A Measurement Study on Linux Container Security: Attacks and Countermeasures

Xin Lin*†
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
linxin@iie.ac.cn

Lingguang Lei†‡
Institute of Information Engineering, CAS
Beijing, China
leilingguang@iie.ac.cn

Yuewu Wang†
Institute of Information Engineering, CAS
Beijing, China
wangyuewu@iie.ac.cn

Jiwu Jing†
Institute of Information Engineering, CAS
Beijing, China
jingjiwu@iie.ac.cn

Kun Sun
George Mason University
Fairfax, USA
ksun3@gmu.edu

Quan Zhou†
Institute of Information Engineering, CAS
Beijing, China
zhouquan@iie.ac.cn

ABSTRACT

Linux container mechanism has attracted a lot of attention and is increasingly utilized to deploy industry applications. Though it is a consensus that the container mechanism is not secure due to the kernel-sharing property, it lacks a concrete and systematical evaluation on its security using real world exploits. In this paper, we collect an attack dataset including 223 exploits that are effective on the container platform, and classify them into different categories using a two-dimensional attack taxonomy. Then we evaluate the security of existing Linux container mechanism using 88 typical exploits filtered out from the dataset. We find 50 (56.82%) exploits can successfully launch attacks from inside the container with the default configuration. Since the privilege escalation exploits can completely disable the container protection mechanism, we conduct an in-depth analysis on these exploits. We find the kernel security mechanisms such as Capability, Seccomp, and MAC play a more important role in preventing privilege escalation than the container isolation mechanisms (i.e., Namespace and Cgroup). However, the interdependence and mutual-influence relationship among these kernel security mechanisms may make them fall into the "short board effect" and impair their protection capability. By studying the 11 exploits that still can successfully break the isolation provided by container and achieve privilege escalation, we identify a common 4-step attack model followed by all 11 exploits. Finally, we propose a defense mechanism to effectively defeat those identified privilege escalation attacks.

*Also with Institute of Information Engineering, CAS, Beijing, China.
†Also with Data Assurance and Communication Security Research Center, CAS, Beijing, China.
‡Corresponding author.

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274720

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Virtualization and security**;

KEYWORDS

Container, Privilege Escalation, Kernel Security Mechanisms

ACM Reference Format:
Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3274694.3274720

1 INTRODUCTION

Container technology is increasingly adopted by the industrial community due to two major advantages. First, the container orchestration tools such as Docker [31] and Kubernetes [2] facilitate the deployment, scaling, and management of the containerized applications. Therefore, containers are increasingly used in the production environment [64]. Also, the cloud vendors begin to provide container services, e.g., Amazon Fargate [11], Microsoft Azure Kubernetes Service [54], etc. Second, as an OS-level virtualization technology, container mechanism is lightweight thus more attractive to the resource-constraint mobile platform. Several container-based Bring Your Own Device (BYOD) solutions [33, 44, 63] have been proposed and deployed.

Meanwhile, security concerns have become the major barrier for further adoption of container mechanism. Particularly, since all containers running on one host share the same Linux kernel, the isolation provided by the container mechanism will become completely invalidated once the kernel is compromised. Therefore, it is necessary to give a systematic evaluation and analysis on the security of the container mechanism. Most existing studies evaluate container security from the system architecture or design principle level. For example, M. Ali Babar et al. [3] give a comparative analysis on the isolation mechanisms provided by three container engines, i.e., Docker, LXD, and Rkt. Thanh Bui et al. [6] briefly compare the security of hardware-based virtualization technology

(e.g., XEN) and OS-level virtualization technology (i.e., container mechanism) from system architecture level. Reshetova et al. [62] theoretically analyze the security of several OS-level virtualization solutions including FreeBSD Jails, Linux-VServer, Solaris Zones, OpenVZ, LxC and Cells etc. Some researchers [46, 56] also evaluate the container security using potential vulnerabilities against specific container mechanisms such as Docker. However, due to the lack of a measurement study of container security using real exploits, it is hard to know the security of containers in the real world. Particularly, due to the small number of exploits reported in the vulnerability databases such as CVE [12] and NVD [58], it is challenging to provide a persuasive security evaluation on container mechanism.

In this paper, we perform a measurement study on Linux container security using real exploits that may be launched to attack containers. First, we manually collect an attack dataset including 223 exploits that may be misused to attack the container platform. The exploit collection is based on one key observation that since container is mainly protected by the security mechanisms that have been integrated in the Linux kernel, many exploits that work on the Linux platform may be theoretically effective on attacking the container platform too. We select the exploits from the Exploit-db [15]. To facilitate the security analysis, we propose a two-dimensional attack taxonomy to classify these potential exploits. On the first dimension, the exploits are classified into four types according to the consequences of the attacks, i.e., sensitive information leakage, remote control, denial of service and privilege escalation. On the second dimension, the exploits are classified into four categories based on the layers the vulnerable programs reside at (or the influence range of the attacks), which include web app layer, server layer, lib layer and kernel layer. After classification, we find the "web app layer" exploits make up the majority of exploits (59.19%), though they introduce minimum influence. Almost all "lib layer" exploits aim at conducting denial of service attacks. Moreover, 76.09% "kernel layer" exploits can cause privilege escalation, and most (68.63%) of privilege escalation attacks are caused by "kernel layer" exploits.

Then we evaluate the security of container mechanism by manually executing the exploits on both the container platform and the original Linux platform. Specifically, we select 88 typical exploits (covering all types mentioned above) from the 223 exploits after removing the exploits that fail on the original Linux platform and redundant exploits in each category of the two-dimensional taxonomy. In total, we find 50 (56.82%) exploits are still effective on the container platform with the default configuration set by Docker. Moreover, we perform some further analysis on the privilege escalation attacks (the total number is 37), which can completely invalidate the security protection provided by container mechanisms after obtaining the root privilege. Particularly, we conduct a more detailed analysis on the effectiveness of the kernel security mechanisms (i.e., Capability [48], Seccomp [29] and MAC [24, 52]), container mechanisms (i.e., Namespace [49] and Cgroup [47]) and CPU protection mechanisms (i.e., KASLR [16], SMAP&SMEP [36]) on preventing privilege escalation on the container platform. The results show that kernel security mechanisms play a more important role on blocking the privilege escalation attacks than the container mechanisms, and the CPU protection mechanisms usually could be bypassed from inside the container. We observe that the kernel security mechanisms do not always work independently, and they are sometimes interdependent and may mutually affect one another. Improper configuration of these kernel security mechanisms may impair their protective capability. For example, a loose Capability (i.e., assigning many capabilities) policy might disable a strict Seccomp configuration (i.e., allowing only a few system calls).

Finally, we develop a new defense mechanism based on enforcing a fine-grained control of the Linux kernel Credential and Namespace mechanisms to defeat the 11 exploits that can break the existing isolation provided by container and thus obtain the root privilege. After a close analysis of the 11 privilege escalation exploits, we discover that they all follow a 4-step attack model consisting of (1) bypassing (or manually disabling) the KASLR mechanism, (2) overwriting certain kernel function pointers via kernel vulnerabilities such as UAF, race condition, improper verification, buffer overflow etc., (3) disabling SMEP&SMAP mechanisms, and (4) invoking the kernel function *commit_creds()* to acquire the ROOT credential with all capabilities. We propose a simple but effective defense mechanism to defeat all these root privilege attacks by constraining the calling of kernel function *commit_creds()*. Our experimental results show that it can effectively defeat all 11 exploits with negligible overhead.

In summary, we make the following contributions.

- We manually create an exploit dataset containing 223 exploits against container solutions. We develop a two-dimensional attack taxonomy to classify the collected exploits.
- We study the effectiveness of existing container security by manually executing 88 typical exploits on original Linux platform and on the container platform. We find more than half (56.82%) of the exploits are still effective on the container platform.
- We provide an in-depth analysis on the privilege escalation attacks, especially the effectiveness of kernel security mechanisms (i.e., Capability, Seccomp, and MAC), container mechanisms (i.e., Namespace and Cgroup) and CPU protection mechanisms (i.e., KASLR, SMAP and SMEP) on preventing privilege escalation against the container platform.
- We extract a kernel privilege escalation attack model by analyzing the 11 exploits that could escape container boundary and gain ROOT privilege, and then propose a defense solution to defeat all these expoits. Experiment results show that the defense solution can block the exploits effectively with negligible overhead.

