Can Container Fusion Be Securely Achieved?

Sahil Suneja, Ali Kanso and Canturk Isci
IBM Research
NY, USA
{suneja, akanso, canturk}@us.ibm.com

***Abstract*—Linux containers are key enablers for building microservices. The application's microservices fall broadly under two categories, the core-microservices implementing the business logic and the utility-microservices implementing middleware functionalities. Such functionalities include vulnerability scanning, monitoring, telemetry, etc. Segregating the utility-microservices in separate containers from the core-microservice containers may prevent them from achieving their functionality. This is due to the strong isolation between containers. By diffusing the boundaries between containers we can fuse them together and enable close collaboration. However, this raises several security concerns, especially that the utility-microservices may include vulnerabilities that threaten the entire application. In this paper, we analyze the different techniques to enhance the security of container fusion and present an automated solution based on Kubernetes to configure utility-microservices containers to fuse with core-microservices containers.**

***Keywords*—Linux containers, security, orchestration, Kubernetes, microservices**

I. Introduction

The last few years have witnessed a widespread adoption of the microservices paradigm for architecting cloud-native software [1]. A common design pattern in microservices is the separation of concerns wherein each microservice is developed and packaged individually. Linux containers are the predominant model of deployment for microservices where each microservice is packaged as a container image that is later instantiated as a container instance. The inter-communication between microservices is commonly achieved through well-defined interfaces, mainly using a RESTfull API (Application Programming Interface) [2]. However, not all microservices can achieve their functionalities through high-level APIs. As an example, we consider a vulnerability scanning microservice, which we refer to as a *utility-microservice*. The utility-microservice performs vulnerability analysis against other microservices that are delivering business logic functionalities. We refer to these microservices as the *core-microservices*. In this example, the utility-microservice scans the filesystem of a core-microservice container looking for vulnerable packages installed, and thereafter checks the running processes within the core-microservice container to make sure no vulnerable package is executing. The core-microservices are typically not aware of the existence of the utility-microservices. The utility-microservices may originate from a third party vendor/provider. In fact, the utility-microservice pattern is frequently used as a means to add functionality to a legacy application that cannot be modified. For instance, a utility-microservice can intercept and encrypt the traffic emitted by a legacy application that does not support encryption. For example, when using Istio [3] to manage the communication between our containerized microservice, a utility-microservice, mediates inbound and outbound communication to the core-microservice. In order to achieve their functionality, utility-microservices often need intrusive access and capabilities in the core-microservice container. Such access can be granted via injecting the utility-microservice into the same container as the core-microservice [4]. However, this is a risky approach, since the utility-microservice may contain malicious code or bugs that can jeopardize our core-microservice and potentially our entire system. In reality, the official CVE (common vulnerability exploit) entries reveal several vulnerabilities in sidecars with utility-microservice functionalities leaking sensitive information (such as application login credentials) [5], enabling arbitrary command execution [6], causing Denial of Service (DoS) [7], amongst other security concerns. An alternative approach is to have the *utility-microservice* execute in its own container in order to isolate it from our core-microservice. However, this restrictive approach will deprive the utility-microservice of vital information it needs to carry out its functionality. For instance, our vulnerability scanning utility-microservice requires access to the memory, processes, disk, and network state of the core-microservice container, all of which are isolated by default among two separate containers.

In this paper, we propose a different approach. Our approach leverages the isolation benefits of having the two microservices executing in different containers. By diffusing certain boundaries between the two containers, we grant the utility-microservice the capabilities and permission it needs to perform its functionality. All while protecting our core-microservice by limiting the visibility and accessibility of our utility-microservice. We refer to this approach as container *fusion*.

The novelty of our work stems from the unique combination of permissions, capabilities, and constraints that we apply to our sidecars without modifying the kernel while maneuvering within the boundaries of abilities of the available kernel constructs. We couple our design with an implementation that makes the fusion transparent to the user who can apply it without having to understand its underlying complexity.

The paper is organized as follows, in Section II we present the background to our work. In section III we discuss the container fusion design. In Section IV we evaluate our design from a security standpoint. In Section V we present our implementation that automatically configures the sidecar containers, and finally, we discuss the related work in Section VI and conclude in Section VII.

