Evaluation on the Security of Commercial Cloud Container Services

Yifei Wu[1,2], Lingguang Lei[1,2(✉)], Yuewu Wang[1,2], Kun Sun[3], and Jingzi Meng[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{wuyifei,wuyifei,wangyuewu,mengjingzi}@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] Center for Secure Information Systems, George Mason University, Fairfax, USA
ksun3@gmu.edu

Abstract. With the increasing adoption of the container mechanism in the industrial community, cloud vendors begin to provide cloud container services. Unfortunately, it lacks a concrete method to evaluate the security of cloud containers, whose security heavily depends on the security policies enforced by the cloud providers. In this paper, we first derive a metric checklist that identifies the critical factors associated with the security of cloud container services against the two most severe threats, i.e., the privilege escalation and container escaping attacks. Specifically, we identify the metrics which directly reflect the working conditions of the attacker. We also extract the metrics essential to achieve privilege escalation and container escaping attacks by investigating the feasible methods for breaking the security measures, including KASLR, SMEP and SMAP, etc. Since memory corruption vulnerabilities are frequently adopted in the privilege escalation attacks, we collect a dataset of the publicly released memory corruption vulnerabilities to assist the evaluation. Then, we develop a tool to collect the metric data listed in the checklist from inside the cloud containers and perform security inspection on five in-service commercial cloud container services. The results show that some containers are enforced with weak protection mechanisms (e.g., with the Seccomp mechanism being disabled), and the KASLR could be bypassed on all five cloud containers. However, even after obtaining ROOT privilege in a container, attackers still can hardly escape from the container on the public cloud platforms, since the necessary files for crafting or compiling a loadable kernel module for the host OS are inaccessible to the container. Finally, we provide some suggestions to improve the security of the cloud container services.

Keywords: Container · Privilege escalation · Kernel security mechanisms · CPU Protection Mechanisms · Container escape

W. Susilo et al. (Eds.): ISC 2020, LNCS 12472, pp. 160–177, 2020.
https://doi.org/10.1007/978-3-030-62974-8_10

1 Introduction

Container technology is increasingly adopted by the industrial community [38]. The primary reason is the flexibility introduced by the container orchestration tools such as Docker [11] and Kubernetes [15], which facilitate the deployment, scaling, and management of the containerized applications. The cloud vendors also begin to provide container services, e.g., Amazon Fargate [7], Google GKE (Google Kubernetes Engine) [12], etc. As a lightweight alternative to the traditional virtual-machine based cloud service, the cloud container service allows the container instances from different tenants to be executed on the same physical or virtual server.

As an OS-level virtualization technology implemented in the Linux kernel, all containers running on one host share the same Linux kernel. There is a consensus that the container mechanism is less secure than the traditional virtualization technology like Xen [17] and KVM [35], etc., due to its kernel-sharing feature. However, it lacks a concrete method to evaluate the security of the cloud container services. Existing studies mainly focus on analyzing the security of the containers running on the local platforms [16,19,27,37]. For example, XinLin et al. [27] provide a measurement study on the security of local Docker container systems, where the processes inside the container are granted with default Docker container permissions. Also, it assumes the attackers could configure portions of the underlying execution environment (e.g., they can select to install a vulnerable kernel system and obtain the image file and the source code of the kernel system). However, execution environment of the remote cloud containers is configured by the service providers and uncontrollable to the attackers. Therefore, security evaluation on the local container platforms could not completely reflect the security of various remote cloud containers, which are deployed with a dedicated kernel system and protection policies.

In this paper, we provide a metric-based method to evaluate the security of cloud container services against the privilege escalation attack (i.e., obtaining ROOT privilege from inside the container) and the container escaping attack, which will seriously damage or even invalid the isolation provided by container mechanism. We first investigate the critical factors associated with the security of a cloud container service and derive a metric checklist to facilitate the security inspection. Specifically, we identify the essential metrics associated with the cloud containers' execution environment, which directly reflect the working conditions of the attackers, including version and updating time of the underlying kernel system (they partially illustrate vulnerability of the underlying kernel system), permissions assigned to the container tenants, and the protection policies configured by the service provider (e.g., whether security measures including Seccomp [10], MAC, KASLR [21], SMEP [4] and SMAP [20] are enabled). We also extract the metrics critical to achieve privilege escalation by investigating the feasible methods for breaking the security measures, which are commonly adopted to defend against privilege escalation [27]. To aid our analysis, we collect a dataset of publicly released memory corruption vulnerabilities, which are nec-

essary for the privilege escalation attacks. Meanwhile, we analyze the procedure to achieve container escaping and identify the related critical metrics.

