System call filtering for container security

ICT Risk Assessment Lecture Exam

Student: Emiliano Sescu

Professor: Fabrizio Baiardi

12/02/2024

UNIVERSITÀ DI PISA

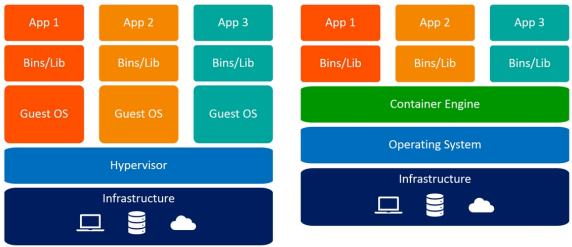
Index

Containers Vs Virtual Machines

Linux containers are primarily used for running large-scale microservice based applications in a cloud environment.

In contrast to traditional *Virtual Machines*, containers provide a way to virtualize an OS so that multiple workloads can run on a single OS kernel.

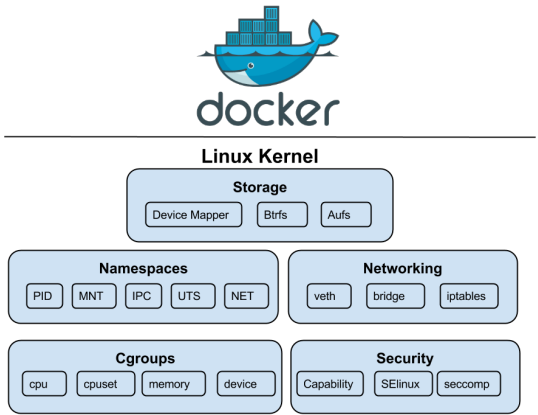
This makes containers very efficient and scalable, but could also endanger the security of the OS kernel.



Virtual Machines Containers

Despite the use of strict software isolation mechanisms provided by the OS, such as *namespaces*, *cgroups* and *capabilities*, a malicious tenant can leverage kernel vulnerabilities to bypass them.

Seccomp (Secure computing) is a sandbox tool in the Linux kernel to restrict a process from invoking certain system calls.



Lecture content

This lecture aims to answer the following two questions:

- How it's possible to craft a well-balanced *seccomp profile*?
- Are there any other techniques to stop adversaries from exploiting kernel vulnerabilities through system call invocations?

The tools that will help us to answer those questions are the following ones:

- **Confine** employs an automated technique (using static code analysis) for generating restrictive system call policies for arbitrary container.
- **SPEAKER** uses a hybrid container profiling approach that leverages a multi-level static and dynamic analysis methodology to determine the required system calls.
- **NIMOS** performs a combination of static and dynamic analyses of exploit codes in an automated way and investigated the existence of such commonly occurring system call sequences.

Index

2 Confine

Design

2 Confine

By relying on static code analysis, **Confine** inspects all execution paths of the containerized application and all its dependencies, and identifies the superset of system calls required for the correct operation of the container.

Confine operates in **three different stage**:

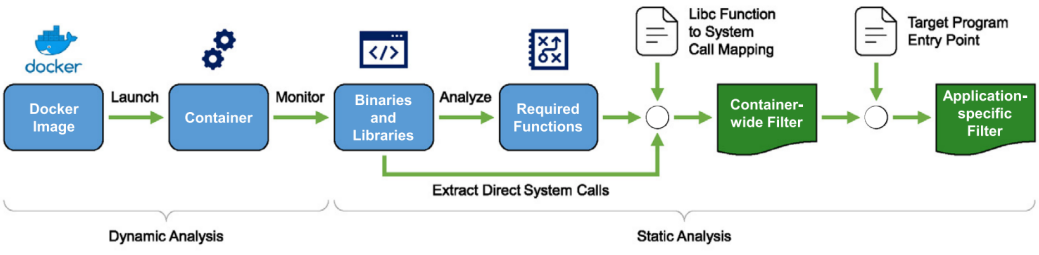
1. It first generates a *container-wide system call filter* that is applied to all programs launched in the container.
2. Then, it creates an *application-specific system call filter* that removes all system calls needed solely during the initialization phase of the container.
3. In its final step, Confine further restricts the remaining system calls needed by the main program by *limiting their argument values*.