II. Background

In our approach for securing our core-microservice we follow the principle of least privilege [8] for our sidecar, starting with running the sidecars as *unprivileged* entities, then giving them access rights via *namespaces* and *capabilities*, and finally restricting their impact abilities via *seccomp* [9] and *netfilter* [10]. In addition to these five

WOC '19, December 9–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7033-2/19/12…$15.00
https://doi.org/10.1145/3366615.3368356

constructs for access controls, we also leverage cgroups to enforce resource constraints on the sidecar sandbox. In this section, we give a brief overview of the above-mentioned constructs.

A. *Privileges and Capabilities*

Unlike privileged processes, unprivileged processes are subject to full permission checking based on the process's credentials. The privileges traditionally associated with superuser are known as capabilities, and such capabilities can be independently enabled and disabled for unprivileged processes. Such capabilities include changing files ownership or overriding disk quota limits among others [11].

B. *Namespaces and Cgoups*

The Linux operating system provides global system resources that are shared among the processes it manages. Such resources include the Network devices, TCP/IP stacks, Posix message queues, etc. A namespace wraps a global system resource in an abstraction. Processes within a namespace execute in an isolated instance of the global resource. As such, changes to the global resource are visible to processes that are members of the same namespace but are invisible to other processes. Namespaces are the backbone of Linux containers [12]. Isolating processes in namespaces does not limit them from consuming more than their fair share of system resources such as CPU and memory. This is where cgroups come into play. Cgroups are the mechanism by which processes resource utilization is constrained.

C. *Seccomp and Netfilter*

Seccomp (short for secure computing mode) is a mechanism to constrain processes from making unauthorized system calls. Should a process attempt to make a forbidden system call for that process (e.g. exec or fork), it will get terminated. Netfilter provides the functionality required for directing packets through a network and prohibiting packets from reaching sensitive endpoints within a network.

III. Container Fusion Design

The sidecar container requires certain capabilities and privileges in the core-container. Yet, it cannot be fully trusted. Hence the containers boundaries need to partially be relaxed but with certain constraints enforced.

A. *Threat Model*

Our goal is to restrict the sidecar's operational environment enough so that it can not do anything malicious. This includes direct impact such as process corruption, as well as indirect interference such as botnet or fork-bomb behavior. For the purpose of data collection, we are okay with giving the sidecar full read-only (R/O) visibility into a core container's state, including possibly any secret keys in there. Multiple instances of the same sidecar may exist at the same time. The adversary can author multiple sidecars that can operate against the same core container. The adversary can also run their sidecar container (as a regular unprivileged cloud tenant) on the same host as the target core container.

B. *Container Fusion Design*

The challenge in securely running untrusted sidecars that perform system-state-extraction lies in balancing the levels of accessibility and constraints afforded to them. We basically need to prevent the sidecars from:

1. impacting the core container execution: this includes direct impact such as process subversion, as well as indirect interference such as DoS via resource-hogging,
2. communicating with the outside world: leaking sensitive information or acting as a botnet, and
3. leaking information to host-local accomplices (e.g. other malicious containers).

To address these demands, the sidecars are isolated away in separate containers, however, the default strong boudaries between containers prohibit them from being able to look inside the target core-container. Thus, we need to provide these sidecars secure access to privileged resources outside the walls of a typical container, specifically–the target core-container's memory, disk, and network state as shown in Fig 1. However, what adds to the complexity of the problem is that some capabilites that we give to the sidecar container can be too powerful, and may be dangerous to the core-container, therefore in many cases, as we will see, we have to cirlce back and block certain aspects of a capability to mitigate its powers. The steps involved in achieving secure container fusion are as follows.

(1) ***Namespace Isolation***

By default, the sidecar has access into its own set of namespaces, separate from the core-container. It can thus only view its private set of processes, mount-points, network devices, user/group IDs, and inter-process communication objects. This isolates the sidecar away and prevents it from having any communication with entities outside its container. This includes communication via the filesystem, the network, or through inter-process communication mechanisms such as shared memory or message queues. We start with full isolation and then share a subset of the namespaces in the next steps.

(2) ***De-privileging***

Next, following the principle of least privilege [8], the sidecar is made an unprivileged entity, by mapping its user ID inside its container to a non-root user ID on the host. This takes away a substantial amount of power from the sidecar to alter any host system state.

(3) ***Resource Isolation***

The sidecar has its own cgroup for resource isolation. Limits can thus be enforced on the sidecar's process count, CPU, memory, and disk usage, preventing it from indirectly impacting the core-container and the host.