Then, we develop a tool to examine the identified metrics listed in the checklist and perform a detailed evaluation on the security of five popular cloud container services[1]. We first explore the execution environment of the cloud container services, and the results show that kernel systems of four container services are last updated in 2019. However, we find two cloud container services have assigned ROOT privilege to the container tenants (i.e., ccs4 and ccs5 in Table 1). The CPU mechanisms (i.e., SMEP and SMAP) are enabled by almost all container services, while the kernel protection mechanisms (i.e., MAC, Seccomp, and KASLR) are not effectively leveraged. For example, Seccomp and MAC are both enabled by only one service, and the KASLR is enabled by two services.

Investigation on the possibility of privilege escalation attack is performed on the containers that are not assigned ROOT privilege (i.e., containers provided by cloud services ccs1, ccs2 and cc3 in Table 1). The results show that KASLR could be successfully bypassed on all services. However, since the underlying kernel system of the three cloud containers was updated recently, we fail to find feasible memory corruption vulnerabilities (and exploits) to bypass SMEP and SMAP on ccs1, cc2 and ccs3. Experiments on the container escaping attacks show that container escaping is difficult on the public cloud platforms even after the attackers obtain ROOT privilege. Since there are no user-space APIs (e.g., system calls) for a process to transfer from one container to another container, container escaping should be achieved by modifying the kernel data. The most generic method to get into the kernel is through a kernel module. However, the Linux system only allows a matching kernel module (e.g., the module compiled with the same header files and symbol table as the running system) to be loaded. In our experiments, container escaping fails on the cloud containers, since the necessary files for crafting or compiling a loadable kernel module are inaccessible to the attackers inside the containers.

We have reported our findings to five cloud service providers, and received responses from most of the providers. After our suggestion, cp2 disabled the Intel TSX (Transactional Synchronization Extensions) [13] mechanism on the ccs2 service to prevent the bypassing of KASLR, and cp5 replied that they would constrain the tenant's capability and enable the KASLR to enhance the security of ccs5.

In summary, we make the following contributions:

– We present a metric-based method and design a tool to evaluate the security of cloud container services against the privilege escalation and container escaping attacks.

[1] As per requirement of some service providers, we use cp1, cp2, cp3, cp4, cp5 to represent the five cloud providers, and ccs1, ccs2, ccs3, ccs4, ccs5 to represent the five cloud container services in the evaluation results.

– We construct a dataset of memory corruption vulnerabilities which are usually necessary when achieving privilege escalation from inside the container to support the work of our evaluation tool.
– We evaluate the security of five in-service cloud container services in detail, and identify the major obstacles for the attackers to escape from public cloud containers. We also provide the suggestions to improve the security of the cloud container services based on our evaluation.

2 Background

2.1 Container Mechanism

Container [28] is a lightweight OS-level virtualization technology implemented in Linux kernel, which provides isolation for one or more Linux processes. The processes running inside a container feel like they own the entire system, although containers running on the same host share the same Linux kernel. Isolation between the containers is achieved through two kernel mechanisms, i.e., Namespace [9] and Cgroup [6]. There are seven types of namespaces, i.e., `user`, `uts`, `net`, `pid`, `mnt`, `ipc` and `cgroup`. Each namespace isolates a specific kernel resource for one container. For example, the `mnt` namespace provides an isolated file system for a container through 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. 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 [32] is a pervasively used container engine that facilitates the management of the containers, such as container creating, deleting, starting and stopping, etc. Popular cloud providers also begin to provide the multi-tenancy cloud container services, such as Amazon Fargate [7], Google GKE (Google Kubernetes Engine) [12], etc. The underlying technology of these services is the container mechanism. Therefore, several tenants might share the same Linux kernel.