System calls identification

2 Confine

The required steps for system calls identification are:

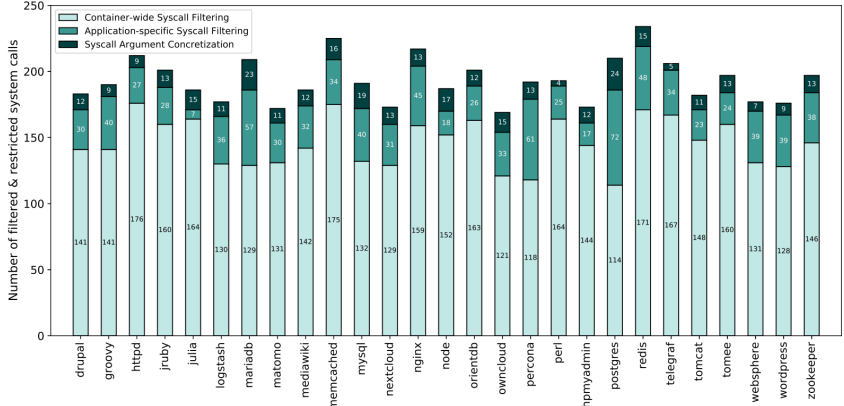
1. Identify all applications that may run on the container.
2. Identify all library functions imported by each application.
3. Map library functions to system calls.
4. Extract direct system call invocations from applications and libraries.
5. Extract hardcoded argument values for identified system calls.



Results: Seccomp profiles of 27 Docker images

2 Confine

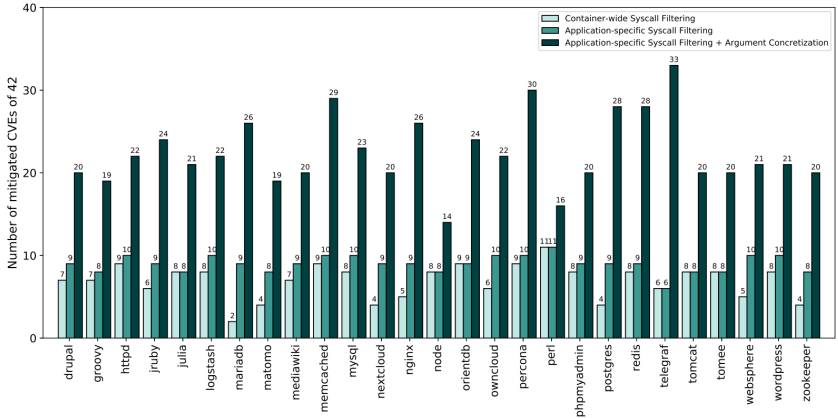
Number of disabled system calls by container-wide (bottom part) and application-specific (middle part) filtering, and of argument restricted system calls (top part):



Results: Mitigated CVEs for 27 Docker images

2 Confine

Given a set of 42 CVEs (extracted from `www.cvedetails.com`), it's possible to notice how the addition of *argument concretization* offers a significant improvement:



Benefits and Drawbacks

2 Confine

Benefits:

- Generate restrictive system call filters and enforce them using a ready-to-use technique
- Non-intrusive approach

Drawbacks:

- Only records dynamically loaded libraries during the initial stages of the container execution
- Relies on the user to provide informations about the image
- Docker images analyzed by Confine must be compatible with Angr for extracting the CFG of the invoked programs and its libraries
- Suffers from overestimating reachable system calls or missing true ones
- Confine strives to only restrict flags and constant arguments
- Confine proxy program itself requires a limited set of system calls to install the application-specific filter

Index

3 SPEAKER

Design

3 SPEAKER

SPEAKER consists in an hybrid approach to limit the system calls usage: given an application container, a whitelist extracted via *static analysis* is enforced along with a complementary whitelist generated with the *dynamic profiling* of the Docker image.

This method automatically analyzes the container behavior to identify **three execution phases** (*booting*, *running*, *shutdown*) and dynamically enforce the corresponding fine-grained system call whitelists.

