2022 IEEE 15th International Conference on Cloud Computing (CLOUD) | 978-1-6654-8137-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLOUD55607.2022.00038

Towards a Security Stress-Test for Cloud Configurations

1st Francesco Minna
Vrije Universiteit Amsterdam (NL)
f.minna@vu.nl

2nd Fabio Massacci
University of Trento (IT)
Vrije Universiteit Amsterdam (NL)
fabio.massacci@ieee.org

3rd Katja Tuma
Vrije Universiteit Amsterdam (NL)
k.tuma@vu.nl

***Abstract*—Securing cloud configurations is an elusive task, which is left up to system administrators who have to base their decisions on "trial and error" experimentations or by observing good practices (e.g., CIS Benchmarks). We propose a knowledge, AND/OR, graphs approach to model cloud deployment security objects and vulnerabilities. In this way, we can capture relationships between configurations, permissions (e.g., CAP_SYS_ADMIN), and security profiles (e.g., AppArmor and SecComp). Such an approach allows us to suggest alternative and safer configurations, support administrators in the study of what-if scenarios, and scale the analysis to large scale deployments. We present an initial validation and illustrate the approach with three real vulnerabilities from known sources.**

***Index Terms*—knowledge graph, AND/OR graphs, containers, security, microservices, cloud**

I. Introduction

Container engines and orchestration tools, such as Docker and Kubernetes, abstract the complexity of the underlying technologies (e.g., Linux namespaces and control groups), allowing companies to reduce the time to market, making it easier to deploy and share applications as containers. Yet, they also pose new security challenges. Indeed, it is hard to understand such hidden complexity in a highly dynamic environment (44% of containers are executed for less than five minutes [1]); also, misconfiguration is the most prevalent vulnerability in the cloud [2], 76% containers are running as root, and 75% of containers are running with critical vulnerabilities [1]. Several tools exist for analyzing container configurations and for vulnerability scanning. Yet, as we illustrate in §III, most "vulnerability scanners" only check for the software bill of materials. When checking for actual configurations (e.g., the Checkov tool by Bridgecrew), most of the time, only best practices are taken into account (i.e., CIS Benchmarks). Such analysis report vulnerable software packages or best practices that are not met; however, they can not perform what-if analysis suggesting safer configurations. To fill this gap, we propose a *knowledge, AND/OR, graphs approach to model cloud deployment security objects and vulnerabilities*, where configurations, permissions, capabilities, and other security features are relevant to the model. Fig. 1 shows the workflow steps of the proposed stress-test approach on Docker containers, namely, (i) building the knowledge graph from configuration files, (ii) enriching the graph with known vulnerabilities (represented as AND/OR graphs), and (iii) testing and fixing the system by interacting with the user.

This work has received funding from the European Union under the H2020 grant 952647 (AssureMOSS).

Listing 1 Vulnerable execution of the container taken from [4].

```
docker run -it -d --cap-add=SYS_ADMIN
  ↪  --security-opt apparmor=unconfined ubuntu
```

The idea behind this approach is equivalent to the stress test for banks [1], but for cloud deployments. The approach would allow administrators to analyze what-if scenarios with known and new vulnerabilities, improving the security posture of cloud deployments by understanding the trade-off between security requirements and functionality loss. To this end, we build knowledge and AND/OR graphs and implement (configuration) parsing and search algorithms, using Python and the Neo4J graph database (§IV), to identify vulnerabilities and correspondingly suggest security fixes (§V). We present an initial validation by illustrating the proposed approach in §VI and discuss the preliminary results and future work in §VII. Finally, we present the concluding remarks in§VIII.

II. Simple Attack Scenarios

Hereby we describe the implications of two recent container attack scenarios, to better illustrate the problem faced by cloud administrators. In 2019, a researcher from Google posted on Twitter a proof of concept (PoC) to escape from privileged Docker containers, by exploiting the release_agent function of control groups (cgroups) [4]. Listing 1 shows a possible vulnerable configuration. This configuration adds system administrator capabilities to the Docker default root user and disables the Docker default AppArmor security profile (i.e., granting the `mount` system call).

