Container Hardening Through Automated Seccomp Profiling

Nuno Lopes
Rolando Martins
Manuel Eduardo Correia
{nunolopes,rmartins,mdcorrei}@fc.up.pt
Faculty of Sciences, University of Porto, Portugal

Sérgio Serrano
Francisco Nunes
{sergio.serrano,francisco.nunes}@talkdesk.com
Talkdesk, USA

Abstract

Nowadays the use of container technologies is ubiquitous and thus the need to make them secure arises. Container technologies such as Docker provide several options to better improve container security, one of those is the use of a Seccomp profile. A major problem with these profiles is that they are hard to maintain because of two different factors: they need to be updated quite often and present a complex and time consuming task to determine exactly what to update, therefore not many people use them.

The research goal of this paper is to make Seccomp profiles a viable technique in a production environment by proposing a reliable method to generate custom Seccomp profiles for arbitrary containerized application. This research focused on developing a solution with few requirements allowing for an easy integration with any environment with no human intervention.

Results show that using a custom Seccomp profile can mitigate several attacks and even some zero day vulnerabilities on containerized applications. This represents a big step forward on using Seccomp in a production environment, which would benefit users worldwide.

CCS Concepts: • **Security and privacy** → **Virtualization and security**; **Software security engineering**.

Keywords: Containers, Security, Seccomp, Fuzzing

ACM Reference Format:
Nuno Lopes, Rolando Martins, Manuel Eduardo Correia, Sérgio Serrano, and Francisco Nunes. 2020. Container Hardening Through Automated Seccomp Profiling. In *Containers Workshop on Container Technologies and Container Clouds (WOC'20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429885.3429966

1 Introduction

Containers are used worldwide in several production environments. They help developers reduce the time needed to configure the required application dependencies and libraries. Given their worldwide usage the need to secure these containerized applications increases. Docker [1] is currently the standard for the container technology industry and thanks to its huge community new projects to better secure containers often arise. The trend to increase security is to develop a custom container runtime to substitute RunC [2] (default container runtime) and better isolate containers in a lower level manner. Other projects aim to secure the container before it is deployed, for example by performing image scanning which compares the version of the several technologies against a vulnerability database. This is similar to security auditing tools, which ensures that a container is running with the industry best practices and is properly configured.
Docker also provides other security options which have not been fully embraced by the community, one example is the Seccomp profile [3]. Where it is possible to create a profile consisting on several syscalls that either will or will not be allowed to execute in the system. One of the biggest challenges is that creating and maintaining an up to date Seccomp profile is complex and presents several difficulties. The first one is knowing which syscalls [4] the containers needs. These might change when a new update is available, requiring a periodic check to ensure that all the features work properly. The absence of periodic checks could cause a self inflected Denial of Service (DoS).

Seccomp can be very efficient when trying to reduce the attack vector on a container since the syscalls that a container can make are only those whitelisted. In other words, in case a malicious actor gets access to the container, by a Remote Code Execution (RCE) vulnerability for example, he/she will be limited to the syscalls that are whitelisted in the profile. If the malicious actor tries to escalate privileges or attempts a container breakout through a syscall that was not specified in the profile, the attack will not go through. This might also be helpful when dealing with some zero day vulnerabilities that need syscalls which are not in the Seccomp profile.

The novelty of our work is to combine the Seccomp security feature with the Continuous Integration/Continuous Development (CI/CD) pipeline therefore solving the challenge of maintaining Seccomp profiles up to date. In order automatically do this, the first step is to create an easy and autonomous way of capturing all the syscalls that a container performs during the unit/integration testing phase. In order to improve our solution, we made use of fuzzing as a way to improve coverage. Once a list of used syscalls is available, it is simply a matter of generating and deploying a new custom Seccomp profile.

1.1 Related Work

Research has been conducted with the aim of trying to reduce the attack vector of containerized applications. We will start by explaining how AppArmor does this, what are its drawbacks and how it compares with our work. We will then focus on the research paper "*Can Container Fusion Be Securely Achieved?*" [5], explain what it is about and how we expanded on their work. At last, we will explore other container tracing solutions.