The remaining of the paper is organized as follows. Section 2 introduces necessary background knowledge. Section 3 describes exploit dataset and the double-dimensional taxonomy of these exploits. Section 4 details the quantitative evaluation and analysis of container security. Section 5 describes the privilege escalation attack model and the defense system. Section 6 discusses the limitations on the defense system. We describe related works in Section 7. Finally, we conclude the paper in Section 8.

2 BACKGROUND

2.1 Linux Container

Linux container [30, 35, 45] is a lightweight OS level virtualization technology that provides isolation and containment for one or more processes. The processes inside a container feel like they own the entire system, though several containers share the same Linux kernel. The isolation is majorly achieved by two kernel mechanisms, i.e., Namespace [49] and Cgroup [47]. There are seven types of namespaces, i.e., user, uts, net, pid, mnt, ipc and cgroup, and each namespace constructs a specific isolated kernel resources for the containers. For example, mnt namespace provides an isolated file system for a container by isolating the file system mount points. After isolation, the files in different mnt namespaces are not visible to each other and cannot affect each other. In another example, net namespace ensures the isolation of network resources, such as IPv4 and IPv6 protocol stacks, socket ports, etc. Compared to the Namespace mechanism that concerns kernel data isolation, the Cgroup mechanism focuses more on performance isolation by limiting the amount of resources (e.g., CPU, memory, devices, etc.) that a container can use. Docker [25, 31, 34, 53] is a pervasively used container engine that facilitates the management of the containers, such as container creating, deleting, starting and stopping, etc. All namespaces except "cgroup" and "user" are supported by Docker, and the Cgroup mechanism is also well supported by Docker. For example, users can limit the CPU, memory and devices resources a container can use by setting the *–cpu*, *–memory* and *–device* options, respectively.

2.2 Linux Kernel Security Mechanisms

One critical security risk of the container mechanism is that all containers running on the same host share the same Linux kernel. If a process inside the container compromises the Linux kernel, the isolation provided by the container mechanism becomes invalid. Therefore, several Linux kernel security mechanisms are adopted to constrain the capability of the processes inside the containers, such as Capability [48], Seccomp [29] and Mandatory Access Control (MAC) mechanisms. Through Capability mechanism, the superuser privilege (i.e., ROOT privilege) is divided into 38 distinct units, known as capabilities. Each capability represents a permission to process some specific kernel resources. For example, the *CAP_NET_ADMIN* capability denotes the permissions to perform network-related operations. By default, the containers created by Docker own 14 capabilities [21]. The Seccomp mechanism constrains the system calls a process can invoke. Docker defines the available system calls for a container through a Seccomp profile file, which by default includes more than 300 system calls [26]. Both Capability and Seccomp are Discretionary Access Control (DAC) mechanisms, and SELinux [52] and AppArmor [24] are two MAC mechanisms adopted by containers. SELinux has been integrated in CentOS / RHEL / Fedora distros, and AppArmor has been integrated in Debian/Ubuntu distros. AppArmor utilizes a path-based enforcement model [5], while SELinux adopts a label-based enforcement model [52].

2.3 CPU Protection Mechanisms

Three CPU protection mechanisms are also commonly used to prevent the attacks to the Linux kernel, i.e., Kernel Address Space Layout Randomization (KASLR) [16], Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP) [36]. KASLR is a well-known technique to make exploit harder by adding a random slide to the kernel base address at boot time, rather than using a fixed base address all the time. SMAP prevents supervisor-mode programs from reading and writing user-space memory, while SMEP prevents supervisor-mode code from unintentionally executing user-space code. SMAP and SMEP could be enabled by setting the 21th and 20th bits of the CR4 register, respectively.

3 ATTACK DATASET DESCRIPTION

We first describe how we collect the exploits that might be effective on the container platform. Next, we adopt a two-dimensional method to classify those potential container attacks. First, we classify the attacks into four categories (i.e., *sensitive information leakage*, *remote control*, *denial of service*, and *privilege escalation*) according to their consequences. Second, we classify them into four layers (i.e., web app, server, lib, and kernel) based on the attacks' influence ranges. We then present the construction of the final exploit dataset using the two-dimensional method.

3.1 Exploit Collection

Currently, there are no existing exploit datasets specifically targeting at the container platform. It is a challenge to create an attack dataset that consists of real exploits that may successfully attack the containers. One key observation is that since container is mainly protected by the security mechanisms that have been integrated in the Linux kernel, many exploits that work on the Linux platform may be theoretically effective on attacking the container platform too. As such, we collect the exploits from the Exploit-db [15], which is a public exploit database containing comprehensive exploits of various platforms such as Windows, Linux, iOS, Android etc. Specifically, Exploit-db divides the exploits into four categories, namely, web application, remote, local & privilege escalation, and denial of service & poc (Proof of Concept). On October 20, 2017, we collect the latest 100 exploits of each category whose targeting platform is Linux. All 400 exploits are published in 2016 and 2017, where 274 CVEs used by the exploits are published in 2016-2017 and only 24 CVEs are published between 2013 and 2015.

Since container is generally used to deploy the background services such as web server, database server etc. [68], we then filter out the exploits which will probably fail on the container platform, such as the ones attacking the Graphical User Interface (GUI) related programs (e.g., browsers, adobe-flash plugins). Finally, we obtain 223 exploits out of 400 which might be effective on the container platforms. For each exploit, we collect its unique EDB-ID (Exploit Database IDentifier), exploit codes to carry out the attack, publishing date, type information on the Exploit-db, and CVE-ID (Common Vulnerabilities and Exposures IDentifier). For some exploits, we also collect the vulnerable programs, for example, the vulnerable web applications.

3.2 Attack Taxonomy

The Exploit-db classifies the exploits into four categories, i.e., web application, remote, local & privilege escalation, and denial of service & poc. However, it uses a mixed classification method. For example, "web application" denotes the attacking object (or the vulnerable programs), while "denial of service" represents the attacking consequence. To give a more intuitive illustration, we choose to classify the exploits from two aspects.

First, we classify the exploits into four layers based on the influence ranges of the attacks, i.e., web application layer, server layer, library layer, and kernel layer. The container is generally used for the development and deployment of underlying background services (e.g., web service, database service) that provide support for multiple upper applications (e.g., web applications) [68]. Therefore, the programs in the container platform have a similar hierarchy structure like in the general operating systems, and compromising the programs in different layers will cause different influence ranges. For example, attacking of the web applications usually causes self-damage with only minimum influence. In contrast, compromising of the Linux kernel has maximum influence, since it will lead to damages to all containers running on the same kernel. The attacks on the server programs (e.g., apache, nginx) and libs (e.g., glibc, libgig) have medium influence, which could impact all applications depending on them.

Second, we divide the exploits into four categories based on the consequences of the attacks, i.e., sensitive information leakage, remote control, denial of service, and kernel privilege escalation. In the following, we summarize the primary attacking methods in each category by manually analyzing the attack principle, text description of CVE [12], and the type of CWE (Common Weakness Enumeration) [13] obtained from NVD [58].

Sensitive Information Leakage. Three attack methods could be adopted in the exploits to steal the sensitive information, i.e., crafting malicious request/response, triggering race condition, and bypassing access control mechanisms. The first method is primarily used to attack the vulnerable programs in the "web application layer" and "server layer". Specifically, the attackers can send a malicious http request/response with crafted parameters to the web app or web server to disclose sensitive files, directories, and information. Second, the attackers can stealthily access sensitive directories and data by triggering race condition (e.g., the exploit with EDB-ID 39771 utilizes race condition in the *perf_event_open()* kernel function to gain the sensitive data in the "/etc/shadow" file). Third, the attackers can utilize the design flaw in the functions or access control mechanisms for illegal access (e.g., CVE-2017-15014, CVE-2017-9150).

Remote Control. Most "remote control" exploits target at vulnerable web applications or servers. Injection is the most common way to control the victims remotely. There are totally three types of injections, which are object injection (e.g., XML External Entity (XEE) injection [37], Hypertext Preprocessor (PHP) files injection), code injection (e.g., Cross Site Script (XSS) injection, Structured Query Language (SQL) injection, Object-Graph Navigation Language (OGNL) injection) and command injection (e.g., OS command injection [14]). In the "injection" method, the attacker usually utilizes improper input validation of the victims to execute arbitrary injected object such as SQL, HTML (Hypertext Markup Language) page, JS (Java Script), PHP file, Python code, OS command etc. "Crafting request/response" is another important method. Specifically, the attackers send malicious http request (e.g., CSRF (Cross-Site Request Forgery) [43] and SSRF [57] (Server-Side Request Forgery)) with crafted fields (e.g., evil cookie) to cheat the victim into bypassing the authentication, opening redirect url, uploading files, or executing commands remotely. Crafting response is rarely used, where the attackers impersonate the server to send crafted response with malicious fields to cheat the requester into executing arbitrary codes, opening redirect url, or uploading files.

