction to Clair OS to generate attack graphs for microservice architectures. 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.

6 CONCLUSION

7 FUTURE WORK

REFERENCES

[1] [n. d.]. Clairctl. https://github.com/jgsqware/clairctl. Accessed: 2018-08-31.
[2] [n. d.]. CoreOS Clair. https://github.com/coreos/clair. Accessed: 2018-08-31.
[3] [n. d.]. PHPMailer < 5.2.18 Remote Code Execution. https://github.com/opsxcq/exploit-CVE-2016-10033. Accessed: 2018-08-31.
[4] [n. d.]. SambaCry RCE exploit for Samba 4.5.9. https://github.com/opsxcq/exploit-CVE-2017-7494. Accessed: 2018-08-31.
[5] [n. d.]. Spring Cloud - Netflix OSS Example. https://github.com/Oreste-Luci/netflix-oss-example. Accessed: 2018-08-31.
[6] 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.
[7] Harold Booth, Doug Rike, and Gregory A Witte. 2013. *The National Vulnerability Database (NVD): Overview*. Technical Report.
[8] 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.
[9] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.
[10] 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.
[11] 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.
[12] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. 2002. Automated generation and analysis of attack graphs. In *null*. IEEE, 273.