2.2 Linux Kernel Security Mechanisms

Isolation enforced by the container mechanism is invalid, if a process inside the container compromises the kernel or escapes the container boundary to enter another container. Therefore, several Linux kernel security mechanisms are adopted to constrain the capability of the processes inside the containers, such as Kernel Address Space Layout Randomization (KASLR) [21], Capability [8], Seccomp [10] and Mandatory Access Control (MAC) mechanisms. The KASLR mechanism makes the Linux kernel boot up at a random base address rather than at a fixed base address. As such, the attackers could not obtain addresses of critical kernel functions, which are usually necessary to compromise the kernel. Capability is a privilege decentralized mechanism, which divides

the superuser privilege (i.e., ROOT privilege) into 38 units, known as capabilities. Each capability represents a permission to operate some specific kernel resources. The Seccomp mechanism constrains the system calls a process can invoke. SELinux [30], AppArmor [1] are two MAC mechanisms frequently used to enforce mandatory access control on the kernel resources.

2.3 CPU Protection Mechanisms

Two CPU protection mechanisms are also frequently used to protect the Linux kernel, i.e., Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP) [4]. SMAP prevents supervisor mode programs from accessing user-space memory, while SMEP prevents supervisor mode programs from executing user-space code. SMAP and SMEP could be enabled by setting the 21st and 20th bits of the CR4 register, respectively.

3 Metric Checklist for Container Security Evaluation

Before performing the security evaluation, we first derive a metric checklist that identifies the critical factors associated with the security of cloud container services. As a technology implemented in the Linux kernel, isolation introduced by container will be seriously damaged or even invalid, if the processes inside a container could obtain the ROOT privilege or escape the container boundary. Therefore, we focus on investigating the security of cloud container services against these two most severe threats, i.e., the possibility to achieve privilege escalation and container escaping from inside the container. Specifically, we first identify the metrics which directly reflect the cloud container's execution environment. Then, we investigate and summarize the feasible methods for breaking the security measures including KASLR, SMEP and SMAP, which are commonly adopted to defend against privilege escalation [27]. Finally, we extract the metrics essential to achieve container escaping.

3.1 Execution Environment Related Metrics

Security of the container services heavily depends on the execution environment, including version and updating time of the underlying kernel system, permissions assigned to the container tenants, and the protection policies configured by the service providers. Version and updating time of the underlying kernel system impact not only the probability of finding feasible memory corruption vulnerabilities, but also the possibility to obtain a matching kernel image, both of which are frequently leveraged in privilege escalation attacks [27]. Thus they are very important to the security of container services. The permission information directly reflects the ability of an attacker, which means the capabilities assigned to container processes on Linux platforms. The protection policies signify the difficulty of launching the attacks, which includes the configuration of both the Linux kernel (see Sect. 2.2) and CPU protection (see Sect. 2.3) mechanisms,

i.e., Seccomp, MAC, KASLR, SMEP and SMAP. As such, we introduce eight execution-environment-related metrics into the checklist, i.e., kernel version and updating time; capabilities assigned to a container tenant; and the policies of Seccomp, MAC, KASLR, SMEP and SMAP.

3.2 Privilege Escalation Related Metrics

The key security mechanisms against privilege escalation are KASLR, SMEP and SMAP [27]. KASLR prevents the attackers from guessing the kernel functions addresses (e.g., *commit_creds()* and *prepare_kernel_cred()* are two kernel functions frequently used in privilege escalation attacks). SMAP and SMEP can prevent the hijacked control flow from accessing the user-space data and executing the user-space code (shellcode). An attacker must bypass these mechanisms to achieve privilege escalation from inside the container [27]. In the following, we summarize the methods which could be used to bypass the KASLR, SMEP and SMAP, and extract the critical factors for achieving the bypassing.