By creating a new cgroup, enabling cgroups notifications on release, and specifying the release_agent script to be executed (once all processes within the same cgroup are killed), an attacker can achieve arbitrary code execution on the container's host. To exploit this technique, a container must either run as privileged or with (the Docker default) root

[1]"the stress test for banks is used to assess their resilience to adverse economic and market developments and contributes to the assessment of systemic risk" [3].

2159-6190/22/$31.00 ©2022 IEEE
DOI 10.1109/CLOUD55607.2022.00038

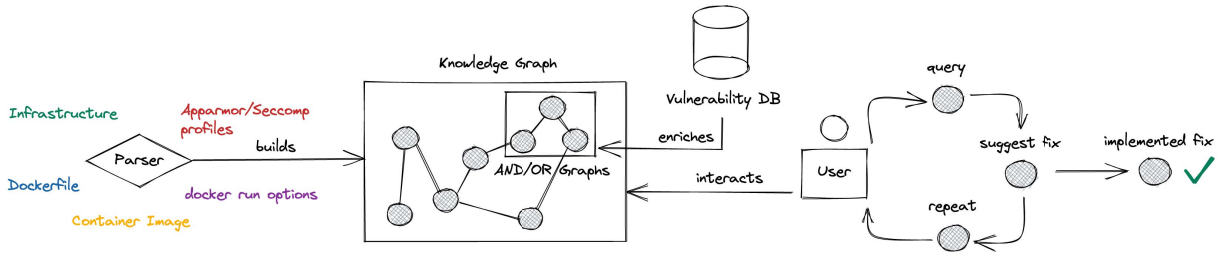

FIGURE I. The workflow of the proposed stress-test approach on Docker containers.

user, the CAP_SYS_ADMIN capability granted and mount system call allowed. By foiling any (or all) preconditions the container escape fails. For example, as a countermeasure, one could drop all capabilities (i.e., with the Docker option `--cap-drop ALL`). Yet this is hardly a side-effect-free solution (e.g., impacting the container performance). Similarly, in March 2022, a vulnerability affecting the Linux kernel leading to privilege escalation was discovered, namely, CVE-2022-0492 [5]. Again, this vulnerability exploits a vulnerable implementation of the release_agent function executed once all processes belonging to a cgroup are killed; in this case, the kernel did not check for the process to have the right permission (i.e., CAP_SYS_ADMIN capability). To exploit this vulnerability, the container must run with (the Docker default) root user (or with the CAP_DAC_OVERRIDE capability) and with mount and unshare system calls allowed; obviously, containers with more permissions (e.g., privileged or with the CAP_SYS_ADMIN capability) are also affected. These two examples show how (permissions) (mis-)configurations are a game-changer. Indeed, misconfigurations allow most updated software to be exploited while vulnerable old versions may not be exploited with "safe" configurations.

III. Related Work

Access control and network security policies for cloud deployments are well studied by providing either access control solutions [6], automatic RBCA policy generation [7], or mechanisms for formal verification of microservice deployments [8]. Yet, the automatic or even computer-aided exploration and validation of configuration policies *as available in practical deployments* (e.g., combinations of Docker commands) and their subsequent testing, still remain largely unexplored in the academic literature.

Microservice Attack Generation by Ibrahim et al. [9] investigated (theoretical) attacks on microservices using vulnerabilities identified in software packages to build attack graphs. Whether these attacks are actually possible has not been tested. Also, this analysis cannot be readily translated to changes into the practical configuration (e.g., drop the SYS_ADMIN capability).