AppArmor

AppArmor [6] makes use of a kernel module, which binds its profile to a program. The profile consists of several capabilities [7] that the program needs in order to run successfully (whitelisting of capabilities). The administrator must create an AppArmor profile and bind it to a specific program. There is a second feature in AppArmor called learning,which as the name goes it is usually enforced when the administrator does not know what capabilities are required. This mode gives access to all capabilities, and logs everything, so when the work is done the administrator can review which capabilities were called and create a custom profile tailored to a specific need. This research paper follows the same steps as the learning feature in AppArmor: allowing all the syscalls to be performed, logging and reviewing them and finally generating a new profile. In the case of AppArmor, one major disadvantage is that the whole process is not automated resulting in a slow and hard process to maintain.

Container Fusion

In previous work [5], a new idea has been presented, called container fusion, which happens when there are two different containers and one needs some privileged access on the sibling container. The access is given based on the least privilege principle [8] meaning that it will only have the required permissions to perform the needed actions therefore limiting the visibility and accessibility of the core container. The containers are isolated using a number of known techniques such as: granting access to the core container with the help of namespaces [9], capabilities, a custom Seccomp profile and hardware restrictions using control groups [10]. In order to successfully define the capabilities and syscalls required to the custom Seccomp profile, they manually determine the capabilities needed by studying the container. Afterwards they review the syscalls that each capability would allow and removed the ones that were not required, and finally could deploy the container with these custom profiles. Looking at the conclusion they propose an automated way of determining capabilities in a sandboxed environment as future work, this is where we expand on their research by automatically generating a custom Seccomp profile.

Tracing Solutions

Nowadays there are several container tracing solutions, most of them work on a high level with metrics such as: central processing unit (CPU) usage, memory utilization, network input and output among others. These solutions range from the Docker engine itself, cAdvisor [11] which creates a hook onto the Docker Daemon and gathers metrics for all of the running containers.

Tracing solutions capable of gathering low level information are quite scarce. The main tool for this is Sysdig Inspect, an open source tool capable of tracing system calls, page faults, errors, threads among others. Another low level tracing solutions is called Tracee, and even tough it is still at an experimental stage the fact that it is open source might accelerate its development process. Tracee is capable of tracing syscalls, arguments, namespaces and other actions performed by a container.

2 Architecture Overview

Our proposal is to help the community embrace the benefits of having reliable Seccomp profiles. We aim to remove Seccomp's main hurdles: the hassle to configure profiles and the difficulties to keep them up to date. Seccomp profiles require knowledge about the syscalls needed by a container, therefore the process starts by implementing a solution to trace every syscall performed by a given container.

In order to easily maintain the Seccomp profiles our tracing solution will be integrated with the Continuous Integration/Continuous Development (CI/CD) pipeline, as it is considered a best practice and a lot of companies are migrating to this environment where it is possible to consistently build and test if the application works as expected. In our research we decided to create a new step in the pipeline. As such, whenever a new feature is added, our tracing solution will capture all the syscalls that are generated by the unit/integration tests and create an updated Seccomp profile as previously described.

Successfully integrating with the CI/CD pipeline requires the tracing solution to already be running in the system. Therefore whenever new code gets pushed into the repository, this will trigger the continuous integration pipeline where Docker will build a container with that application and run the unit/integration tests, this represents the first three stages from figure 1. Since our tracing solution is already running, all the syscalls generated by the unit/integration tests will be captured and a Seccomp profile will be generated based on them. The final stage in figure 1 represents the deployment of the container with the custom Seccomp profile. This workflow is illustrated in figure 1.