1) Bypassing KASLR. The KASLR mechanism has been introduced since Linux kernel 3.14, which makes the kernel image decompress itself at a random location during the booting time. It can be enabled by setting the CONFIG_RANDOMIZE_BASE option when compiling the kernel, and it has been enabled by default since kernel 4.12. Without KASLR, the base address of the kernel code will be configured at $0 \times$ FFFFFFFF81000000. In theory, the number of slots available to the KASLR mechanism for achieving base address randomization is 256 on the $\times$ 86-32 platforms and 512 on the $\times$ 86-64 platforms [21]. In general, there are mainly two approaches to achieve KASLR bypassing, i.e., reading sensitive files and launching cache-based side-channel attacks.

a) *Bypassing KASLR through Reading Sensitive File.* Two types of files might be used to bypass KASLR. First, the `dmesg` file under the directory of `/var/log` may contain kernel-address related sensitive information (e.g., the kernel's base address might be obtained by searching the keywords such as "Freeing SMP" or "Freeing unused" in the `dmesg` file). However, we might not be able to obtain exact addresses of the critical kernel functions (e.g., *native_write_cr4()*) with only the kernel's base address, since the offsets of the kernel functions (to the base address) vary when the kernel images are compiled with different compilers (e.g., the `gcc` compilers of different versions) or different compiling options (e.g., the options defined in the `.config` file). Therefore, in order to obtain the exact addresses, kernel images of the running systems are also necessary. Second, the address of each kernel function could be obtained directly from the `/proc/kallsyms` file, if it is set as `readable` to the user.
b) *Bypassing KASLR through Cache-based Side-channel Attacks.* Since the low entropy of the KASLR's implementation, cache-based side-channel attacks are also frequently used to bypass KASLR. Basically, the attacks are launched

based on the observation that it takes less time to access a content residing in the cache than the one in the memory. And the most effective cache-based side-channel attack for bypassing KASLR is called TLB-cache-based [23,24] side-channel attack.

TLB-cache-based side-channel attacks are also known as double-page fault attacks [23,24], and they are accomplished based on a feature of some Intel CPUs. When a user program accesses a privileged kernel address, the processing procedure will be slightly different for the mapped and unmapped addresses. As illustrated in Fig. 1, when a mapped address is accessed, a TLB entry will be created in the TLB cache before the kernel delivers a segment fault signal to the user program (since the privilege check fails). But for an unmapped address, no TLB entry will be created. Therefore, the attackers can deduce whether a kernel address is mapped or not, by accessing the same address twice and comparing the time duration of receiving the segment fault signal. As such, the base address of the kernel image could be obtained by probing the whole region of the kernel space. However, the time to execute segment fault handler function is also counted into the duration (i.e., `t1` and `t2`), and it is usually far longer than the difference caused by TLB hit or miss. For obtaining stable results, it is better to reduce the noise caused by the segment fault handler as far as possible. Yeongjin Jang et al. [24] proposed a highly stable solution (named DrK) by leveraging the Intel TSX (Transactional Synchronization Extensions) instructions. With TSX, the CPU will directly inform the segment fault to the user program without the attendance of Linux kernel, as such the noise caused by segment fault handler is omitted.

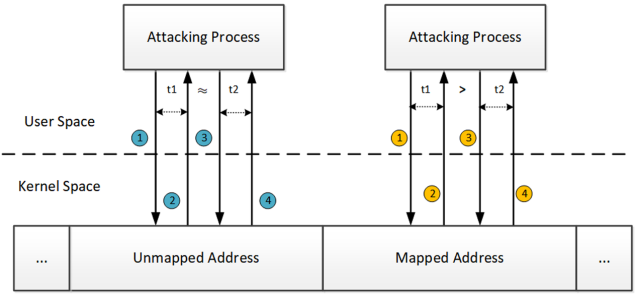

Fig. 1. TLB-cache based side-channel attack to bypass KASLR. ① Access a kernel address `p`; ② Receive a segment fault signal (since the privileged check fails) and record the time duration for obtaining the signal (`t1`); ③ Access `p` again; ④ Record the time duration for the second access (`t2`). When accessing a mapped address, a TLB entry will be created in step ②, and `t2` will be smaller than `t1` (since the TLB hit). Or else, no TLB entry will be created, then `t2` and `t1` will be almost the same.