Security industrial tools offering different kinds of vulnerability scans and static analysis for cloud environments are also available on the market. Configuration analyzers (e.g., snyk [2] and Docker Bench [3]) analyze configuration files and settings and only check them against common or custom security best practices and policies (e.g., compliance with CIS Benchmarks); instead, other tools only provide compliance tests [4]. Vulnerability Scanners (e.g., Clair [5] and Trivy [6]) typically retrieve a software bill of material (SBOM) from container images and compare them against vulnerability repositories (e.g., NVD[7] or Debian Security Bug Tracker[8]) to check if the current version of a package in use is *reported to be* vulnerable. While they all formally meet the NIST definition of vulnerability scanning ("a technique used to identify hosts attributes and associated vulnerabilities" [10]), their actual operation is not what a security expert would intuitively expect, for example, from the common `nmap` network scanner [11], or even less from a static analyzer for software security [12]. Also, such tools often generate false alerts as they are based on the same overcautious vulnerability reporting that is well known in secure software engineering for the analysis of vulnerabilities in open source libraries [13], [14].

Automatically suggesting a "fix" for (cloud) misconfigurations is, to this day, an open challenge. All previous tools do not suggest or provide any fix for detected vulnerabilities, leaving up to the end user the mitigation process to eliminate the alerts.

Knowledge graphs for cybersecurity, in particular, of cloud environments, have already been investigated in the past. Yet, most existing approaches use knowledge graphs for security awareness, and intelligence sharing [15], [16], rather then security analysis of configurations. A comprehensive review can be found in [17]. Banse et al. [18] builds a Cloud Property Graph (CloudPG) using both static code analysis and an ontology; similarly to our idea, they propose an analysis framework to query the graph to identify security issues. However, they focus on bridging the gap between static code analysis of the application and its runtime environment. While

[2] https://snyk.io/product/container-vulnerability-management/
[3] https://github.com/docker/docker-bench-security
[4] e.g., terrascan by Accurics, built on top of OPA, and Bridgecrew, which are open source, along with CIS-CAT Pro, and Lacework which are commercial.
[5] https://github.com/quay/clair
[6] https://github.com/aquasecurity/trivy
[7] https://nvd.nist.gov/
[8] https://security-tracker.debian.org/tracker/

the outcome is a sophisticated analysis of data flow leaks, this approach requires full access to the application source code. Our approach complements this work by going in the opposite direction, which is the analysis of cloud deployments to suggest alternative and safer configurations.

IV. Building the Graphs

Hereby we describe the methodology for capturing and analyzing cloud configurations by using a knowledge graph and representing known vulnerabilities as AND/OR graphs.

Build a knowledge graph. First, we parse the configuration files to represent the infrastructure on which the container engine is running (e.g., Vagrantfile), the container engine being used (e.g., Docker), the container image (e.g., Dockerfile), and finally, the docker run options that will be used to run the container (e.g., --privileged, --volume, and --user) along with the security profiles (e.g., AppArmor and Seccomp). We then map these objects to nodes and their logical relations to edges in the knowledge graph; for example, the $Container$ (node) has capability (edge) SYS_ADMIN (node), and the $Docker\_Engine$ (node) is using (edge) $Kernel$ (node) of version (edge) 4.9 (node). In addition, we also model possible configuration *alternatives*; for example, we add nodes representing alternative versions of the Linux kernel or the Docker engine. Most state-of-the-art tools do not parse the assigned permissions (e.g., Linux capabilities and system calls), and the Docker run options for possible misconfigurations; the latter, in particular, are important because Dockerfile instructions can be overwritten by the Docker run options.

We use the Python language to parse configuration files and the Neo4J graph database to store the knowledge graph and vulnerabilities AND/OR graphs described as follow.

Build vulnerabilities graphs. Secondly, we enrich the knowledge graph with a dataset of known vulnerabilities and misconfigurations [19]. We use a two-dimensional taxonomy to classify container vulnerabilities and misconfigurations, slightly based on the attack taxonomy proposed in [20]. Within the first dimension we classify the vulnerabilities in three categories based on the affected component, namely, *Container*, *Kernel*, and *Engine*. Within the second dimension, we classify the vulnerabilities based on the MITRE ATT&CK tactics and techniques (e.g., privilege escalation and network service scanning) [19].