(4) ***Access to disk state***

Isolated in its own mount namespace, the sidecar cannot see the core-container's disk-level system state such as configuration files, logs, package databases, etc. Thus, to be able to access this state, the core-container's root filesystem (rootfs) is mounted read-only inside the sandbox. But, since the sidecar's and the core-container's user IDs are different, the sidecar may be unable to read the core-container's files in the mounted rootfs, because of discretionary-access-control (DAC) permission checks. We, therefore, grant the sidecar container CAP_DAC_READ_SEARCH capability to see core-container's files (only reads, no writes).

The sidecar expects to find the relevant files at paths relative to the root directory ('/'), as in the case of as sidecar that does package analysis, it will look for packages in predefined directories (e.g. /bin). However, the mounted rootfs will have a different path depending on the mounting point. Thus, we leverage the *chroot* mechanism so that the sidecar and any imported libraries can work as-is, believing they are operating on the core-container's root directory: '/'. Nevertheless, as an unprivileged user, the sidecar can not call the chroot syscall, therefore, we also grant the CAP_ SYS_CHROOT capability to the sidecar to enable this view change during data collection.

(5) ***Access to memory state***

Similar to the mount namespace restriction, since the sidecar has its separate PID namespace, it cannot see the core-container's process state. We, therefore, share the core-container's PID namespace with the sidecar, giving it access to the core-container's memory state via procfs.

However, despite the read-permission-check exception granted in Step 4, the sidecar can still not see the files or sockets opened by the core-container processes. Basically, reading a process open files or sockets, by dereferencing /proc/<pid>/ fd/* symlinks, requires that the read-only ptrace access mode PTRACE_MODE_READ be set, amongst other flags. The lack of a mechanism to grant only this limited credential to a userspace entity made us resort to granting the more powerful Linux capability CAP_SYS_PTRACE capability to the unprivileged sidecar. Section C describes this issue in more detail.

Since the *ptrace* capability is too powerful (giving the sidecar the ability to kill/hang/corrupt core-container processes), we use seccomp to block the 'harmful' system calls which this capability enables– ptrace() and process_vm_writev(). Thus, by blending the ptrace capability with seccomp, we achieve our goal of giving the sidecar just enough power to only read, and not impact, the core-container's memory state.

(6) ***Access to network state***

Being in its own network namespace prevents the sidecar from seeing the core-container's network connections. To enable reading such a state, the core-container's network namespace is shared with the sidecar.

Although access to the core-container's network namespace does not allow packet manipulation, it does open up two avenues of nefarious actions by the sidecar. First, although the unprivileged sidecar cannot disrupt the core-container's network connections, it can use it to communicate with the outside world– create backdoors, steal secrets, act as botnet, etc. To avoid this, *netfilter* based packet filtering is employed to block a sidecar's access to the outside world, except for "white-listed" secure communication channel to ship out collected data to the monitoring backend.

The second concern is a potential DoS attack by the sidecar, where the sidecar can hoard all of the unprivileged network ports the core-container has access to. This can then prevent a core-container application to communicate via the network, if it hasn't already bound to its desired port. To resolve this, we resort to using *SElinux* [13] to allow only a few ports for the sidecar to bind to.

(7) ***Access to resource stats***

Since the sidecar is run in a separate cgroup for resource isolation, in order to gather the core-container's resource usage stats, the sidecar is also granted access to the core-container's cgroup filesystem. DAC settings ensure read-only behavior by default.