2) Bypassing SMEP and SMAP. SMEP and SMAP are introduced into the Linux kernel since version 3.0 and 3.7, respectively. In general, the attackers can disable SMEP and SMAP by redirecting a corrupted kernel pointer to the *native_write_cr4()* function through memory corruption vulnerabilities. For example, with parameter `0x407f0`, *native_write_cr4()* will set the 20th and 21st bits of the CR4 register as zero (i.e., disabling the SMEP and SMAP). However, this method requires to leverage a memory corruption vulnerability in Linux kernel, i.e., seeking out (and overwriting) a corrupted kernel pointer that points to a function taking one and only one parameter. SMAP is sometimes disabled by default, then the SMEP could be disabled similarly but with loosen requirement of the corrupted kernel pointer (i.e., no additional requirement on the parameters). Specifically, the attackers can craft a malicious Return Oriented Programming (ROP) chain by concatenating exploitable kernel gadgets, and the chain realizes the similar function of *native_write_cr4()* (i.e., setting CR4 register). Then, they store the ROP chain as user-space data and redirect a corrupted kernel pointer to execute a "stack pivot" instruction, which will put the address of the ROP chain to the `esp` (Extended Stack Pointer) register and thereby make the ROP chain being executed. Although stored as user-space data, the chain could be read from kernel since SMAP is disabled. Also, the chain could be successfully executed since it is constructed by concatenating exploitable gadgets in the kernel space. However, the attackers need to bypass KASLR and obtain the kernel image of the running system (which is necessary for obtaining the accurate addresses of the exploitable kernel gadgets), before crafting a usable ROP chain.

On the whole, five factors are critical in compromising KASLR, SMEP and SMAP, which are the accessibility of `dmesg` and `/proc/kallsyms`, availability of the TSX instructions, and the possibility to find feasible memory corruption exploits and matching kernel images for the underlying Linux kernel system. These privilege-escalation-associated metrics are also introduced into the checklist.

3.3 Container Escaping Related Metrics

Although container escaping is easy on the local platforms after the attackers obtain the ROOT privilege, it is not an easy task on the public cloud platforms. There are no user-space APIs (e.g., system calls) for transferring a process from one container to another container, and a process' `container` attribute is defined through the data field (i.e., *nsproxy*) of the kernel data structure (i.e., *task_struct*). Therefore, container escaping could be achieved by modifying the kernel data. In general, there are two ways to get into the kernel from user-space after obtaining the ROOT privilege, i.e., finding and exploiting a feasible kernel memory corruption vulnerability, and crafting a loadable kernel module. The former needs a feasible vulnerability. The later needs a compiling environment for a kernel module or needs to bypass the verification of loading a kernel module.

Two verification will be performed before loading a kernel module, i.e., whether the module contains a `Vermagic` value matching the running kernel system,

and whether the kernel functions and structures utilized in the module are attached with correct CRC (Cyclic Redundancy Check) values [40]. `Vermagic` is a unique string that identifies the version of the kernel system on which the module is compiled. A kernel module could be successfully loaded when both checks pass. Therefore, the attackers can craft a loadable kernel module by either compiling it on the running systems (corresponding files such as kernel symbol table and kernel header files, etc. should be accessible), or compiling an incorrect kernel module and substituting the `Vermagic` and CRCs values with the correct ones. And kernel address of the memory accommodating each kernel function's (or structure's) CRC value usually could be obtained by reading the `/proc/kallsyms` file. For example, the address of kernel function A's CRC value is marked as `_kcrctab_A` in the `/proc/kallsyms` file. And the `Vermagic` value could be derived from an existing loadable kernel module. As such, the attackers could craft a loadable kernel module if they can find an existing loadable kernel module and obtain the `_kcrctab_*` values.

Based on the analysis above, we introduce three container-escaping-associated metrics into the checklist, which are the availability of the header files, `Vermagic` value and CRC value associated with the underlying Linux kernel system.

3.4 Memory Corruption Vulnerabilities

As illustrated in Sect. 3.2, when performing privilege escalation attack, the attackers need to overwrite certain kernel memory through memory corruption vulnerabilities in the Linux kernel, such as UAF (Use-After-Free), race condition, improper verification, buffer overflow, etc. It is pretty unlikely to patch all vulnerabilities considering the large code size of the Linux kernel. Distribution of memory corruption vulnerabilities partially reflects the possibility to achieve privilege escalation from inside the containers. Therefore, we provide a statistic analysis on the emerging and fixing pattern of the memory corruption vulnerabilities in this section.