Denial of Service. There are two types of denial of service, i.e., resource exhaustion and disability. The attackers either overly consume resources like CPU and memory by infinitely looping the calculation or constantly allocating the memory, or cause kernel panic or application crashing through triggering buffer/heap overflow or crafting illegal input data.

Privilege Escalation. The most severe consequence of the attacks is privilege escalation, through which the attackers obtain the kernel root privilege. Two methods are utilized in the exploits to achieve privilege escalation, i.e., memory modification and file modification. "Memory modification" means that the attackers change the control flow by overwriting certain data structure in the memory, or execute malicious privileged-binaries by modifying the stack memory. "File modification" means the adversaries modify privileged files (e.g., "/etc/passwd", "/etc/crontab") to change the superuser's password or modify the file attributes controlling privileges (e.g., ACL xattr data) to execute malicious programs with root privilege. For example, the attackers first symlink a malicious object to a privileged directory, or inject a malicious command to the component which is in the privileged context. Then, they can achieve privilege escalation by executing *setuid(0)* in the privileged context or directory.

3.3 Exploit Dataset

Table 1 shows the exploit dataset, containing the exploits that might be effective in the container platform. We depict the number of the exploits using the two-dimensional taxonomy described in Section 3.2. In total, there are 223 exploits and 148 vulnerabilities (or CVEs) involved. One exploit might involve several vulnerabilities, and one vulnerability might be utilized in many exploits (e.g., CVE-2017-1000366 is utilized in exploits with EDB-ID 42276, 42275 and 42274). Therefore, the exploit number is not equal to the vulnerability number. In some columns, the total number is less than the sum of all rows in the column. This is because some exploits cause two or more different consequences (e.g., the exploit with EDB-ID 43015 causes "sensitive information leakage" and "remote control"). Although compromising web applications introduces minimum influence, the "web app layer" exploits make up the majority of exploits (59.19%). Almost all "lib layer" exploits are aiming at denial of service attacks. Moreover, 76.09% "kernel layer" exploits can cause privilege escalation, and most (68.63%) of privilege escalation attacks are caused by "kernel layer" exploits.

Table 1: Exploit Dataset

Categories \ Layers	Web App	Server	Lib	Kernel	Total
Sensitive Information Leakage	10	5	\	2	17
Remote Control	115	16	1	\	132
Denial of Service	\	3	15	9	27
Privilege Escalation	10	6	\	35	51
Total	132*	29*	16	46	223*

* Total number is less than the sum of all rows in the column when some exploits cause two or more different consequences.

4 SECURITY EVALUATION OF CONTAINER

We utilize the exploit dataset collected in Section 3 (Table 1) to evaluate the security of container mechanism, including the security enhancement provided by container to the programs inside it and the isolation strength provided by the container on protecting the Linux kernel.

4.1 Experiment Setup

We evaluate the security of container mechanism by comparing the execution results of exploits on original Linux platform and on the container platform. As shown in Figure 1 a), for the exploits compromising user space programs, including "web app layer", "server layer" and "lib layer" exploits, we deploy vulnerable programs inside the container and run the exploits from outside the container (i.e., control host), to evaluate whether container provides extra security enhancement for the programs inside it. As shown in Figure 1 b), for the exploits attacking Linux kernel, we deploy the vulnerable kernel and execute the exploits from inside the container, to evaluate whether container provides enough isolation. Specifically, Docker 17.09.1-ce which supports Linux kernel security mechanisms including Seccomp, Capability and MAC, is used to construct the containers. Some exploits require lower version Linux distros which are not by default shipped with Docker 17.09.1-ce. On these distros we install the Docker 17.09.1-ce via static Docker Linux binary [27] manually. When one exploit works well on several Linux distros, we choose to utilize the Linux distro that supports more Linux security mechanisms (e.g., the exploit with EDB-ID 41762 is effective on both Ubuntu 14.04 and 15.10, while only Docker installed on Ubuntu 14.04 supports the Seccomp mechanism [29]. Thus we choose to use Ubuntu 14.04.).

In total, 88, instead of 223, exploits are utilized in the evaluation. First, we remove 41 exploits that fail on the original Linux platform. The primary reasons for those failures are two-fold. One is because the vulnerable old version programs are no longer available (e.g., the exploit with EDB-ID 43015 aims at the vulnerable FileRun program of version 2017.03.18, however, only the latest version could be downloaded from the vendor's webpage). The other reason is that the exploit codes could not execute successfully (e.g., some exploit codes are inconsistent with their descriptions on the Exploit-db). Second, for exploits in the same layer (except the privilege escalation exploits) with similar attack principles, we choose only one exploit as the representative in order to avoid redundant exploits (e.g., all exploits with EDB-ID from 42667 to 42689 achieve remote control on the web apps through SQL injection by using different URLs).

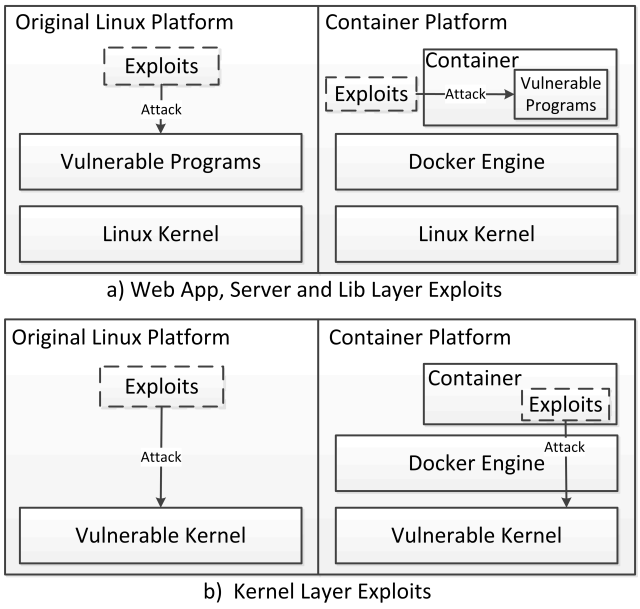

Figure 1: Experiment Setup

4.2 Result Overview

Table 2 shows the preliminary results of our experiment. Results on the container platform are obtained with the default security configuration set by Docker. "Lin" means the number of exploits execute successfully on the original Linux platform, and "Con" denotes the number on the container platform. In total, 56.82% of the 88 exploits are still effective on the container platform. Most exploits (except the ones causing privilege escalation) in the "web app layer", "server layer" and "lib layer" remain effective, which means container does not provide much security enhancement for the programs inside it.

Sensitive Information Leakage. The exploits in the "web app layer" and "server layer" obtain the sensitive information through crafting HTTP request (e.g., CSRF) or response (e.g., the exploit with EDB-ID 41783 which utilizes CVE-2016-6816), and are not blocked by the container mechanisms. Reason for the failure of the exploit in the "kernel layer" is that it needs to utilize the vulnerable eBPF API, invocation of which requires the CAP_SYS_ADMIN capability. And this capability is by default removed for the processes inside container.

Remote Control. Container mechanism fails in blocking the "remote control" attacks via both injection (e.g., XSS, XEE, SQL, OGNL, and command injection etc.) and crafting request (e.g., CSRF) or response (e.g., the exploit with EDB-ID 40920 which utilizes CVE-2016-9565).

Table 2: Security Evaluation Result Overview

Categories \ Layers	Web App	Server	Lib	Kernel	Total
	Con/Lin[1]	Con/Lin[1]	Con/Lin[1]	Con/Lin[1]	Con/Lin[1]
Sensitive Information Leakage	3 / 3	2 / 2	\	0 / 1	5 / 6
Remote Control	23 / 23	7 / 7	1 / 1	\	31/ 31
Denial of Service	\	2 / 2	7 / 7	2 / 6	11 / 15
Privilege Escalation	0 / 4	0 / 3	\	4 / 30	4 / 37
Total	25*/ 29*	11 / 14	8 / 8	6 / 37	50*/ 88*

[1] "Lin" means the number of exploits execute successfully on the original Linux platform, and "Con"denotes the number on the container platform.
* Total number is less than the sum of all rows in the column. This is because some exploits cause two or more different consequences.