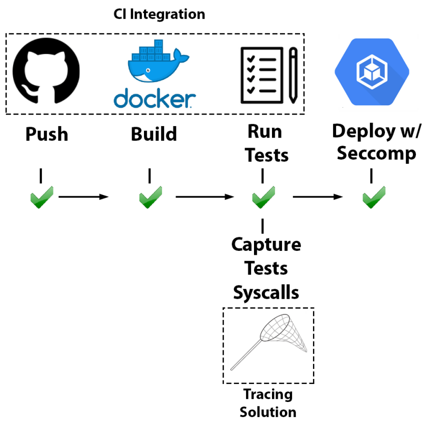

Figure 1. Proposal Workflow

The base architecture of the tracing solution consists of two components: the Container Tracing and the Container Exit Check. Obviously in order to capture the syscalls a Testing Container needs to exist, but note that this is decoupled from our tracing solution as seen in figure 2. The Testing Container represents the containerized application that will be traced and later deployed with a custom Seccomp profile.

This workflow will start as soon as a new Testing Container is spawned, since it will start to generate syscalls from a different namespace thereby alerting the Container Tracing. Which is responsible for capturing syscalls from containers, and writing them into an output file. In order to know when the container has exited the Container Tracing performs a unary gRPC [12] call to the last component, the Container Exit Check. This last component must check when a specific container has finished its execution and then respond to the gRPC call. This check is not done inside the Container Tracing since it creates significant overhead, for this reason only we decided to have a separate component responsible for this task. When all this is completed, the Container Tracing will take the capture files and generate a JSON Seccomp file whitelisting the syscalls that were traced. Upon completion, it is possible to move onto the last stage of our workflow and deploy the container assigning it the custom Seccomp profile. This architecture is illustrated in figure 2.

It may be difficult to fully cover the needed syscalls in more complex containers as there is always the possibility of a false negative occurring (a syscall that is not in the profile but should be). To respond to this challenge, a fuzzing solution is proposed in an attempt to increase the coverage of needed syscalls. Fuzzing should only be an option when

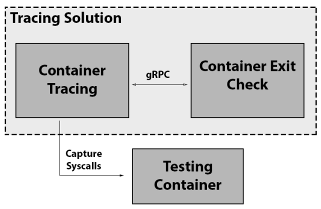

Figure 2. Tracing Solution Architecture

the unit/integration testing does not provide sufficient coverage on all of the functionalities that the container has. In the case of a container having unit/integration tests for all of the use cases then fuzzing is not recommenced since it might explore unwanted syscall paths. The fuzzing solution is based on Docker, where it is possible to use general containers capable of fuzzing web servers, sockets, APIs as well as the possibility of using a container created by the user to a specific technology. A configuration file is required to specify each fuzzing container as well as options for their creation.

3 Implementation

In this section we will explain in detail how our proposal is implemented. We start by describing how our tracing solution works, in other words, the process of capturing syscalls from a container and generate a new Seccomp profile based on them. We follow by explaining how it is possible to integrate the previously described solution with the CI pipeline and how this removes the hassle of configuring each profile. We finish with a break down about how fuzzing integrates with this whole process and when it is recommended to use.

3.1 Capture syscalls and generate a Seccomp profile

One of the first problems we encountered, was having a reliable method of capturing the syscalls. The solution here proposed is to make use of eBPF [13] tracing capabilities which creates a kernel hook on a given syscall. Tracing syscalls used by the container happens when a syscall is called, causing our eBPF program to also be called. Finally there must be a check to determine from what namespace the syscall came from. Since we are only interested in capturing the Testing Container syscalls, in order not to clutter the output all the syscalls performed from the host and by the Container Tracing itself will be ignored.

Our BCC program will create a kernel hook for each syscall in the system and then write the syscall called to the specific output file. The file with the captured data has several parameters: the time on which the syscall was called, the process that called it, the namespace and finally the syscall itself. There is the possibility of adding more parameters, these will provide more information about the environment in order to facilitate any type of data analysis.

One of the advantages of using eBPF is that it provides flexibility on which container technology to use. Since the syscalls are being captured at a kernel level this approach works on Docker, Podman [14] and even other container technologies that have not been yet developed. Nonetheless, there is the possibility of having any method do this and in case another tracing technology is preferred the output should be a file with one syscall per line. This can easily be accomplished with a tool such as Strace as long as the arguments used by the syscall are erased.