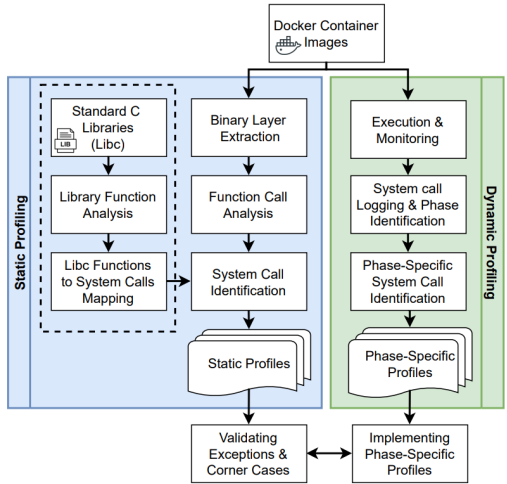
The final seccomp profile for each execution phase is determined by both results of the lists obtained with static and dynamic analysis:

- If a process invokes a syscall out of the static and dynamic whitelists, it will be killed
- If the invoked system call is found only in the static profile, it will be logged for further inspection
- If the invoked system call is found within the dynamic profile, it will be normally executed

Multi-level approach

3 SPEAKER

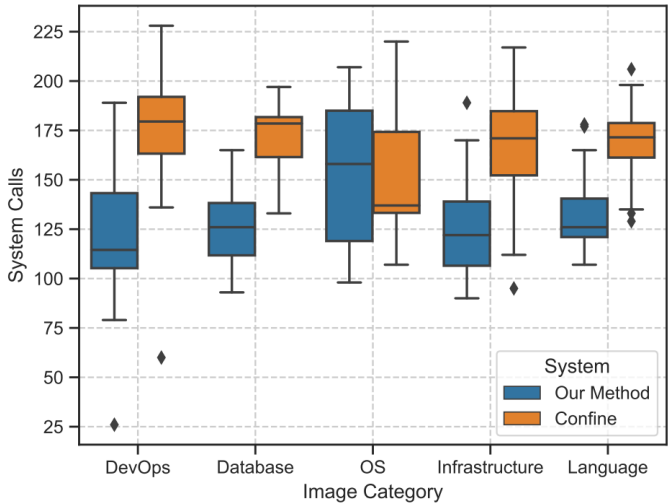
As illustrated in the figure on the right, the implemented multi-level approach consists of two main components: **static profiling** and **dynamic profiling**.



Results: Static profiling of 150 Docker images

3 SPEAKER

Comparison with Confine over the required system calls for every image category:



Experimental results for database and web server containers

3 SPEAKER

For the vulnerability reduction study, 92 *CVE entry-related* (extracted from `cve.mitre.org`) and 21 *high-privileged system calls* were taken into consideration.

Number of identified system calls during booting, running, and shutdown execution phase:

	Image Name	Unique Syscalls (Static/Dynamic)	Boot.	Run.	Shutd.
Database	MySQL	150/109	106	47	32
	Postgres	120/101	97	53	31
	Mongo	146/103	96	48	27
	Redis	127/77	69	27	22
Web Server	Wiki.js (node.js)	141/82	70	70	8
	Dokuwiki (nginx)	145/114	73	110	52
	MediaWiki (Tomcat)	132/101	101	88	45
	XWiki (httpd)	134/105	74	82	16
	phpBB (httpd)	134/101	92	91	14

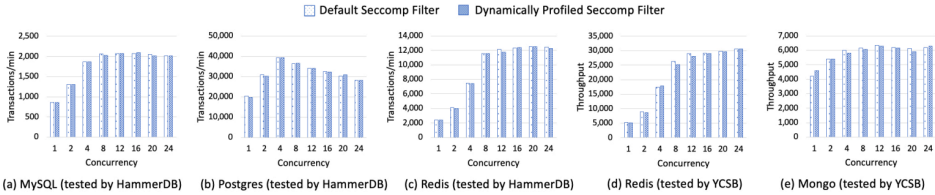
Number of required/removed vulnerable syscalls related to CVEs/high-privileged functionalities:

	Image Name	Required System Calls			Removed System Calls		
		Total	CVE	H.P.	Total	CVE	H.P.
Database	MySQL	47	17	1	301	75	20
	Postgres	53	22	3	295	70	18
	Mongo	48	21	2	300	71	19
	Redis	27	12	1	321	80	20
Web Server	Wiki.js	70	27	4	278	65	17
	DokuWiki	110	43	13	238	49	8
	XWiki	88	32	7	260	60	14
	MediaWiki	82	33	5	266	59	16
	phpBB	91	35	5	257	57	16