Table 3: Function of Security Mechanisms in Preventing Privilege Escalation Attacks

EDB-ID	CVE-ID	Security Mechanisms				
		Namespace	Cgroup	Capability	Seccomp	MAC
Web App Layer						
43002	CVE-2017-15276			●		
40921	CVE-2016-9566			●		
42305	CVE-2017-6970			●		
40938	CVE-2014-6271			●		
Server Layer						
40768	CVE-2016-1247			●		
40678	CVE-2016-6663			●		
40450	CVE-2016-1240			●		
Kernel Layer						
41994*	CVE-2017-7308					
43127* 43029*	CVE-2017-5123					
40871*	CVE-2016-8655					
40489 40435 44300 40049	CVE-2016-4997			● NET_ADMIN[1]		
41458	CVE-2017-6074			● NET_ADMIN[1]		
43418	CVE-2017-1000112			● NET_ADMIN[1]		
41995	CVE-2016-9793			● NET_ADMIN[1]		
42887	CVE-2017-1000253			●		
42274 42275 42276	CVE-2017-1000366 CVE-2017-1000371 CVE-2017-1000379 CVE-2017-1000370			●		
40003 39277	CVE-2016-0728				●	
39992	CVE-2016-1583			●	●	●
41762	CVE-2017-1575			●	●	●
41763	CVE-2017-1576			●	●	●
39166 39230	CVE-2015-8660			●	●	●
40847 40616 40611	CVE-2016-5195	●		●		
40839 40838		●		●	●	
40759 39772	CVE-2016-4557	●		●	●	
41999	CVE-2016-2384	●	●			

• Security mechanism blocks the exploit.
* Exploit bypasses all 5 security mechanisms.
[1] Exploit can achieve privilege escalation when the "NET_ADMIN" capability is included in the *cap_bset* of the caller process. Other exploits marked "•" in "Capability" column can only be successful when all 38 capabilities are included in the *cap_bset*. The "*cap_bset*" defines the highest privilege a process could reach.

Denial of Service. Most "denial of service" attacks on the servers and libs are not blocked by the container mechanism. However, we find this is because Docker does not constrain the amount of resource a container could use through Cgroup mechanism by default. After limiting the CPU and memory resource through *–cpu* and *–memory* options, 5 extra attacks are blocked, where 1 in "server layer" and 4 in "lib layer". The rest "denial of service" attacks in the "server layer" and "lib layer" that are not prevented by Cgroup can cause two main damages. One is kernel panic, and the other is app crashing. We see 4 "denial of service" exploits in the kernel layer are blocked because of the Namespace isolation, unsufficient capabilities, and vulnerable APIs prohibited by Seccomp. For example, net namespace can successfully block CVE-2017-14489 and CVE-2017-16939, which require NETLINK to communicate with Linux kernel.

Privilege Escalation. The main reason for the failure of the "privilege escalation" attacks is the capability constraints enforced by Docker. As described in Section 2.2, Docker gives only 14 capabilities to the processes inside the containers by default. Therefore, most "privilege escalation" attacks inside containers can only obtain 14 capabilities, rather than all 38 capabilities (i.e., ROOT privilege). We will give a more detail analysis on the "privilege escalation" exploits in the next section.

4.3 Analysis of Privilege Escalation Attacks

The biggest security weakness of the container mechanism is sharing Linux kernel. And the "privilege escalation" attacks could completely disable the isolation provided by container. In this section, we give a more detail analysis on the security mechanisms (i.e., Namespace, Cgroup, Capability, Seccomp, MAC, KASLR, SMAP, and SMEP) associated with the "privilege escalation" exploits, to illustrate the reasons for the success and failure of the exploits on the container platform. Specifically, we first analyze the attack principles of each exploit by studying the exploit description on Exploit-db, CVE, CWE, blogs written by the authors and so on. Then, we perform static analysis and dynamic debugging on the source codes of the vulnerable apps and exploits. Finally, we figure out the attack preconditions (e.g., required capabilities, associated APIs/commands/files, CPU protection mechanisms that need to be bypassed etc.) and the methods to bypass the security mechanisms for each exploit.

CPU Protection Mechanisms. CPU protection mechanisms are the first line of defense to protect the Linux kernel, however they usually could be bypassed from inside the container. First, most exploits require to bypass the KASLR mechanism. For example, *commit_creds()*, *prepare_kernel_cred()* are two important methods leveraged by the exploits to elevate the privilege, and

native_write_cr4() is the method commonly used to disable the SMAP&SMEP protection. In order to obtain the addresses of these static functions whose offset to the kernel base address is constant, exploits need to bypass the KASLR mechanism and obtain the kernel base address. A feasible method is as follows. Generally, the *dmesg* command used to print kernel syslog is by default available in the container. When the system boots up, the kernel text address will be recorded in the kernel syslog, and could be obtained by searching the key words such as "Freeing SMP" or "Freeing unused". However, this way of figuring out the kernel text address works only for some time after booting, as syslog only stores a fixed number of lines and starts dropping them at some point.

Second, in order to access user space data and execute user space code from kernel, exploits usually need to bypass the SMAP&SMEP [1] mechanisms. Generally, there are two methods. The first one is to use the vulnerabilities such as buffer overflow, race condition, UAF (Use After Free) etc., to generate the condition where certain kernel data structure can be covered. By overwriting a pointer and its parameter in this structure, making it point to the address of *native_write_cr4()* and assigning zero to the parameter, the CR4 register will be set to 0 and SMAP&SMEP will be disabled. The second method is finding the exploitable gadgets which could be used to set the CR4 register, and concatenating them into a malicious invocation chain via ROP (Return-oriented Programming) [71] attack.

Container Mechanisms. The container specific security mechanisms, i.e., Namespace and Cgroup, block about 21.62% "privilege escalation" attacks. For example, mnt namespace blocks exploits utilizing CVE-2016-5195 [69] and CVE-2016-4557. These exploits escalate privilege by modifying the read-only files associated with the superuser account (e.g., "/etc/passwd", "/etc/crontab"). But processes isolated by the mnt namespace can only modify the files inside the namespace, thus are not able to modify the real target files on the host or change the password of the host's ROOT. It also blocks exploits utilizing CVE-2016-2384, which leverages the vulnerability on the USB midi device and requires the device to be visiable inside the container. Exploits utilizing CVE-2016-2384 can also be blocked by the Cgroup mechanism, as the USB midi device is usually not mounted in the container by default.

Kernel Security Mechanisms. Other Linux kernel security mechanisms including Capability, Seccomp and MAC, block about 86.49% attacks with the default configuration enforced by Docker. However, if we loose the constrain by giving the *CAP_NET_ADMIN* capability to the processes inside container, only 67.57% attacks could be blocked. Since "*CAP_NET_ADMIN*" controls the operations such as configuring the IP address, routing table, firewall etc., it is sometimes necessary for the processes running inside the container.

- **Capability** The exploits are blocked by the Capability mechanism in two cases. First, the vulnerable API invoked in the exploit can execute successfully only when the attacking process owns specific capability, while the capability is by default not given to the processes inside the container. For example, the exploits utilizing CVE-2017-6074 need to invoke the vulnerable API *setsockopt()* with the parameter set as "*IPV6_RECVPKTINFO*", while execution of it requires the "*CAP_NET_ADMIN*" capability.

  Second, each process is associated with a "*cap_bset*" structure which defines the highest privileges the process could reach. Normally, the process on the control host is assigned with a "*cap_bset*" containing full 38 capabilities. Therefore, the exploits could obtain ROOT privilege on the original Linux platform. However, the process inside the container is only assigned with a "*cap_bset*" including only 14 capabilities by default. Therefore, the privilege escalation attacks inside the container can only obtain 14 capabilities rather than ROOT privilege (i.e., 38 capabilities).