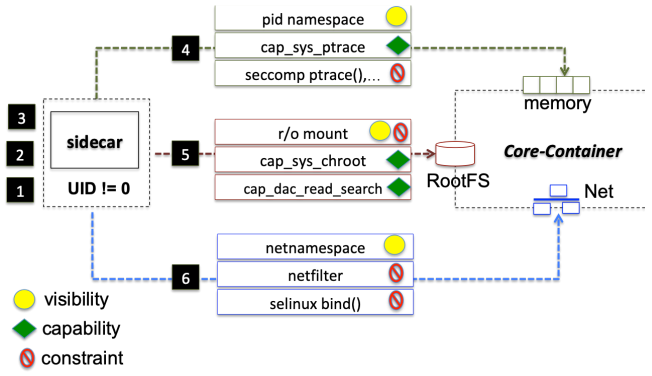

Fig. 1. Design Overview

C. Discussion

Certain reads from a sidecar to the target core-container's memory state, need a less-powerful read-only access mode set-PTRACE_MODE_READ. However, there is no direct mechanism to grant just these credentials to a userspace entity. Thus, as the nearest alternative, we had to grant the more-powerful CAP_SYS_PTRACE capability to the unprivileged sidecar. This is indeed an overkill and an artifact of coarse-grained capabilities in the Linux kernel vs. Capsicum's [14] fine-grain capabilities. This leads to bigger problems since CAP_SYS_PTRACE enables PTRACE_MODE_ATTACH thereby allowing 'write' operations, enabling the ability to kill and corrupt core-container processes. Also, the *Commoncap* Linux Security Module (which is always invoked in the kernel) does not distinguish between PTRACE_MODE_READ and PTRACE_MODE_ATTACH.

To counteract this power, we use *seccomp* to block the 'harmful' system calls which this capability enables–ptrace() and process_vm_writev(). Read-only syscalls- kcmp(), pro cess_vm_readv() and get_robust_list() do not need to be blocked. Another dangerous syscall enabled by CAP_ SYS_PTRACE is move_pages(), but for that syscall to go through CAP_SYS_NICE is needed, which we don't provide.

Beyond access via *syscalls*, the capability in question also grants power to the sidecar to access the core-container's memory directly via /proc/<pid>/mem. However, DAC controls come to the core-container's rescue in this case, preventing the sidecar from being able to write to the core-container. Writes in this fashion would additionally require CAP_DAC_OVERRRIDE to be granted to the sidecar, while we only confer the read-only capability CAP_DAC_READ_SEARCH in Step 4 of Section III.B.

IV. Implementation

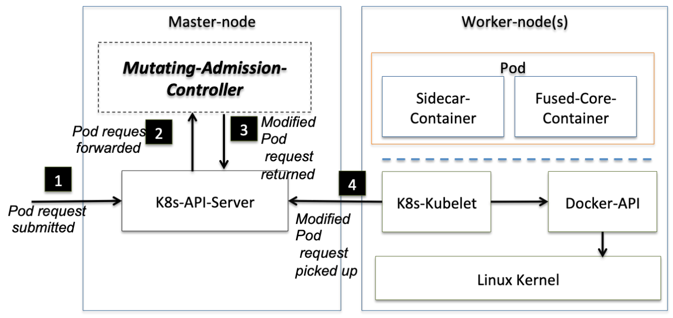

Fig. 2. Implementation Overview

For our implementation, we chose Kubernetes [22] as our container orchestration framework, coupled with Docker [23] as our container engine (leveraging ContainerD and RunC). Our fusion approach, being based upon fundamental kernel-level constructs, is applicable to the other container engines as well and can also be enforced through other container orchestration frameworks such as Docker Swarm. We did not need (or want) to modify the Linux kernel or Docker; we built our fusion solution with already-exported kernel functionality. We leveraged the Kubernetes *shareProcessNamespace* and *capabilities* features to configure our container, we also use the resource *limits/requirements* to bound our sidecar. We also leverage *network-policies* and *pod-security-context* to configure our containers. In order to keep our approach transparent to the user, we define a Kubernetes *mutation admission controller* that can (1) intercept a request to create a Kubernetes Pod*. (2) Modify the

* In Kuberentes, a Pod is a grouping of one or many containers that share the same network namespace, and lifecycle.

specification of the pod by adding our configuration options that fuse our core-container with the sidecar. (3) Resubmit the modified pod request to Kubernetes to be created. The admission controller only modifies pods that have special *labels* specifying the container to be fused: *<fuse-container-source=...>, <fuse-container-target=...>*. All other pods missing those labels will be ignored by our admission controller. Due to the lack of space, we leave out certain implementation details and modifications that were needed to Kubernetes' Kubelet− the agent running on each cluster node and invoking the container engine. Figure 2 illustrates our solution.

V. Evaluation

In this section, we evaluate the security posture of our container fusion and the performance overhead. The assumption here is that the sidecar may contain malicious code that an adversary can leverage to compromise our core-container. We consider the alternatives to our approach (i.e. using fused containers) is to (1) run the utility-microservice in the same container as our core-microservice or (2) directly on the host with enough privileges to access the core-container state.