The last part to successfully capture all the syscalls from a container is to know when to stop. For this, the Container Tracing sends a unary gRPC call to the Container Exit Check. This communication is done using a Protobuf, with a string specifying the Docker container identifier. Using this information the Container Exit Check can make use of the Docker Software Development Kit (SDK) for Python and check the status for that specific container, once the status is "exited" it can respond to the gRPC call. Once the call is answered, the Container Tracing will no longer expect syscalls from that namespace and can start generating the custom Seccomp profile. To generate a Seccomp profile we take the capture file and compile a list of syscalls with no duplicates, once this is done we simply whitelist those and set a default action to block all the syscalls that were not mentioned.

Once all has been implemented, it is possible to start to trace containers and generate a custom Seccomp profiles for our needs. These profiles can be used when running a new container, simply specifying the profile location in the *–security-opt seccomp* flag using the Docker command line interface. As a proof of concept we took a vulnerable Apache container with an RCE vulnerability in Apache-Struts, CVE-2018-11776 [15], we traced the syscalls and generated a custom profile. Once the container with the custom profile was deployed, it was concluded that it was no longer possible to exploit the RCE vulnerability.

Requirements

In order to set up this environment, a few requirements must be installed in the system. The Container Tracing is the easiest to set up, since it only requires Docker to be installed. We provide a Dockerfile with all the dependencies needed, an important note is that in case you are using eBPF as a tracing method Linux 4.1 or above is necessary.

Regarding the Container Exit Check you need Python3 and some custom Python packages such as: Docker SDK and gRPC are necessary. The Docker SDK is used to check the status of a container and the gRPC is needed to answer the gRPC call from the Container Tracing.

3.2 Pipeline Integration

This proof of concept was done using GitLab CI/CD pipeline [16]. The first step is to configure GitLab runner. This is responsible for creating a container, running the unit/integration tests and sending the results back to GitLab. In order to set up the GitLab Runner there are two options: locally and remotely. The remote option makes use of GitLabs shared runner hence, all of the process will happen on their servers, since our solution requires the capture of syscalls generated this is not a feasible option. Therefore the pipeline integration must be done locally. Start by installing *GitLab-Runner* on the system, setting up the runner using the flag *register* and supplying the *gitlab-ci* token for the project. The last step would be to create a file named *.gitlab-ci.yml* in the root of the repository. This file is where the pipeline is defined.

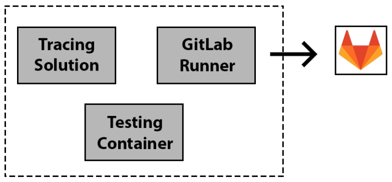

Figure 3. CI/CD Integration

From figure 3 it is clear that GitLab Runner and our tracing solution are running simultaneously, representing long term containers in the architecture, while the testing container is completely mutable and can be created and destroyed according to the pipeline.

This process results in a fully automatic environment that traces syscalls on new commits and generates a Seccomp profile based on those syscalls.

Requirements

One underlying requirement here is that whichever CI/CD technology is chosen it must have an option to perform the unit/integration tests inside a container. If this option is absent, all the syscalls will be generated by the host and will not be traced. Most of the CI/CD technologies have what is called a shell executor, which executes commands in a shell such as:*Docker build* and *Docker run.* Below is a generic example on what a *.gitlab-ci.yml* file would look like in order to build and run a container

```
build_image:
  script:
    - Docker build -t container .
    - Docker run container
```

3.3 Fuzzing

As previously stated fuzzing should only be done as a last resort when the unit/integration tests alone are unable to cover all the functionalities of the application.

The implementation is based on a configuration file named *conf.json* where the user can configure a custom fuzzing container, define domains/IPs to fuzz and use a file with known inputs that will be mutated in order to see how the application responds to malformed inputs. Our solution provides some general fuzzing containers, which have several entry points depending on the type of technology you want to fuzz. Since the fuzzing is done on a wordlist base it is important that it is rich enough in order to generate new syscalls paths, the user can even create a custom grammar to achieve a more complete fuzzing solution. All the containers defined in the *conf.json* file will be executed sequentially.