  For example, exploits associated with CVE-2016-5195, CVE-2016-4557 achieve privilege escalation by first modifying the files related to superuser account and then executing the privileged programs such as SU, SUDO etc. Since the privileged programs in the container have only 14 capabilities by default, the exploits can not gain real ROOT privilege. Exploits attacking through "stack clash" [23] are similar (e.g., CVE-2017-1000366, CVE-2017-1000371, CVE-2017-1000379, CVE-2017-1000370). They leverage the feature that shared libraries of the privileged binaries will be also executed with privileges. As privileged binaries inside container own only 14 capabilities, the crafted ".so" can neither be executed under real ROOT. Exploits utilizing design flaw of overlayfs (e.g., CVE-2017-1576, CVE-2017-1575) to enable the SUID attribute of unprivileged program and change the program's owner to "ROOT" can also be blocked, as the "ROOT" user inside container owns only 14 capabilities. Exploits attacking via modifying the file with a symlink to the privileged file (e.g., CVE-2016-1247), or injecting files to a privileged directories (e.g., CVE-2014-6271) are the same. As the privileged files and directories in the container do not have full capabilities, malicious command injected or linked can not be executed under ROOT privileged context.
- **Seccomp** Exploits are blocked by the Seccomp mechanism, as some system calls required for the attacks are prohibited by the Seccomp policy, such as the *mount()* system call in CVE-2016-8660, the *keyctl()* system call in CVE-2016-0728, and the *ptrace()* system call in EDB-ID 40839 and 40838 associated with CVE-2016-5195.
- **MAC** The Selinux and AppArmor mechanisms mainly prevent the attacks (e.g., CVE-2016-1583, CVE-2017-1575, CVE-2017-1576 and CVE-2015-8660) that require to mount certain file systems.

Relationships Among the Security Mechanisms. The security mechanisms (i.e., Namespace, Cgroup, Seccomp, Capabilities and MAC) do not always work independently, they may mutually affect one another at sometime. First, some exploits could only be blocked when both the Namespace and Capability policies are configured properly. A loose Namespace policy might disable a strict Capability policy, and vice versa. For example, the exploits with EDB-ID 40847, 40616 and 40611 in Table 3 can be used to obtain ROOT privilege by either loosing the Namespace policy (e.g., allowing the control host's "/etc/passwd" file to be visible inside the container) and meanwhile modifying the files associated with the superuser's privileges (e.g. "/etc/passwd"), or loosing the Capability policy (i.e., assigning the processes' cap_bset with the full 38

capabilities) and executing the privileged programs (e.g., SU) inside the container. Functions of the Namespace or Capability mechanisms on the exploits with EDB-ID 40839, 40838, 40759 and 39772 are similar. One difference is these four exploits require also some specific system calls to be allowed through Seccomp mechanism. For example, the *ptrace()* system call is necessary for the exploits with EDB-ID 40839 and 40838, and the *bpf()* system call is required by exploits with EDB-ID 40759 and 39772.

Second, a kernel function sometime could be enabled by setting either Capability or Seccomp mechanism. For example, the exploits with EDB-ID 40839, 40838 need to invoke the *ptrace()* system call, and *ptrace()* can execute successfully by either removing the *CAP_SYS_PTRACE* requirement in the Seccomp policy or adding *CAP_SYS_PTRACE* capability through Capability mechanism. Similarly, the *bpf()* system call invoked by the exploits with EDB-ID 40759, 39772 can execute normally by either removing the *CAP_SYS_ADMIN* requirement in the Seccomp policy or adding *CAP_SYS_ADMIN* capability through Capability mechanism.

Third, some exploits can be blocked when either Seccomp mechanism or MAC mechanism are set properly. For example, one primary reason for the failure of CVE-2017-1575 inside container is that the exploit can not mount overlayfs in the container. By default, the mount operations are forbidden by both AppArmor policy and Seccomp policy. However, we find the exploit could only work well when both policies allow the mount operations.

Fourth, some exploits can only be blocked when both Namespace and Cgroup mechanisms are configured correctly. For example, the CVE-2016-2384 requires to access malicious USB midi device, which could be allowed through mnt namespace (e.g., mount the device in the container) or Cgroup mechanism (e.g., use *–device* option when starts the container). Two mechanisms should work together to protect the container, if Cgroup mechanism adds the device in container, the Namespace blocking is invalid, and vice versa.

4.4 A Brief Summary

We can draw following conclusions based on the above analysis. First, container does not provide much security enhancement for the programs inside it, except the exploits aiming to achieve privilege escalation. Second, Cgroup mechanism is not effectively utilized by default in defending against the DoS attacks. Third, the kernel security mechanisms such as Capability, Seccomp and MAC play a more important role in preventing privilege escalation attacks than the container isolating mechanisms (i.e., Namespace and Cgroup). The former blocks 67.57% privilege escalation attacks while the later blocks only 21.62%. Fourth, each kernel security mechanism (including Namespace, Cgroup, Seccomp, Capabilities and MAC) restricts kernel permissions from different angles in a fragmented way, while the the relationships among them are intricate and complicate. Improper configuration of these security mechanisms might lower their protective capability.

5 DEFEATING KERNEL PRIVILEGE ESCALATION ATTACKS

We first derive a general attack model by analyzing the privilege escalation exploits which are still effective on the container platform, and then propose a defense system.

5.1 Kernel Privilege Escalation Attack Model

As illustrated in Table 3, with the default security mechanisms enforced by Docker, only 4 "privilege escalation" exploits work well inside the container. Another 7 exploits are blocked as the "*CAP_NET_ADMIN*" capability is by default not available inside the container. However, the "*CAP_NET_ADMIN*" capability controls the operations such as configuring the IP address, routing table, firewall etc., thus is likely to be allowed in practice. Therefore, we consider all the 11 exploits as the successful ones on the container platform.

By analyzing the 11 exploits, we find they follow a common 4-step attack model as depicted in Figure 2. First, they bypass (or manually disable) the KASLR mechanism to obtain the address of critical kernel static functions whose offset to the kernel base address is constant such as *native_write_cr4()*, *commit_creds()* and *prepare_kernel_cred()* etc. Second, they exploit the kernel vulnerabilities such as UAF, race condition, improper verification, buffer overflow etc., to enable the overwriting of the pointers of some kernel functions which could be easily triggered to execute. For example, the exploit with EDB-ID 41994 utilizes the "improper verification" vulnerability of the *setsockopt()* system call (i.e., CVE-2017-7308) to substitute the content of "retir_blk_timer->func" pointer with the address of *native_write_cr4()*. And the function addressed by "retir_blk_timer->func" will be triggered when the package is received too slowly. Third, they overwrite these kernel functions' pointers so that they point to the addresses of the critical kernel static functions (e.g., *native_write_cr4()*) or the kernel address of the first gadget of the crafted ROP chain. As such, the SMEP&SMAP mechanisms will be disabled when these kernel functions are invoked. This step could be omitted if the SMEP&SMAP mechanisms are manually disabled. Finally, they repeat the second step and overwrite the pointer of the kernel function to point to a malicious user space function or shellcode, which invokes the kernel function *commit_creds()* to apply for ROOT credential (i.e., ROOT privilege with 38 capabilities). As the SMEP&SMAP is disabled, the user space function or shellcode could be executed in supervisor mode.

In general, all "privilege escalation" exploits share the same attack goal, i.e., obtaining the full 38 capabilities. Theoretically, the attackers could achieve this by directly overwriting the kernel data associated with a process's capabilities. However, it is nearly impossible. First, it is difficult for the exploit codes in the user space to locate specific kernel data structures, as they are in different address space. Second, the user programs can only access kernel data through specified system calls, rather than write any kernel data directly. Therefore, almost all "privilege escalation" exploits need to find a vulnerable system call to enable the overwriting of some specific kernel data (i.e., step 2 in Figure 2), and rely on the supervisor-mode executed user space codes (i.e., step 3) to invoke the privilege-elevating kernel codes (i.e., step 4).