A. *Security Analysis*

Selection of Exploits. In order to test the efficacy of our approach, we considered a comprehensive set of attack categories and verified the inability of the exploits to impact the core-container. We focused on the categories from the Exploit Database – a public archive of exploits used by penetration testers [15]. These include: local & privilege escalation, denial of service (DoS), remote exploits, as well as web application exploits. We also considered all of the attack categories from the popular Hansman and Hunt's attack taxonomy [16], namely: virus, worms, trojans, buffer overflows, DoS, network attacks, password attacks, information gathering, information corruption, information disclosure, service theft and subversion attacks, combined across all of the attack classification dimensions of the taxonomy.

The first three columns of Table 1 shows how the attack vectors we consider (column 1), to portray possible avenues of attack specific to a cloud monitoring setting and covering each of the above categories, map to them (column 2,3). An attack may map to several categories; only the closest matches are indicated in the Table, restricted by space. Actual reported vulnerabilities in data collecting sidecars do indeed fall under the attack vectors coverage, such as arbitrary command execution [6] and DoS [17], as do attacks against the software core such as privilege escalation [18].

While we test our fusion approach against specific instance(s) of each type of exploit, we believe it to be generally applicable against other implementations of the exploits as well, since the approach targets the fundamental avenues of attack, rather than any instance-specific vulnerability.

Execution of Exploits. The last column of Table 1 presents examples and sources of the particular exploit instances we tested in our approach. In some cases, we ran the exploits using the Metasploit penetration testing framework [19] running inside the sidecar container, while the core-container runs a highly vulnerable Metasploitable image [20]. The corresponding metasploit module (containing the exploit code) then acts as an untrusted sidecar, trying to subvert the target core-container it runs against. While in other cases, we got the exploit code from github, or implemented our own.

In another experiment, we run this directly inside the core-container, as malicious actions of a hypothetical code.

Direct vs. Indirect Impact. Some of the exploits inherently cannot directly hurt the core-container, although they can cause an indirect impact. For example, a code injection attack has a direct process-corruption impact, but not an output falsification attack, where the sidecar falsely states that the core-container is out of memory, potentially triggering a policy-driven core-container re-instantiation. This is indicated as a 'Y' or an 'N' in columns 4 and 5, referring to the ability, or lack thereof, of an exploit to directly infect the core-container, when the malicious code runs inside the core-container or the host (column 4), versus running in a fused sidecar container (column 5).

Impact of Out-of-Scope Exploits. The first 7 attacks are out-of-scope of our threat model (Section III.A) and are highlighted as 'OOS' in Column 1 of Table 1. The first 4 of these are indirect-impact causing, like the aforementioned output falsification attack. Another indirect-impact case is that of a compromised monitoring backend, where, for example, the adversary gains access to core-container's credentials shipped by the sidecar, enabling remote access to the core-container. Similar is the case of a compromised host, where the sidecar is able to leak sensitive core-container information to a host-local accomplice via host-level side channels. A fourth indirect-impact attack vector is output format exploitation, where a malicious sidecar writes badly formatted data to its output file, in the hope of exploiting programming errors in the software code (e.g. buffer overflow). A successful subsequent subversion or privilege escalation can then potentially impact the core-container negatively. Output volume-based exploits, on the other hand, are in-scope and isolated using the blkio (block IO) cgroup controller.

The rest of the out-of-scope exploits are powerful enough to cause a direct core-container impact. This includes attacks against a weakly configured core-container, e.g. having insecure setuid binaries [21] lying around. The sidecar can execute such a binary to potentially escalate is privileges to that of the core-container user. A buggy kernel is also out-of-scope.

Impact of In-Scope Exploits. The different constraints added to the fused sidecar enable it to contain all of the in-scope exploits (row number 8 onwards), which would have otherwise directly impacted a core-container, assuming they were shipped inside third-party utility-microservice. This is indicated as an 'N' in Column 5, referring to the inability of an exploit to infect the core-container, when the former is run inside the fused sidecar. Compare this to the scenario when the exploit (masquerading as a legitimate code) is run inside the core-container or the host–a 'Y' in Column 4, indication successful core-container infection.

B. *Performance Analysis*

We evaluated the overhead of our solution on Ubuntu 16.04 KVM VM, with 4 vCPUs and 8G RAM, running Docker 1.12.1. The VM runs on quad-core / 16G RAM / Intel Core-i7 2.80GHz host machine, running Ubuntu 16.04 and QEMU 2.5.0. We observed an average overhead of 71ms reflected in the setup time of 344ms to run with fusion as opposed 273ms to running without fusion.