1) Dataset. We collect a memory corruption vulnerability dataset by manually analyzing the vulnerabilities published on the National Vulnerability Database (NVD) between 2008 and 2018 [3]. NVD is the U.S. government repository of standards based vulnerability management data. Each vulnerability is assigned with a Common Vulnerabilities and Exposures (CVE) ID. First, we pick out all vulnerabilities used to compromise Linux kernel by investigating the vulnerability description on the NVD website. Then, we further find out the vulnerabilities which could be exploited to corrupt kernel memory. On one way, we will include all vulnerabilities which are explicitly stated the consequences of overwriting kernel memory or gaining privileges (through memory corruption), e.g., the vulnerabilities which use vulnerable system calls to generate UAF (Use-After-Free), race condition, buffer overflow, integer overflow, etc.

For the vulnerabilities without explicit statements of overwriting kernel memory, we analyze the work principles of the vulnerabilities to check whether they

could be exploited to corrupt kernel memory. For example, the descriptions of the vulnerabilities with ID CVE-2011-0709 and CVE-2017-8890 only state that they will cause DoS (Denial of Service) attacks through NULL pointer dereference and DF (double free), respectively. Since dereferencing of a NULL kernel pointer and double freeing of kernel memory have a high possibility of being exploited to cause kernel memory corruption [5], so they are also counted. To achieve a more accurate analysis, besides the descriptions on the NVD website, we also refer to the reports associated with the vulnerabilities on other websites, such as "SecurityFocus" [39] and "Red Hat Bugzilla" [14] (both websites are utilized by the technical communities to track bugs and discuss the details of the bugs).

Since our goal is to evaluate the possibility of launching attacks from inside containers, we exclude those vulnerabilities which are difficult to be exploited inside the container. For example, the vulnerabilities requiring the capabilities (e.g.,CAP_SYS_ADMIN) or system calls (e.g., ptrace(), bpf(), keyctl(), clone(), etc.) or operations (e.g., mounting a file system or image files) which are not available in the containers.

2) Number of Memory Corruption Vulnerabilities. In total, we find 374 kernel memory corruption vulnerabilities, and the number of each year is illustrated in Fig. 2. 54% (202) of vulnerabilities are explicitly stated that they could be exploited to corrupt kernel memory, while other 172 are identified by analyzing the vulnerabilities' work principle. On average, there are 34 memory corruption vulnerabilities each year. In addition, target kernel versions of the vulnerabilities change synchronously along with the updating of the Linux kernel. For example, the vulnerabilities published between 2012 and 2014 mainly target at Linux kernel 3.x, while the ones published in 2018 mainly focus on Linux kernel whose version is higher than 4.14. This shows that the memory corruption vulnerabilities are hardly to be cleared up even Linux kernel is continually

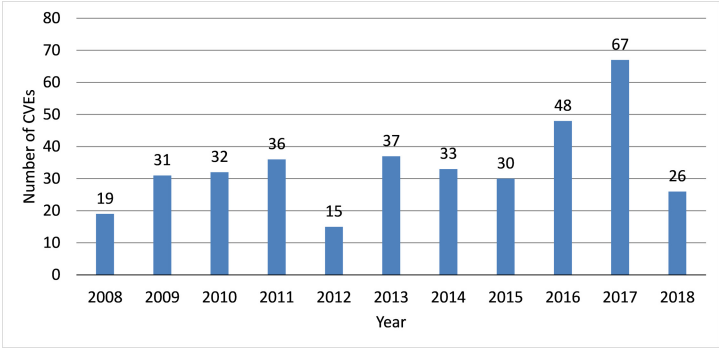

Fig. 2. Number of the memory corruption vulnerabilities

updated. Therefore, there is always a possibility to achieve kernel memory overwriting through kernel vulnerabilities.