5.2 Countermeasures

According to the attack model, four actions are involved to elevate the privilege, which are bypassing the KASLR mechanism, bypassing the SMEP&SMAP mechanisms, overwriting pointers of some kernel functions which are easy to be invoked, and invoking the

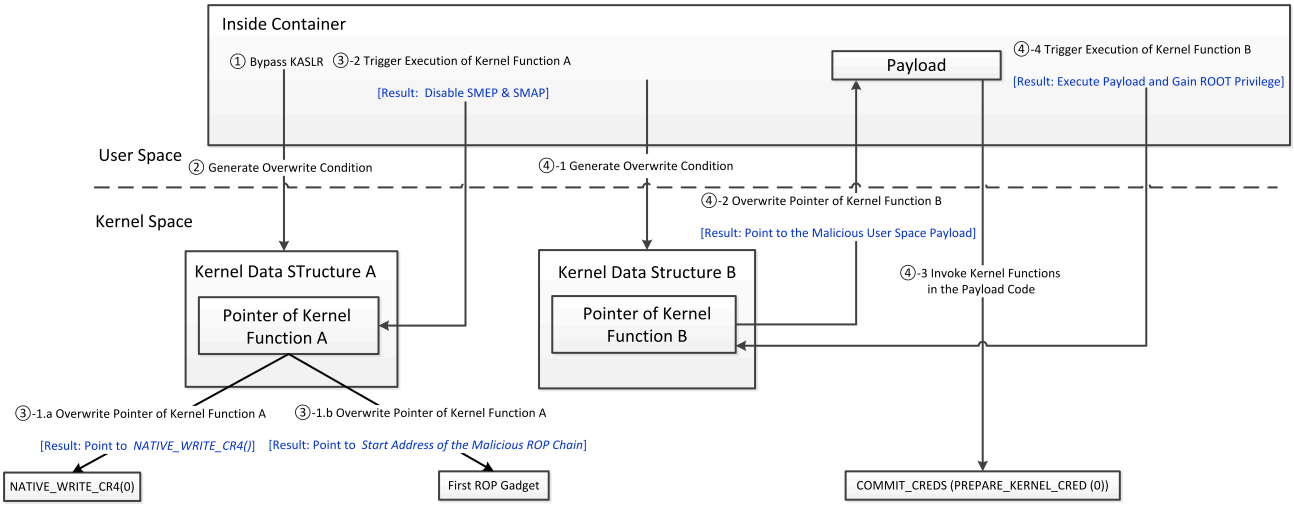

Figure 2: Kernel Privilege Escalation Attack Model

commit_creds() kernel function. Therefore, the "privilege escalation" attacks could be blocked by disabling any one of the four actions. However, it is difficult to find a general and comprehensive approach to prevent the attackers from bypassing the KASLR mechanism and overwriting kernel data. The KASLR mechanism has been proclaimed dead by many researchers, as current implementations of KASLR have fatal flaws [19]. Overwriting of the specific kernel functions' pointers is achieved by exploiting the vulnerabilities in the Linux kernel, such as UAF, race condition, improper verify, buffer overflow etc. And it is pretty unlikely to patch all vulnerabilities considering the large code size of Linux kernel. The CPU mechanisms SMAP&SMEP are easy to be disabled if the attackers compromise the KASLR mechanism and gain the ability to overwrite the pointers of some kernel functions. Therefore, we propose a defense system by forbidding the *commit_creds()* to be utilized to elevate the privilege inside the container.

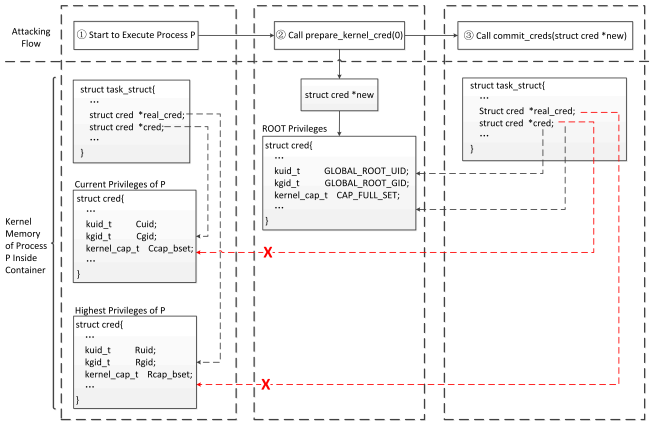

Figure 3: Privilege Escalation Procedure through *commit_creds()*

Privilege Escalation Procedure through *commit_creds()*. The procedure to elevate privilege through the *commit_creds()* is illustrated in Figure 3, which is also the detail of the fourth step in Figure 2. In Linux kernel, the credential associated with a process is stored as two fields inside the *task_struct* structure, i.e., *cred* and *real_cred*. The *cred* field represents the current privileges (including capabilities, GID, UID etc.) of the process, and could be temporarily modified during execution of the process. The *real_cred* field represents the highest privileges a process could reach, and normally could not be changed. Generally, two static kernel functions whose offset to the kernel base address is constant, are used to achieve the privilege escalation. First, the *prepare_kernel_cred(0)* function is invoked to construct a credential with ROOT privilege (i.e., *GLOBAL_ROOT_UID*, *GLOBAL_ROOT_GID*, and *CAP_FULL_SET*). Then, the *commit_creds()* function takes the return of the *prepare_kernel_cred(0)* as the parameter, and is invoked to update the *real_cred* and *cred* of current process with the parameter (i.e., ROOT privilege). Normally, invocation of *commit_creds()* by a non-privileged process will fail. However, in the attack model illustrated in Figure 2, *commit_creds()* can be executed successfully as the SMEP&SMAP mechanisms are disabled (step 3) and the user space program invoking the *commit_creds()* is triggered in the kernel mode (step 2 & step 4).

Defense System. We observe that the processes inside the container usually do not need ROOT privilege (By default, only portions of capabilities are assigned to the container tenants. For example, 14 capabilities are assigned to the Amazon container tenants [11], and 9 capabilities are assigned by the OpenShift container service [32]. And a normal container tenant is not allowed to apply for a container with more capabilities.). However, the privilege escalation attacks inside a container will misuse the *commit_creds()* to apply for the ROOT privilege (i.e., 38 capabilities). Therefore, we modify the implementation of the *commit_creds()* function, and enforce a check before updating the *real_cred* and *cred* of the caller process. Specifically, we first check whether the caller process is inside a container or on the control host. The process is described as a

Table 4: Performance of the Defense System

Benchmarks	1 Parallel Copy			4 Parallel Copies		
	Original	Modified	Overhead	Original	Modified	Overhead
Dhrystone 2 using register variables	3586.3	3558.6	0.77%	13308.0	13399.1	-0.68%
Double-Precision Whetstone	800.9	826.3	-3.17%	3369.8	3371.7	-0.06%
Execl Throughput	726.7	738.0	-1.55%	4346.2	4450.2	-2.39%
File Copy 1024 bufsize 2000 maxblocks	3379.1	3392.2	-0.39%	3857.3	3729.8	3.31%
File Copy 256 bufsize 500 maxblocks	2341.8	2329.2	0.54%	2390.8	2416.7	-1.08%
File Copy 4096 bufsize 8000 maxblocks	4930.0	4893.3	0.74%	6279.1	6204.6	1.19%
Pipe Throughput	2261.3	2264.5	-0.14%	8339.2	8489.1	-1.80%
Pipe-based Context Switching	77.7	77.9	-0.26%	4143.1	4163.9	-0.50%
Process Creation	686.1	681.2	0.71%	2211.3	2312.9	-4.59%
Shell Scripts (1 concurrent)	2498.7	2540.5	-1.67%	8947.2	9197.0	-2.79%
Shell Scripts (8 concurrent)	6570.0	6627.4	-0.87%	7683.3	8232.6	-7.15%
System Call Overhead	3161.3	3184.8	-0.74%	6345.5	6517.9	-2.72%
System Benchmarks Index Score	1681.1	1689.3	-0.49%	5183.4	5240.1	-1.09%

task_struct structure in Linux kernel which contains a *nsproxy* field storing the process's namespaces information (e.g., namespace ID). The processes on the Linux control host (e.g., the init process) are also associated with a *nsproxy* field whose value is however constant, i.e., *init_nsproxy*. Therefore, we can judge whether a process is inside a container or not, by comparing whether its *nsproxy* is equal to *init_nsproxy*. If the *commit_creds()* is invoked from inside a container, we will further check whether it is a privilege escalation operation, and stop modifying the *real_cred* and *cred* if it is. We determine whether it is a privilege escalation operation by comparing the input parameter (i.e., new credential) of the *commit_creds()* method and the *real_cred* of current process. If the uid/gid in the new credential is smaller or the *cap_bset* in the new credential is larger, it may be a privilege escalation operation. In total, our defense solution needs to add less than 10 lines of codes.

5.3 Effectiveness and Performance

We enable the "*CAP_NET_ADMIN*" capability for the processes inside the container, besides the default 14 capabilities assigned by Docker. We find all 11 exploits could obtain the ROOT privilege from inside the container before deploying the defense system, and fail to achieve privilege escalation after the deployment. The results prove that our defense system can effectively block the privilege escalation attacks on container platform.