We represent each vulnerability as an AND/OR graph, where the root is the CVE and the rest of the nodes represent the attack assumptions encoded as AND and OR conditions, as already presented in prior work [21]. For example, given a Linux kernel vulnerability, we use an OR node as a parent node of a set of children nodes each representing a vulnerable kernel version (e.g., Linux kernel versions between 5.1 and 5.9). We then link the knowledge graph, representing the underlying infrastructure, to each AND/OR vulnerability graph, based on the current container configuration; for example, if a container has a capability that is an assumption of a CVE, this will be represented as the $Container$ (node) has capability (edge) SYS_ADMIN (node) assumption of (edge) $CVE-yyyy-xyz$ (node)). The vulnerabilities dataset is represented using a JSON file; in particular, Listing 2 shows the JSON representation of the first container escape described in §II.

Listing 2 The JSON representation of the container escape attack described in [4].

```
"Escape_1": [{
        "engine": "docker",
        "mitre_tactic": "privilege_escalation",
        "mitre_technique": "escape_to_host",
        "pre_conditions": [{
                "user": "root",
                "capability": "CAP_SYS_ADMIN",
                "syscall": "mount",
                "read_only_fs": "false",
                "no_new_priv": "false"
        }],
        "post_conditions": [{
                "impact": "privilege_escalation"
        }]
}]
```

Fig. II shows the Neo4J representation of the knowledge graph of an Ubuntu container, run as shown in Listing 1, linked to the AND/OR graph representing the container escape attack represented in Listing 2. Due to space reasons, additional nodes representing alternative versions, capabilities, and system calls are not shown in the figure.

V. Suggesting a Fix

Hereby we describe the methodology for suggesting security fixes, based on user preferences, to mitigate vulnerabilities.

Given current configurations and vulnerabilities, we can query the graph to unveil the presence of potentially exploitable vulnerabilities; for example, we can query whether a set of permissions to exploit a vulnerability is granted to a running container, or a vulnerable software version is being used. We present a query example in §VI. To rank and suggest the prioritized fixes, we use the **Analytic Hierarchy Process** (AHP). AHP is a multiple criteria decision-making methodology, where a hierarchy of solution alternatives is created, then the user is asked to provide weights to each solution alternative, and finally, pair-wise alternatives are compared until the optimal solution is obtained [22]. Before evaluating the presence of vulnerabilities within the graph, we interact with the end-user to rank (i.e., weight) the list of possible fixes that can mitigate attacks. For example, we may ask the user to assign a preference, with 1 least favorite and 9 most favorite, to run a container as not root user or to remove certain capabilities. The following is the list of possible fixes that we suggest to the user: version_upgrade, not_privileged, not_root, not_capability, not_syscall, read_only_fs, no_new_priv. This ranking will help us in suggesting the most effective fix that, at the same time, the user is also most willing to accept and implement.

Test & Fix for vulnerabilities. At any given time, we test (by querying the graph) and fix (by suggesting and eventually implementing security fixes) the system. To this aim, we rely