As shown in Fig. 2, the number in 2017 is higher than others, and it is about twice of the average value. It is possible due to the dramatic increase of the total reported vulnerabilities in 2017 [31]. After further analysis, we find the reasons for the surge of the vulnerabilities in 2017 are mainly two-fold. First, the assignment process of CVE numbers was improved in 2017, where the CVE numbers could be assigned in a matter of hours or days through filling a web form. But before 2017, the assigning process is more tedious, and it takes far more time. Therefore, the higher number of vulnerabilities in 2017 does not necessarily mean that more vulnerabilities are discovered this year, but more researchers apply for and get CVE numbers successfully. Second, with the popularization of cloud computing, mobile Internet, and IoT devices in 2017, the generalization of cyberspace attacks and the lack of security awareness lead to an increase in the number of vulnerabilities [41].

3) Release Time of the Patches. Besides the vulnerability number, the time duration for an exposed vulnerability to be patched is also critical to the attackers. Therefore, we also analyze the patch release time for the 374 kernel memory corruption vulnerabilities identified. Normally, the patch for each vulnerability is also published on the NVD website, along with the vulnerability. In the situation when more than one patches are released for a vulnerability, the earliest release time will be utilized. Figure 3 depicts the statistics results of the patch release time, which shows more than 97% of vulnerabilities are patched within 5 months after the CVE numbers are assigned. And we are not able to find patches for 4 vulnerabilities, i.e., the ones labeled as "Unknown" in Fig. 3. We find patches of about 52% of vulnerabilities are released before the CVE numbers are assigned. The reasons might be two-fold. First, it takes a long time for the CVE number to be reviewed and assigned, so there is a lag. The researchers who discover

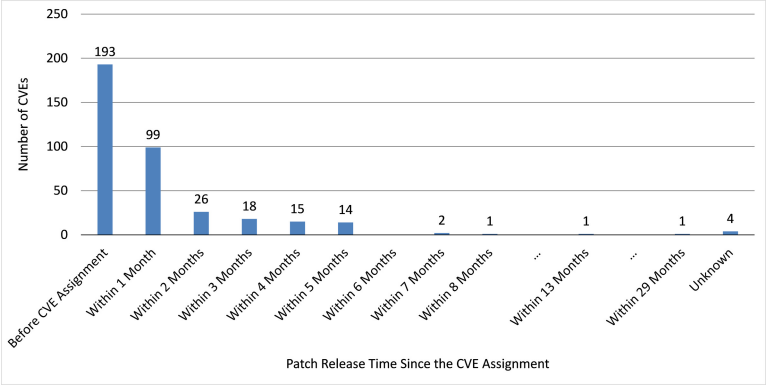

Fig. 3. Statistics on the patch release time since the CVE assignment

the vulnerability (or those who are aware of the vulnerability) have developed and published a patch before the CVE number is assigned. Second, the CVE assignment is intentionally delayed, as such the attackers could not utilize the published vulnerability to launch zero-day attacks.

4 Evaluation and Analysis

In this section, we perform an evaluation on five popular cloud container services, which are increasingly utilized to deploy industrial applications [36]. Particularly, we develop a tool to facilitate the collection of metric data listed in the checklist from inside the cloud containers. Most metric data could be obtained through existing Linux commands, e.g., the capabilities assigned to a container process could be fetc.hed through the `getpcaps` command, the kernel version could be obtained through the `uname -a` command. We investigate the availability of memory corruption vulnerability by checking the underlying kernel version against the dataset collected from NVD (see Sect. 3.4). To explore whether the kernel image and header files of the underlying kernel system are available, we collect a repository which includes kernel image files and header files downloaded both from the virtual machines of these cloud service providers and the Linux Kernel Archives [2].

All container services have the concept of regions, i.e., the containers might be deployed on servers located at different physical regions. For example, ccs5 allows a user to apply for a container from one of the three regions. As such, we randomly select three regions for each container service and investigate whether the configuration varies for the containers deployed on servers located at different physical regions. The results show that containers deployed in different regions share the same configuration. Therefore, we evaluate one representative container for each service. All data associated with the container services were collected in August 2019.

4.1 Container Execution Environment Detection

Table 1. Execution Environment of the Cloud Containers

Cloud container service	Kernel version		Permissions	Protection mechanisms				
	Version	Update date	No of Caps	Seccomp	MAC	KASLR	SMEP	SMAP
ccs1	4.14	2019/06	14	√	×	×	√	×
ccs2	4.14	2019/06	14	×	√	√	√	√
ccs3	4.15	2019/05	14	×	×	√	√	√
ccs4	3.10*	2018/04	37†	×	×	×	√	√
ccs5	4.1.51	2019/02	38	×	×	×	√	√

√ represents the protection mechanism is enabled, and × means it is disabled.
* The underlying kernel system is Red Hat.
† Since CAP_AUDIT_READ is not supported in this kernel system, 37 capabilities represent ROOT privilege.