We also evaluate the overhead of the defense system on a VMWare virtual machine with Ubuntu 16.04 LTS AMD 64 OS installed (Linux kernel 4.8.0#1), and configured with quad-core 2.80GHz CPU and 2GB memory. The host machine is Dell Precision 5520 with 8GB memory and quad-core 2.80GHz CPU. The results are illustrated in Table 4 and obtained through the UnixBench tool [66], which show that the defense system introduces negligible overhead.

6 DISCUSSION ON LIMITATION

We propose a simple but effective defense mechanism by hardening the *commit_creds* method, which is misused by existing privilege escalation exploits to modify the credentials bound to a process. However, it is not a complete solution against all potential privilege escalation exploits. First, after completing the third step in Figure 2 (i.e., bypassing the SMEP&SMAP mechanisms), the attackers may directly modify the kernel data related to *real_cred*. Since *real_cred* is stored at a fixed offset to the *task_struct* data structure, it could be modified once the *task_struct* is located. Second, armored attackers might launch ROP attacks that utilize the gadgets of kernel codes to modify the *real_cred* without first bypassing the SMEP&SMAP mechanisms. Furthermore, attackers may misuse other kernel functions besides *commit_creds* to launch other types of attacks such as container escape. For instance, the attackers might invoke the *switch_task_namespaces()* method to achieve container escape. Though our defense system can be extended to avoid the misuse of these kernel functions (e.g., forbidding the processes inside the containers to invoke the *switch_task_namespaces()* method for container switch), it is challenging to identify a complete list of all potential kernel functions that might be misused.

To defeat those potential attacks, a defense solution is better to be implemented outside Linux kernel. For instance, as a hypervisor-based kernel data protection mechanisms, Sentry [67] partitions the kernel memory into regions with different access control policies, which detail when the sensitive data in each region could be accessed by the kernel. AllMemPro [40] configures the read, write, and execution permissions on a memory page through the Extended Page Tables (EPT), and controls the EPT in the hypervisor. Alternatively, hardware-based kernel protection solutions are promising to prevent those attacks too. For example, PrivWatcher [9] utilizes ARM TrustZone mechanism to protect the process credentials against memory corruption attacks. We leave it as one future work to develop a more comprehensive defense solution.

7 RELATED WORK

7.1 Container Security

Existing research work on container security mainly focuses on three aspects, i.e., security of the container images [20, 60, 61, 65], security of the container orchestration tools like Docker [10, 28], and security of Linux container mechanism [7, 72]. Some researches also work on evaluating the security of container mechanism. For example, M. Ali Babar et al. [3] give a comparative analysis on the isolation mechanisms provided by three container engines, i.e., Docker, LXD, and Rkt. Thanh Bui et al. [6] briefly compare the security of hardware-based virtualization technology (e.g., XEN) and OS-level virtualization technology (i.e., container mechanism) from system architecture level. Reshetova et al. [62] theoretically analyze the security of several OS-level virtualization solutions including FreeBSD Jails, Linux-VServer, Solaris Zones, OpenVZ, LxC and Cells etc. However, these works mostly evaluate container security from the system architecture or design principle level, while we evaluate with a measurement approach.

Some researchers [46, 56] also evaluate the container security using potential vulnerabilities against specific container mechanisms such as Docker. For example, A. Martin et al. [50] do a vulnerability-oriented risk analysis of the container, classify the vulnerabilities into five categories and perform a vulnerability assessment according to the security architecture and use cases of Docker. A Mouat et al. [56] sort the vulnerabilities of container platform into kernel exploits, DoS, container breakouts, poisoned images, compromising secrets. Z. Jian et al. [38] summarize two models to achieve Docker container escape, propose a defense tool by inspecting the status of namespaces, and evaluate the tool with 11 CVE vulnerabilities. Due to the small number of exploits reported in the vulnerability

databases such as CVE [12] and NVD [58], it is challenging to provide a persuasive security evaluation on container mechanism. In this paper, we evaluate security of container in real world using an attack dataset containing 223 exploits.

There are also some studies on enhancing the security of container mechanism through the Linux kernel security mechanisms (e.g., Capability, Seccomp, MAC etc.) [41, 51] DockerPolicyModules [4, 8]. Our defense system is proposed based on analysis of the privilege escalation exploits.

7.2 Attack Taxonomy

Some researchers work on taxonomy of certain types of Linux attacks or vulnerabilities. For example, L. Yi et al. [42, 70] propose a two-dimensional taxonomy of network vulnerabilities in Unix/Linux Systems. S. Hansman et al. [22] propose a taxonomy that consists of four dimensions in the computer and network attack field. R. Sánchez-Fraga et al. [17] and J. Mirkovic et al. [55] make taxonomy for DDoS attacks based on highlight commonalities and important characteristics. N. Gruschka et al. [18] focus on taxonomy on cloud services attacks. D. Papp et al. [59] derive a taxonomy for attacks on embedded systems. Some researchers just analyze or classify the attacks without making a clear taxonomy. For example, K. Ko et al. [39] present the coarse characteristic classification and correlation analysis of source-level vulnerabilities in Linux kernel. In this paper, we propose a two-dimensional taxonomy for the attacks specifically for the container platform.

8 CONCLUSION

Linux container is increasingly utilized by the industrial community. Although it is a consensus that container mechanism is not secure, a concrete and systematical evaluation is absent. In this paper, we perform a measurement study on container security using an attack dataset containing 223 exploits. We first make a taxonomy from two dimensions. Then we evaluate the security of the container with the experiment results of 88 typical exploits filtered out from the dataset. Since we find the privilege escalation exploits can successfully escape outside the container and compromise the host, we give an in-depth analysis on them. Under the protection of security mechanisms, there are still 11 exploits can break the container isolation. Fortunately, we find a common attack model that all 11 exploits follows. Further more, we propose a defense mechanism to defeat all 11 exploits. In general, we prove that the security of container mechanism depends on the security of the kernel, while the interdependence and mutual-influence relationship requires careful configurations to effectively defeat privilege escalation attacks.

ACKNOWLEDGMENTS

We would like to thank our shepherd Zhiqiang Lin and our anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Key Research and Development Program of China under Grant No.2016YFB0800102, the National Natural Science Foundation of China under Grant No.61802398, the National Cryptography Development Fund under Award No.MMJJ20180222, the U.S. ONR grants N00014-16-1-3214 and N00014-16-1-3216, and the NSF grants CNS-1815650.