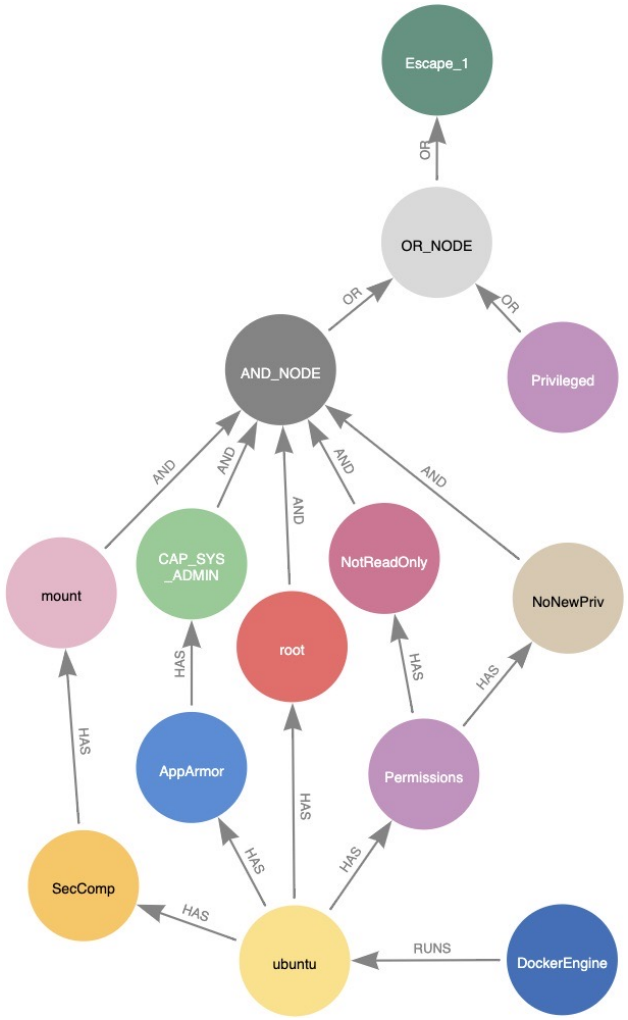

FIGURE II. The Neo4J representation of the knowledge graph of an Ubuntu container, run as shown in Listing 1, linked to the AND/OR graph representing the container escape attack represented in Listing 2.

on the user to accept the best alternative fix (based on the provided weights). An attack is mitigated by implementing a fix (i.e., invalidating one attack's assumption), which is reflected in the graph by removing or adding an edge.

Algorithm 1 shows the procedure to stress-test a cloud system by unveiling the presence of one vulnerability. The steps of the algorithm are (1) parsing the configuration files and building the knowledge graph, (2) adding and link the vulnerability AND/OR graph, (3) fetching user preferences, and finally (4) suggesting the best fix. To this aim, the algorithm iteratively suggests security fixes to mitigate the vulnerability assumptions, following the user preferences. Eventually, the algorithm can automatically accept the most preferred fix.

In some cases, the user can also decide to intentionally keep a few low-risk vulnerabilities to avoid trading off the performance of the system. In such cases, although the vulnerability query will still be satisfied, this approach may help move towards a "safer" configuration state (e.g., by running a container with all and only needed permissions, instead of all).

Algorithm 1 Stress-test a cloud system for one vulnerability.

Require: Configuration files
Require: Vulnerability
Ensure: Safe configuration
1: Build Knowledge graph;
2: Enrich the graph with the vulnerability;
3: Get list of user preferred fixes;
4: GET_PREFERRED_FIX($vuln\_query$)
5: **function** GET_PREFERRED_FIX(vul_query)
6: **while** $vuln\_query == true$ **do**
7: Collect vulnerability assumptions;
8: Order assumptions by AHP;
9: **repeat**
10: Suggest assumption to remove;
11: **if** $user\_accepts\_fix$ **then**
12: remove edge;
13: **until** $user\_not\_happy$

In general, the configuration of realistic cloud deployments may contain several vulnerabilities. In this case, the problem requires finding a globally optimal solution (i.e., the set of fixes most preferred by the user). Therefore, fixes for one vulnerability must be additionally weighted in combination with other vulnerabilities fixes. Currently, our algorithm relies on user input (AHP weights) to determine such preferences. In future work, we plan to extend the current implementation to automatically compute the set of globally optimal fixes; also, when dealing with many containers and vulnerabilities, we plan to evaluate whether the "*user is happy*" in a scalable way, for example, by setting a global minimum risk threshold for accepting fixes.

VI. Preliminary Evaluation

We investigated the cost, in terms of number of nodes and edges in the graphs, for storing different container configurations and vulnerabilities. In addition, we tested the feasibility of the proposed approach on three well known vulnerabilities.