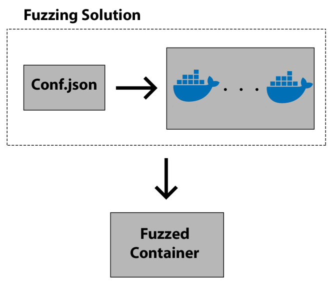

Figure 4. Fuzzing Solution Workflow

The workflow depicted in figure 4, shows how the fuzzing solutions works. An initial configuration is provided, which will fuzz a general webserver as well as some Transmission Control Protocol (TCP) connections such as sockets. The user can blindly deploy the fuzzing solution and in case any of defined vectors are available, it will fuzz the container. It is however recommended that the user provides a custom wordlist, either one for that specific technology or a more general one (good examples can be found from the SecLists repository [17]). Besides the default fuzzing containers the user can add custom containers to the configuration file thereby enabling new entrypoints for other technologies and protocols. Fuzzing should be conducted with caution to make sure unwanted code paths are not explored. This entails understanding the container and protocols that are being fuzzed as well as the type of wordlist that is being used.

Requirements
Since several containers are being deployed to perform fuzzing it is only required that Docker and the Docker SDK for Python are installed in order to communicate with the Docker Daemon.

4 Evaluation

In this section, the security posture of our proposed solution is evaluated. Starting by identifying some of the drawbacks, followed with an overhead analyses during the continuous integration phase. Finally understanding what steps to take in order to determine how a current Seccomp profile behaves to a specific vulnerability.

Drawbacks

In order to successfully synchronize the Seccomp profile with the application, extensive and reliable unit/integration tests are required. In the best case scenario they will cover all of the application functionalities including how the application responds to malformed inputs. If this is not the case the user might get overconfident about the profile therefore having a false sense of security. In the case of having an incomplete profile the consequences will be that some functionalities of the application will not work as expected or in the worst case, not work at all resulting in a self-inflicted DoS, hence the need for strong unit/integration tests.

Fuzzing

When the application does not have a complete set of unit/integration tests a fuzzing solution is proposed. However fuzzing does not guarantee that the full length of needed syscalls will be covered. Therefore if the user relies completely on the fuzzing solution and does not try to implement effective unit/integration tests it can result in a self-inflicted DoS. Fuzzing should be seen as a complement to the tests where high entropy inputs are supplied to the several functionalities.

Overhead

During the continuous integration phase, all syscalls performed by the containers will be traced, thereby creating some overhead. Container benchmark tests took place in order to better understand how much this phase will be affected. These are divided into CPU intensive operations, network throughput and disk I/O.

These benchmarks were performed ten times for each type, all the tests were conducted on google cloud with an n1-standard-1 machine which has 1vCPU and 3,75 GB of memory with Docker version 19.03.8. The average of the collected results as well as the standard deviation are presented in the following table.

Type	W/ Tracing Sol. (s)	W/O Tracing Sol. (s)
CPU	50.726 σ=0.4877	25.501 σ=0.4307
Network	60,318 σ=0.1856	30.434 σ=0.5475
I/O	20.92 σ=0.2786	17.82 σ=0.4717

Table 1. Overhead table

It can be concluded that while tracing syscalls, both the CPU and network intensive task will be approximately 2x slower, on the other hand, disk I/O does not present significant overhead.

Mitigation

In the case of our current Seccomp profile having syscalls that can be used to exploit a container, the currently proposed solution identifies the dangerous syscalls and offers the option to remove them from the Seccomp profile. Removing a syscall from a Seccomp profile should be done with extra caution since it may damage some functionalities. In order to prevent this, all the functionalities should be tested beforehand to ensure that everything is working as expected.