REFERENCES

[1] Aaron Adams. 2015. Xen SMEP (and SMAP) bypass. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/april/xen-smep-and-smap-bypass/
[2] The Kubernetes Authors. 2018. Production-Grade Container Orchestration. https://kubernetes.io/
[3] M Ali Babar and Ben Ramsey. 2017. Understanding Container Isolation Mechanisms for Building Security-Sensitive Private Cloud. *Technical Report, CREST, University of Adelaide, Adelaide, Australia* (2017).
[4] Enrico Bacis, Simone Mutti, Steven Capelli, and Stefano Paraboschi. 2015. DockerPolicyModules: Mandatory Access Control for Docker containers. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*. IEEE, 749–750. https://doi.org/10.1109/CNS.2015.7346917
[5] Mick Bauer. 2006. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal* 2006, 148 (2006), 13.
[6] Thanh Bui. 2015. Analysis of Docker Security. *CoRR* abs/1501.02967 (2015). arXiv:1501.02967 http://arxiv.org/abs/1501.02967
[7] Ramaswamy Chandramouli. 2017. *Security Assurance Requirements for Linux Application Container Deployments*. US Department of Commerce, National Institute of Standards and Technology.
[8] Jeeva Chelladhurai, Pethuru Raj Chelliah, and Sathish Alampalayam Kumar. 2016. Securing Docker Containers from Denial of Service (DoS) Attacks. In *IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016*. IEEE, 856–859. https://doi.org/10.1109/SCC.2016.123
[9] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. 2017. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*. 167–178. https://doi.org/10.1145/3052973.3053029
[10] Théo Combe, Antony Martin, and Roberto Di Pietro. 2016. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62. https://doi.org/10.1109/MCC.2016.100
[11] Amazon Company. 2018. AWS Fargate. https://aws.amazon.com/fargate/?nc1=h_ls
[12] MITRE Corporation. 2018. About CVE. https://cve.mitre.org/about/index.html
[13] MITRE Corporation. 2018. About CWE. https://cwe.mitre.org/about/index.html
[14] MITRE Corporation. 2018. CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). https://cwe.mitre.org/data/definitions/78.html
[15] Exploit Database. 2018. About The Exploit Database. https://www.exploit-db.com/about-exploit-db/
[16] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/
[17] Rolando Sánchez Fraga, Eleazar Aguirre Anaya, and Raúl Acosta Bermejo. 2014. Taxonomy for Denial-of-Service Vulnerabilities in the Linux Kernel. (2014).
[18] Nils Gruschka and Meiko Jensen. 2010. Attack Surfaces: A Taxonomy for Attacks on Cloud Services. In *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*. IEEE, 276–279. https://doi.org/10.1109/CLOUD.2010.23
[19] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clementine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. (2017), 161–176.
[20] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. 2015. Over 30% of official images in docker hub contain high priority security vulnerabilities.
[21] Serge E Hallyn and Andrew G Morgan. 2008. Linux capabilities: Making them work. In *Linux Symposium*, Vol. 8.
[22] Simon Hansman and Ray Hunt. 2005. A taxonomy of network and computer attacks. *Computers & Security* 24, 1 (2005), 31–43. https://doi.org/10.1016/j.cose.2004.06.011
[23] Qualys Research Team in Security Labs. 2017. The Stack Clash. https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt
[24] Docker Inc. 2018. AppArmor security profiles for Docker. https://docs.docker.com/engine/security/apparmor/
[25] Docker Inc. 2018. Docker overview. https://docs.docker.com/engine/docker-overview/
[26] Docker Inc. 2018. Docker Seccomp Profile. https://github.com/moby/moby/blob/master/profiles/seccomp/default.json
[27] Docker Inc. 2018. Install Docker CE from binaries. https://docs.docker.com/install/linux/docker-ce/binaries/
[28] Docker Inc. 2018. Protect the Docker daemon socket. https://docs.docker.com/engine/security/https/
[29] Docker Inc. 2018. Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/
[30] Docker Inc. 2018. WHAT IS A CONTAINER. https://www.docker.com/what-container
[31] Docker Inc. 2018. WHAT IS DOCKER. https://www.docker.com/what-docker
[32] Red Hat Inc. 2018. Red Hat OpenShift Online. https://www.openshift.com/products/online/

[33] VMware Inc. 2018. VMWare Airwatch BYOD. http://acestandard.org/zh-hans/solutions/bring-your-own-device-byod

[34] Wikimedia Foundation Inc. 2018. Docker (software). https://en.wikipedia.org/wiki/Docker_(software)

[35] Wikimedia Foundation Inc. 2018. LXC. https://en.wikipedia.org/wiki/LXC

[36] Wikimedia Foundation Inc. 2018. Supervisor Mode Access Prevention. https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention

[37] Wikimedia Foundation Inc. 2018. XML external entity attack. https://en.wikipedia.org/wiki/XML_external_entity_attack

[38] Zhiqiang Jian and Long Chen. 2017. A Defense Method against Docker Escape Attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, ICCSP 2017, Wuhan, China, March 17 - 19, 2017*. ACM, 142–146. https://doi.org/10.1145/3058060.3058085

[39] Kwangsun Ko, Insook Jang, Yong-hyeog Kang, Jinseok Lee, and Young Ik Eom. 2005. Characteristic Classification and Correlation Analysis of Source-Level Vulnerabilities in the Linux Kernel. In *Computational Intelligence and Security, International Conference, CIS 2005, Xi'an, China, December 15-19, 2005, Proceedings, Part II*. 1149–1156. https://doi.org/10.1007/11596981_172

[40] Igor Korkin. 2018. Hypervisor-Based Active Data Protection for Integrity and Confidentiality of Dynamically Allocated Memory in Windows Kernel. *CoRR* abs/1805.11847 (2018). arXiv:1805.11847 http://arxiv.org/abs/1805.11847

[41] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-Phase Execution of Application Containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*. Springer, 230–251. https://doi.org/10.1007/978-3-319-60876-1_11

[42] Yi Li and Xin-Ming Li. 2006. A New Taxonomy of Linux/Unix Operating System and Network Vulnerabilities. *Journal of Communication and Computer* 3, 8 (2006), 16–19.

[43] Xiaoli Lin, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. 2009. Threat Modeling for CSRF Attacks. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009, Vancouver, BC, Canada, August 29-31, 2009*. 486–491. https://doi.org/10.1109/CSE.2009.372

[44] Cellrox ltd. 2015. Cellrox Mobile Virtualization. http://www.cellrox.com/

[45] Canonical Ltd. 2018. LXC Introduction. https://linuxcontainers.org/lxc/introduction/#

[46] Tao Lu and Jie Chen. 2017. Research of Penetration Testing Technology in Docker Environment. (2017).

[47] Linux Man. 2017. Cgroup_namespaces-overview of Linux cgroup namespaces. http://www.man7.org/linux/man-pages/man7/cgroup_namespaces.7.html

[48] Linux Man. 2018. Capabilities - overview of Linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html

[49] Linux Man. 2018. Namespaces-overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html

[50] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. 2018. Docker ecosystem - Vulnerability Analysis. *Computer Communications* 122 (2018), 30–43. https://doi.org/10.1016/j.comcom.2018.03.011

[51] Massimiliano Mattetti, Alexandra Shulman-Peleg, Yair Allouche, Antonio Corradi, Shlomi Dolev, and Luca Foschini. 2015. Securing the infrastructure and the workloads of linux containers. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*. IEEE, 559–567. https://doi.org/10.1109/CNS.2015.7346869

[52] Bill McCarty. 2005. *Selinux: Nsa's open source security enhanced linux*. Vol. 238. O'Reilly. http://www.oreilly.de/catalog/selinux/index.html

[53] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[54] Microsoft. 2018. Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service/

[55] Jelena Mirkovic and Peter L. Reiher. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. *Computer Communication Review* 34, 2 (2004), 39–53. https://doi.org/10.1145/997150.997156

[56] A Mouat. 2015. Docker Security Using Containers Safely in Production.

[57] IAN MUSCAT. 2017. What is Server Side Request Forgery (SSRF). https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/

[58] NIST U.S. Department of Commerce. 2018. NVD. https://nvd.nist.gov/

[59] Dorottya Papp, Zhendong Ma, and Levente Buttyán. 2015. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *13th Annual Conference on Privacy, Security and Trust, PST 2015, Izmir, Turkey, July 21-23, 2015*. IEEE, 145–152. https://doi.org/10.1109/PST.2015.7232966

[60] K. C. Quest. 2018. docker-slim: Lean and mean docker containers. https://github.com/docker-slim/docker-slim

[61] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick D. McDaniel. 2016. Towards Least Privilege Containers with Cimplifier. *CoRR* abs/1602.08410 (2016). arXiv:1602.08410 http://arxiv.org/abs/1602.08410

[62] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. 2014. Security of OS-Level Virtualization Technologies. In *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*. 77–93. https://doi.org/10.1007/978-3-319-11599-3_5

[63] Samsung. 2018. Samsung Knox Workspace. https://www.samsungknox.com/en/solutions/it-solutions/knox-workspace

[64] Sconway. 2017. Kubernetes Continues to Move from Development to Production. https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/

[65] Rui Shu, Xiaohui Gu, and William Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*. ACM, 269–280. https://doi.org/10.1145/3029806.3029832

[66] Ben Smith, Rick Grehan, Tom Yager, and DC Niemi. 2011. Byte-unixbench: A Unix benchmark suite. *Technical report* (2011).

[67] Abhinav Srivastava and Jonathon T. Giffin. 2012. Efficient protection of kernel data structures via object partitioning. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*. 429–438. https://doi.org/10.1145/2420950.2421012

[68] James Turnbull. 2014. *The Docker Book: Containerization is the new virtualization*. James Turnbull.

[69] Nick Wilfahrt. 2016. Dirtycow vulnerability Details. https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails

[70] L. I. Yi, L. I. Xinming, and Xianggang Jiang. 2005. A Taxonomy of Software Vulnerabilities in Unix/Linux Systems. *Computer Engineering* 31, 6 (2005), 4–6.

[71] QIAN Yi, WANG Yi-jun, and XUE Zhi. 2012. ROP Attack and Defense Technology based on ARM. *Information Security & Communications Privacy* (2012).

[72] Xiaowei Zhao, Hong Yan, and Jiantong Zhang. 2017. A critical review of container security operations. *Maritime Policy & Management* 44, 2 (2017), 170–186.