The cost of storing containers. Tab. I shows the cost, in terms of number of nodes and edges, of storing a container image and several containers (using the Docker default security profile, with a subset of capabilities and system calls allowed). Specifically, we use one node to represent the host virtual machine, 50 nodes to represent the versions of Linux kernel (from version 3.9, released after March 20 2013, when Docker was released), one node to represent the Docker engine, 132 nodes to represent the versions of `Docker`, 83 nodes to represent the versions of `containerd`, and 18 nodes to represent the versions of `runc`. In addition, we use 41 nodes to represent Linux capabilities and 364 nodes to represent Linux system calls.

This amounts to 691 nodes and 6 relationships (first row in Tab. I). Additional nodes are used to represent container

permissions, security profiles, and Docker run options; for example, custom security profiles (e.g., user-defined AppArmor profiles), may require additional nodes and relationships corresponding to the granted capabilities and system calls, as well as relevant Docker run options (e.g., *--user* and *--volume*).

TABLE I
THE COST OF STORING CONTAINER IMAGES AND (DEFAULT) CONFIGURATIONS INTO THE KNOWLEDGE GRAPH.

Object/s	#_Nodes	#_Edges
Initialization	691	6
1 Image	693	8
1 Container (default)	702	349
100 Container (default)	999	1.339
1.000 Container (default)	3.699	10.339

In the previous table, we considered all containers to be instantiated from different images, thus with two additional nodes and relationships. Overall, the number of nodes and edges in the knowledge graph does not grow exponentially. This is promising as it would allow us defining efficient algorithms to suggest the best security fixes.

Tab. II shows different Docker run options with the corresponding number of edges; the number of nodes in the graph remains the same.

TABLE II
LIST OF POSSIBLE DOCKER RUN OPTIONS WITH THE CORRESPONDING NUMBER OF EDGES.

Docker run options	#_Edges
`--cap-drop ALL --cap-add NET_BIND_SERVICE`	335
`--cap-add NET_ADMIN --security-opt apparmor=unconfined`	348
`--privileged`	420

Intuitively, additional configurations exist, for example, as a combination of adding and removing capabilities, granting or denying system calls, using custom users or a read-only filesystem.

The cost of storing vulnerabilities. Tab. III shows the dataset of vulnerabilities we collected from different sources, along with the number of nodes and edges used to represent each vulnerability. We classify vulnerabilities in three categories, namely, *container profile misconfigurations*, *Linux kernel bugs*, and *container engine vulnerabilities*, as suggested in [23]. Within each category, we have a subcategory representing the impact of each vulnerability or attack based on the MITRE ATT&CK tactic and techniques for containers [19]. To validate our stress-test approach, we retrieved a subset of vulnerabilities belonging to each category. In particular, for *container profile misconfigurations* we retrieved 6 escaping attacks, for *Linux kernel bugs* we retrieved 24 CVEs (only ten shown in Tab. III), mostly from [20] and, for *container engine vulnerabilities*, considering only Docker (or Docker subcomponents, such as `runc` and `containerd`), we retrieved 6 CVEs from [23]. In total, our dataset contains 36 vulnerabilities.

Listing 3 A Neo4J query to check a container relationship.

```
MATCH (c:Container {name: "Nginx"})
MATCH (p:Permissions {name: "Privileged"})
RETURN EXISTS( (c)-[:HAS]-(p) )
```

TABLE III
INFORMATION ABOUT THE VULNERABILITIES DATASET, DIVIDED BY CATEGORIES, TOGETHER WITH THE NUMBER OF NODES AND EDGES.