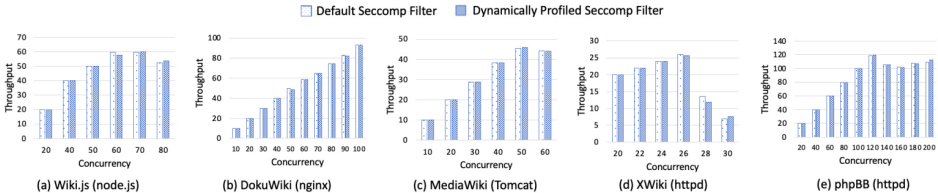
H.P. = high-privileged

Performance overhead for database and web server containers

3 SPEAKER



Performance overhead for the Database containers.



Performance overhead for the Web Server containers.

Benefits and Drawbacks

3 SPEAKER

Benefits:

- Better static profiling technique
- Non-intrusive approach
- Dynamic profiling reduce overestimation of required system calls of the containerized apps that are based on interpreters
- Leverages Dynamic profiling with a reasonable performance overhead

Drawbacks:

- Dynamic profiling requires testing tools and benchmarking datasets
- Limitations of static analysis
- Sanity checking on the parameters of system calls is not performed
- The system calls required only for the container initialization are not removed from the static profile
- The timing-based method used for phase identification requires some extra system calls
- A SPEAKER process runs outside the containers to enforce dynamic profiles

Index

4 NIMOS

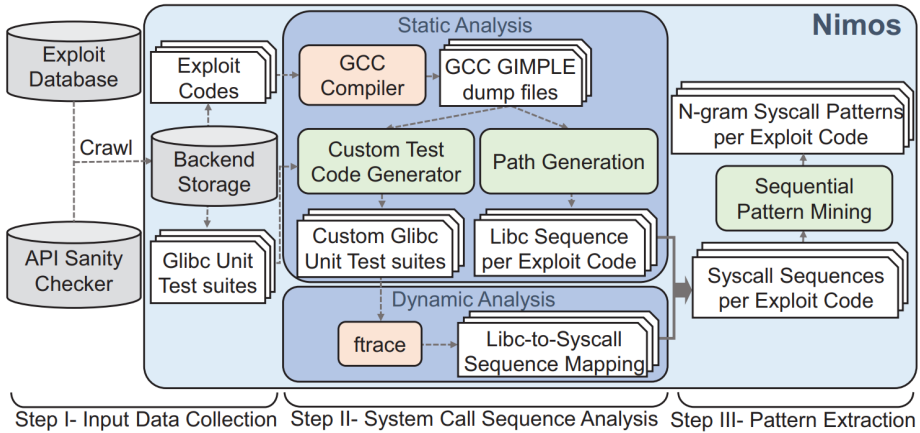
NIMOS performs a combination of static and dynamic analyses of exploit codes in an automated way and investigated the existence of such *commonly occurring system call sequence*.

Common system call sequence patterns across the different exploits are then *mined*, and their effectiveness for sequence-based filtering is analyzed.

Design for sequence extraction

4 NIMOS

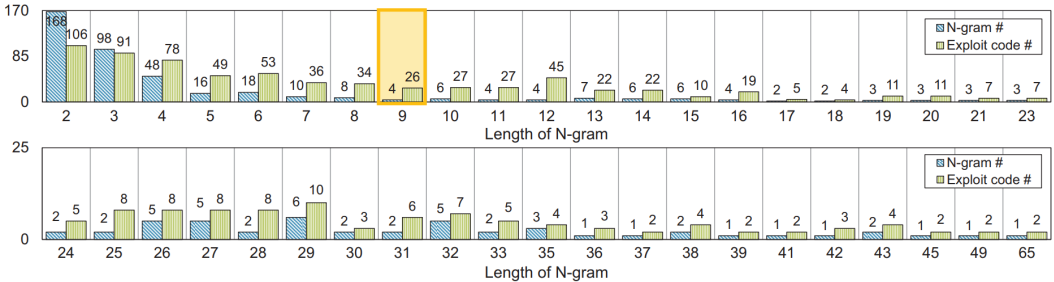
Architecture of NIMOS and processes for system call *sequence extraction and analysis*:



Results: Syscall Sequence Patterns across Exploits

471 *system call sequence patterns* were found across 106 exploit codes (extracted from `www.exploit-db.com`).