In order to successfully mitigate a vulnerable Seccomp profile, firstly it is necessary to trace the syscalls from a working proof of concept of the vulnerability. Using our tracing solution described in section 4.1, it is possible to trace and generate a Seccomp profile based on the syscalls used by the proof of concept. Secondly it is necessary for the user to determine which syscalls are needed by the container, which can be done with the help of the unit/integration tests or by the user manually interacting with each needed functionality. This results in two temporary profiles, one that maps the syscalls used by the exploit and the other maps syscalls needed by the application. It is then possible to use our program to check which syscalls are in the profile of the proof of concept and which are not in the other profile. The syscalls returned will be the ones used by the exploit, removing these syscalls from our profile is a temporary mitigation measure to the vulnerability and should be a last resort solution.

Using the previously described Apache container with an RCE vulnerability (CVE-2018-11776) the exploit was analysed and a group of dangerous syscalls was identified. Removing these syscalls from the Seccomp profile resulted in mitigating the vulnerability. Several tests using containers with different vulnerabilities were conducted, such as: CVE-2019-11043 [18] and CVE-2016-10033 [19] which resulted in not only mitigating the vulnerability but also having a better understanding of the exploit. We provide an example on how this works in practice. Where poc.json represents a Seccomp profile mapping the syscalls from the proof of concept container and usecase.json maps the syscalls required for the container to run normally.

```
python3 mit.py −poc poc.json −useC
usecase.json
Vulnerable syscalls: statfs vfork select
```

5 Acknowledgments

This work was supported by project COP POCI-01-0247-FEDER-039598; Rolando Martins was partially supported by project EU H2020-SU-ICT-03-2018 No. 830929 CyberSec4Europe.

6 Conclusion

Correctly using a custom Seccomp profile is arguably a more secure approach as it limits the attack surface of a container. We have demonstrated that this can be achieved and incorporated within the CI/CD pipeline thereby removing the hassle of maintaining these profiles up to date. Results have showed that using a custom Seccomp profile can mitigate several vulnerabilities that require syscalls which are not present in the profile.

We believe that bringing attention to a new way of using these profiles is the first step to having a more secure environment by making Seccomp a viable technique. At the moment the cost of using our solution is approximately 2x slower. But this overhead is only relevant in the unit/integration tests therefore only this phase is affected.

References

[1] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[2] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[3] J. Corbet, "Seccomp and sandboxing," *LWN. net, May*, vol. 25, 2009.

[4] M. Caceres, "Syscall proxying-simulating remote execution," *Core Security Technologies*, 2002.

[5] S. Suneja, A. Kanso, and C. Isci, "Can container fusion be securely achieved?," in *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, pp. 31–36, 2019.

[6] P. Δήμου, "Automatic security hardening of docker containers using mandatory access control, specialized in defending isolation," 2019.

[7] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1205–1220, 2019.

[8] T. E. Levin, C. E. Irvine, and T. D. Nguyen, "Least privilege in separation kernels," in *International Conference on E-Business and Telecommunication Networks*, pp. 146–157, Springer, 2006.

[9] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, vol. 1, pp. 101–112, Citeseer, 2006.

[10] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," *Haifux, May*, vol. 186, 2013.

[11] J. Makai, "New solutions in it monitoring: cadvisor and collectd," tech. rep., 2015.

[12] X. Wang, H. Zhao, and J. Zhu, "Grpc: A communication cooperation mechanism in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 3, pp. 75–86, 1993.

[13] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–8, IEEE, 2018.

[14] L. Heinrich, T. Maeno, A. Forti, and P. Nilsson, "Continuous analysis–user containers on the grid," tech. rep., ATL-COM-SOFT-2019-015, 2019.

[15] "Cve-2018-11776." https://nvd.nist.gov/vuln/detail/CVE-2018-11776. (Last accessed: 14.04.2020).

[16] J. M. Hethey, *GitLab Repository Management*. Packt Publishing Ltd, 2013.

[17] "Seclists." https://github.com/danielmiessler/SecLists. (Last accessed: 19.05.2020).

[18] "Cve-2019-11043." https://nvd.nist.gov/vuln/detail/CVE-2019-11043. (Last accessed: 14.04.2020).

[19] "Cve-2016-10033." https://nvd.nist.gov/vuln/detail/CVE-2016-10033. (Last accessed: 14.04.2020).