Categ.	Sub-cat.	CVE	#_Nodes	#_Edges
Container misc.	Escape to host	Cgroup escape	8	12
		Cap. SYS_MODULE	6	7
		Kernel fs /sys	2	2
		Host devices	7	11
		Host /root	4	6
		Docker socket	4	6
Kernel bugs	Exploitation	CVE-2022-0847	11	19
		CVE-2022-0492	58	62
		CVE-2022-0185	22	25
		CVE-2020-14386	44	44
		CVE-2017-7308	47	47
		CVE-2017-5123	10	10
		CVE-2016-8655	37	40
		CVE-2016-4997	20	20
		CVE-2017-6074	23	23
		CVE-2017-1000112	27	27
Docker vuln.	Code injection	CVE-2019-14271	9	14
	Exploitation	CVE-2020-15257	106	109
	Escape to Host	CVE-2016-9962	27	29
		CVE-2018-15664	66	66
		CVE-2019-5736	299	300
	DoS	CVE-2020-13401	221	224

The number of nodes and edges in the table corresponds to the AND/OR graph representing each vulnerability.

Vulnerabilities Queries. To illustrate the feasibility of our approach, we performed an initial evaluation on three vulnerabilities, namely, *Cgroup escape*, *CVE-2022-0492*, and *CVE-2020-13401* (green rows in Tab. III). To the best of our knowledge, an algorithm to traverse AND/OR graph stored in Neo4j does not yet exist. In future work, we plan to implement such an algorithm to automatically check the presence of vulnerabilities. However, for our initial validation, we manually checked whether a vulnerability is exploitable or not. Listing 3 shows a Neo4J query to check whether a certain property (e.g., permission, capability, or system call) belongs to a container or not.

In future work, based on the current implementation, we plan to test this approach on all 36 vulnerabilities with different configurations (e.g., with different AppArmor profiles and docker run options) and, eventually, on different container

engines. By doing so, we can evaluate more extensively the results of stress tests and explore the support of what-if scenarios; for example, we can claim the most effective mitigation against most attacks but also evaluate the most dangerous configurations. Also, at the end of each stress-test execution, we could use the number of removed edges (i.e., attacks assumptions) as an indicator of the resilience of the cloud system; eventually, the indicator can be used as a measure of quality in the context of more comprehensive validations.

VII. Discussion

The proposed approach might also be suitable to build attack paths in more complex environments (e.g., Kubernetes clusters). Efficient graph algorithms could be used to cast and then address a number of security issues. For example, reachability (e.g., given the current configuration, whether the attacker can reach a new state), minimum weight traversal (e.g., the easiest exploitation path), and vertex cuts (e.g., suggesting a policy or configuration change that invalidates an attack). Also, for a dynamic picture, administrators could run the stress-test at a given time interval and evaluate the output at each time, comparing, for example, the number of changes in the graph (e.g., the number of removed edges) as a risk analysis metric of the the current configuration. We plan to extend our implementation and support more container engines (e.g., Podman and CRI-O), infrastructure-as-code tools (e.g., Vagrant and Terraform), as well as integrate the approach into a container orchestration tool (e.g., in Kubernetes as an admission controller).

VIII. Conclusion

In this paper, we proposed a stress-test approach for cloud configurations, by using a knowledge graph to represent deployments and permissions, and AND/OR graphs to represent vulnerabilities. We described our solution along with algorithms to unveil vulnerabilities and semi-automatically suggest security fixes to mitigate them; we also described a first implementation and evaluation on Docker containers. We believe this is the first step towards a dynamic and continuous evaluation mechanism for risk analysis in cloud environments while suggesting security fixes. In the future, more validation on scalability and query performance is needed, especially with hundreds or thousands of containers.

Acknowledgments

This work has been partially supported by the European Union under the H2020 grant 952647 (AssureMOSS).