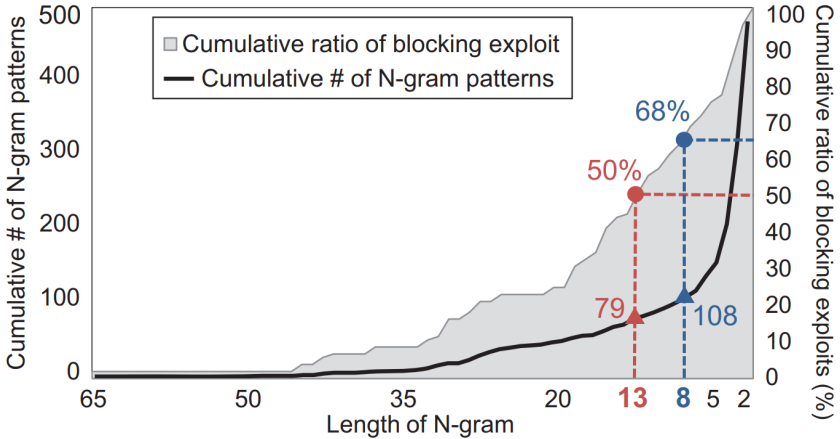
The figure below shows a distribution of the number of patterns discovered and the number of exploit codes matched, for each pattern length:



Results: Syscall Sequence Patterns across Exploits

4 NIMOS

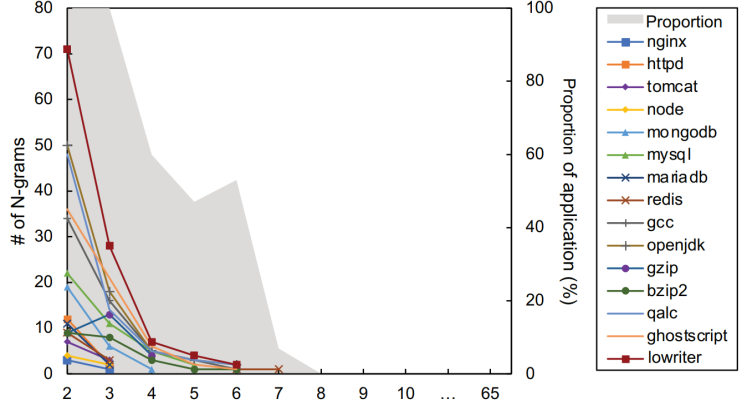
Cumulative exploit coverage as N-grams is collectively used in the range of length 65 and downward:



Analysis of Benign Syscall Sequence

4 NIMOS

Given a set of 15 popular applications, the figure below shows the number of N-gram syscall sequence patterns from exploit codes found in application traces and the proportion of applications having N-gram syscall sequence patterns from exploit codes:



The second column of the table (Ex.) below shows the number of exploit codes that cannot be mitigated even if Confine's seccomp profiles are in place. The third column (Mitigable Ex.) shows how many of the exploits in the second column can be mitigated using syscall sequence patterns.

# of Images	Ex.	Mitigable Ex.	# of Images	Ex.	Mitigable Ex.
1	44	38 (86.4%)	3	14	11 (78.6%)
2	33	27 (81.8%)	17	14	10 (71.4%)
1	20	16 (80%)	1	13	11 (84.6%)
2	19	15 (78.9%)	2	13	10 (76.9%)
4	18	14 (77.8%)	21	13	9 (69.2%)
1	17	14 (82.4%)	3	12	9 (75%)
39	17	13 (76.5%)	2	12	8 (66.7%)
20	16	12 (75%)	2	11	8 (72.7%)
7	15	12 (80%)	1	11	7 (63.6%)
18	15	11 (73.3%)	2	9	6 (66.7%)

Benefits and Drawbacks

Benefits:

- It is possible to pair NIMOS with other tools that enchance seccomp security profile of the container
- Good results for attacks that can still be performed with non-filtrable system calls

Drawbacks:

- Extending the implementation with an algorithm to enable online tracking of system call sequences using seccomp is a challenging problem yet to be solved
- The impact on the performance of these algorithms in a container environment is not yet known
- Challenges of mimicry attacks

System call filtering for container security

Conclusions