VI. Related Work

Privilege separation: Compartmentalization is one of the first building blocks towards application security, and can be done manually via application restructuring [24], but with significant programmer effort, or automatically via program analysis. We employ privilege separation in our approach, by running the utility-microservice code as unprivileged entities, with access to the privileged guest state being mediated by other kernel constructs.

OS isolation: The kernel constructs to manage access rights and restrictions, by themselves, are insufficient for comprehensive isolation. As pointed out in [14], DAC/MAC are inadequate for application privilege separation. Fine-grained Type Enforcement policies (as in SElinux) are inflexible, difficult to write and maintain, and thus, in practice, broad rights are conferred. *Chroot* limits only file system access, and switching credentials via setuid offers poor protection against weak DAC protections on namespaces. Namespaces- based view separation itself precludes cross-domain (e.g., a container) visibility. Linux capabilities, in their current form, still confer too much power than required for fine-grained access control. Capsicum [14] enables finer granularity capabilities via file-descriptor-level access control. However, since it combines security policy with code in the application, this makes it harder to cleanly specify and analyze a security policy. It also requires kernel modifications, and (minor) application modifications to make use of the proposed kernel construct.

TABLE I. EVALUATION TABLE

Exploits / attack vectors	Categories from		Direct guest infection ability when run inside:		Neutralized how?	Examples
	Hansman Hunt Taxonomy	Exploit DB	Host or core-container	fused-sidecar		
[OOS] Output falsification	Service Theft	-	N	N	Doesn't directly hurt the guest, indirect impact possible	Fake OOM signal → Policy alarm → guest re-instantiation or quarantine
[OOS] Output format exploitation	Buffer Ovrflow	-	N	N	Doesn't hurt the guest, indirect impact possible, preventable via interface audit	Buffer overflow → subvert software core → [privilege escalation] → destroy guest
[OOS] Compromised monitoring backend	Info Leak	-	N	N	Doesn't directly hurt the guest, indirect impact possible	Leaked ssh keys → remote shell
[OOS] Compromised host	Info Leak	-	N	N	--	Handover guest secrets to entity with external N/W privileges
[OOS] Kernel bugs	Subvert; DoS	PrivEsc, DoS	Y	Y	--	Dirty Cow CVE-2016-5195: gist.github.com/ rverton, Metasploit: BPF Priv Esc CVE-2016-4557
[OOS] Insecure setuid binaries in guest	Subvert	PrivEsc	Y	Y	--	Local privilege escalation → shell, Metasploit: HP smhstart OSVDB-91990
[OOS] Weakly configur--ed applications	Passwd; N/W	Web App	Y	Y	--	Local (unix-socket-based) MySQL access via weak password
Kernel-level rootkits	Trojan; Subvert	PrivEsc	Y (N in core-cnt'r)	N	Linux capability not given to sandbox or guest	Adore (syscall hooking), SucKIT (/dev/kmem)
Data corruption / tampering	DoS Virus Corrupt	DoS	Y	N	Guest rootfs mounted as read-only (RO rootfs)	Ransomware
Fork bomb, Memory hog, CPU hog	DoS	DoS	Y	N	Separate cgroups	[*shell*] `:(){ :|: & };:` [*shell*] `stress -c[-m] 1`
Port hijacking	DoS	DoS	Y	N	bind() limited via selinux / disabled via seccomp	Prevent webserver init by hoarding ports 80,8080
Zip bomb	DoS Virus Trojan	DoS	Y	N	RO rootfs, separate cgroups	Metasploit: gzip memory bomb DoS . Evilarc: github.com/ptoomey3/evilarc
Code injection, process corruption/termination	Subvert; Corrupt	PrivEsc, DoS	Y	N	Privilege separation, blocked ptrace(), RO /proc/<pid>/mem, no /dev/mem	sigsleeper: github.com/cys3c/PELT , http://pyrasite.com/
Library hooking (LD_PRELOAD)	Subvert	PrivEsc	Y	N	Can't change guest proc (no ptrace()), RO rootfs, preloading doesn't work with suid binaries	vlany: github.com/mempodippy/vlany
Covert execution	Service Theft	-	Y	N	Any procs (hidden or not) in sandbox harmless to guest, visible on host, separate cgroups	mimic: github.com/cys3c/PELT , processhider: github.com/gianlucaborello/libprocesshider
Backdoor, reverse shell , botnet	Worm; N/W	Remote	Y	N	No N/W communication with outside world	vlany: github.com/mempodippy/vlany
Privilege escalation (remote & local via lo)	Subvert; N/W	PrivEsc, Remote	Y	N	No N/W communication with outside world, `lo` interface communication also blocked in netfilter	Metasploit: Samba cmd exec CVE-2007-2447 , Metasploit: java deserialization CVE-2008-5353
Local privilege escalat--ion via filesystem	Subvert	PrivEsc	Y	N	RO rootfs, blocked `lo` interface communication	Metaploit: chkrootkit local privilege escalation CVE-2014-0476