References

[1] Sysdig, "Sysdig 2022 cloud-native security and usage report," (Accessed on: 28/03/2022). [Online]. Available: https://sysdig.com/2022-cloud-native-security-and-usage-report/ 1

[2] N. S. A. (NSA), "Mitigating cloud vulnerabilities," (Accessed on: 09/03/2022). [Online]. Available: https://media.defense.gov/2020/Jan/22/2002237484/-1/-1/0/CSI-MITIGATING-CLOUD-VULNERABILITIES_20200121.PDF 1

[3] E. E. B. Authority), "Eba will run its next eu-wide stress test in 2023," (Accessed on: 22/03/2022). [Online]. Available: https://www.eba.europa.eu/eba-will-run-its-next-eu-wide-stress-test-2023 1

[4] T. of Bits, "Understanding docker container escapes," (Accessed on: 23/03/2022). [Online]. Available: https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/ 1, 3

[5] P. A. U. 42, "New linux vulnerability cve-2022-0492 affecting cgroups: Can containers escape?" (Accessed on: 23/03/2022). [Online]. Available: https://unit42.paloaltonetworks.com/cve-2022-0492-cgroups/ 2

[6] X. Li, Y. Chen, and Z. Lin, "Towards automated inter-service authorization for microservice applications." New York, NY, USA: Association for Computing Machinery, 2019, p. 3–5. 2

[7] A. B. Fadhel, D. Bianculli, and L. C. Briand, "Model-driven run-time enforcement of complex role-based access control policies," 2018, pp. 248–258. 2

[8] C. Gerking and D. Schubert, "Component-based refinement and verification of information-flow security policies for cyber-physical microservice architectures." Hamburg, Germany: IEEE, 2019, pp. 61–70. 2

[9] A. Ibrahim, S. Bozhinoski, and A. Pretschner, "Attack graph generation for microservice architecture," ser. SAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1235–1242. 2

[10] NIST, "Vulnerability scanning," 2021, (accessed: 08/02/2021). [Online]. Available: https://csrc.nist.gov/glossary/term/Vulnerability_Scanning 2

[11] G. F. Lyon, *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. US: Insecure. Com LLC (US), 2008. 2

[12] B. Chess and G. McGraw, "Static analysis for security," vol. 2, no. 6, pp. 76–79, 2004. 2

[13] I. Pashchenko, H. Plate, S. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," vol. SE-13, no. 01, pp. 1–1, sep 5555. 2

[14] S. Dashevskyi, A. D. Brucker, and F. Massacci, "A screening test for disclosed vulnerabilities in FOSS components," vol. 45, no. 10, pp. 945–966, 2018. 2

[15] S. Noel, E. Harley, K. H. Tam, M. Limiero, and M. Share, "Cygraph: graph-based analytics and visualization for cybersecurity," in *Handbook of Statistics*. Elsevier, 2016, vol. 35, pp. 117–167. 2

[16] Y. Jia, Y. Qi, H. Shang, R. Jiang, and A. Li, "A practical approach to constructing a knowledge graph for cybersecurity," *Engineering*, vol. 4, no. 1, pp. 53–60, 2018. 2

[17] K. Zhang and J. Liu, "Review on the application of knowledge graph in cyber security assessment," in *IOP Conference Series: Materials Science and Engineering*, vol. 768, no. 5. IOP Publishing, 2020, p. 052103. 2

[18] C. Banse, I. Kunz, A. Schneider, and K. Weiss, "Cloud property graph: Connecting cloud security assessments with static code analysis," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 13–19. 2

[19] T. M. Corporation, "Containers matrix," (Accessed on: 22/03/2022). [Online]. Available: https://attack.mitre.org/matrices/enterprise/containers/ 3, 5

[20] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 418–429. 3, 5

[21] R. Sawilla and D. Skillicorn, "Partial cuts in attack graphs for cost effective network defence," in *2012 IEEE Conference on Technologies for Homeland Security (HST)*, 2012, pp. 291–297. 3

[22] O. S. Vaidya and S. Kumar, "Analytic hierarchy process: An overview of applications," *European Journal of Operational Research*, vol. 169, no. 1, pp. 1–29, 2006. 3

[23] M. Reeves, D. J. Tian, A. Bianchi, and Z. B. Celik, "Towards improving container security by preventing runtime escapes," in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 38–46. 5