VII. CONCLUSION

In this paper, we proposed an approach and implementation to diffuse boundaries between containers. We evaluated our approach by verifying the successful containment of several exploits in the sidecar across multiple dimensions. We analyzed different attack vectors and exploits and demonstrated that our approach achieves the fusion without compromising the security of the target container. As future work, we will examine automatically figuring out the needed capabilities/privileges needed by the sidecar by running it independently in a sandboxed environment, observing its behavior and requirements, and extract the desired configuration based on this analysis before fusing it with our core-container.

REFERENCES

[1] Sam Newman Building Microservices: Designing Fine-Grained Systems. O'reilly Media, Inc, USA – 2018

[2] Erik Wilde, Rest: from research to practice. Springer 2014

[3] Istio Service Mesh, available at: https://istio.io, accessed on September 2019.

[4] Shripad Nadgowda, Sahil Suneja and Canturk Isci, "RECap: Run-Escape Capsule for On-demand Managed Service Delivery in the Cloud", in the proceedings of the 10th Workshop on Hot Topics in Cloud Computing. Boston, USA. 2018

[5] CVE-2014-4701: nagios-plugins: check_dhcp Arbitrary Option File Read. https://access.redhat.com/security/cve/cve-2014-4701.

[6] CVE-2013-4215: nagios plugins: IPXPING_COMMAND uses fixed location in /tmp. https://bugzilla.redhat.com/show_bug.cgi?id=957482.

[7] CVE-2018-18245: Advisory: Nagios Core Stored XSS via Plugin Out- put. https://herolab.usd.de/wp-content/uploads/sites/4/2018/12/ usd20180026.txt.

[8] The New Stack. TOOLS AND PROCESSES FOR MON-ITORING CONTAINERS. https://thenewstack.io/ identifying-collecting-container-data/.

[9] seccomp - operate on Secure Computing state of the process. http://man7. org/linux/man-pages/man2/seccomp.2.html.

[10] netfilter / iptables. https://www.netfilter.org/.

[11] Overview of Linux capabilities. http://man7.org/linux/man-pages/ man7/capabilities.7.html.

[12] Overview of Linux namespaces. http://man7.org/linux/man-pages/ man7/namespaces.7.html.

[13] Linux Man Pages. collectd_selinux: Security Enhanced Linux Policy for the collectd processes. https://www.systutorials.com/docs/linux/man/ 8-collectd_selinux/.

[14] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, volume 46, page 2, 2010.

[15] Offensive Security. Exploit Database. https://www.exploit-db.com/.

[16] P. Hunt and S. Hansman. A taxonomy of network and computer attack method- ologies. *Computers and Security*, 24(1):31–43, 2003.

[17] CVE-2007-5623 nagios-plugins check_snmp possible buffer overflow. https: //bugzilla.redhat.com/show_bug.cgi?id=348731.

[18] Dawid Golunski. Nagios-Exploit-Root-PrivEsc-CVE- 2016-9566. https://legalhackers.com/advisories/ Nagios-Exploit-Root-PrivEsc-CVE-2016-9566.html.

[19] Metasploit | Penetration Testing Software. https://www.metasploit.com/.

[20] Metasploitable 2 Exploitability Guide. https://metasploit.help.rapid7.com/docs/metasploitable-2-exploitability-guide.

[21] Michael C. Long. Attack and Defend: Linux Privilege Escalation Techniques of 2016. https://www.sans.org/reading-room/whitepapers/linux/ attack-defend-linux-privilege-escalation-techniques-2016-37562.

[22] Kubernetes: Production-Grade Container Orchestration, http://kubernetes.io

[23] Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker. com/.

[24] D. G. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *Proceedings of the 1st European Workshop on System Security*, pages 40–46. ACM, 2008.