Execution environments of the cloud containers are illustrated in Table 1. First, kernel systems of all container services except ccs4 are last updated in 2019. However, we find ccs4 and ccs5 have already assigned ROOT privilege to the container tenants. The CPU mechanisms (i.e., SMEP&SMAP) are enabled by almost all container services, while the kernel protection mechanisms are not effectively leveraged. For example, Seccomp and MAC are both enabled by only one service, and the KASLR is enabled by two services. A study of the privilege escalation vulnerabilities [27] shows that 11 exploits are blocked by Seccomp and MAC. Furthermore, KASLR is a necessary step for privilege escalation. Improperly setting of these mechanisms might let pass a series of exploits.

4.2 Privilege Escalation Evaluation

Table 2 depicts the possibility to achieve privilege escalation from inside the cloud containers. Since ROOT privilege has already been assigned to the container tenants of ccs4 and ccs5 as illustrated in Table 1, we investigate the possibility of privilege escalation attacks on the other three cloud container services. As illustrated before, the essential problems for achieving privilege escalation are bypassing KASLR, SMEP and SMAP. From Table 2 we can see that the KASLR could be successfully bypassed on all services through either reading the `/proc/kallsyms` file or conducting TLB-cache based side-channel attacks with the help of Intel TSX mechanism (The `/proc/kallsyms` file is also accessible on the containers provided by ccs4 and ccs5). As for the bypassing of SMEP and SMAP, we can obtain the feasible kernel images to craft an ROP chain for the containers provided by ccs1 and ccs2, since both services utilize the same kernel images for the virtual machines and containers. However, since kernel systems of the three container services were updated recently, we fail to find feasible memory corruption vulnerabilities (and exploits) to bypass SMEP and SMAP.

Table 2. Results of Privilege Escalation Attacks on the Cloud Containers*

Cloud Container Service	Bypassing KASLR				Bypassing SMEP&SMAP		
	dmesg	/proc/kallsyms	TSX	Success	Feasible Exploits	Kernel Image	Success
ccs1	√	√	×	Y	×	√	N
ccs2	√	×	√	Y	×	√	N
ccs3	√	×	√	Y	×	×	N

√ represents the item is accessible (or available) to the attackers, and × means it is inaccessible (or unavailable).
* "ccs4" and "ccs5" are not illustrated, since the containers of these two services have already been assigned ROOT privilege.

4.3 Container Escaping Evaluation

Table 3. Results of Container Escaping on the Cloud Containers*

Cloud Container Service	Compiling Environment	Bypassing Verification	
	header files	Vermagic(loadable module)	CRC(_kcrctab_*)
ccs4	×	√	×
ccs5	×	×	√

√ represents the item is accessible (or available) to the attackers, and × means it is inaccessible (or unavailable).
* "ccs1" , "ccs2" and "ccs3" are not illustrated, since lacks of feasible vulnerability for the containers of these three services.

The results of container escaping are shown in Table 3. Due to the lacking of feasible vulnerability, we use the second method to get into the kernel. Because ccs4 and ccs5 have already assigned the container tenants ROOT privilege, we have the capability to load module. We find the kernel header files are inaccessible to the containers for both services, so we could not directly compile a loadable kernel module. Meanwhile, we find the `_kcrctab_*` values are absent on the containers of ccs4 (although the `/proc/kallsyms` file is accessible), while no existing loadable kernel modules could be found in the containers provided by ccs5.

5 Discussion and Future Work

We give some suggestions to enhance the security of cloud container services from the following aspects. First, the kernel mechanisms including Seccomp, Capabilities and MAC should be enabled and set with as strict policies as possible, which might block a series of exploits. For example, in the study performed by XinLin et al. [27], 67.57% of exploits are blocked by these kernel mechanisms. Second, the KASLR mechanism should be effectively utilized by not only being enabled, but also with the sensitive files (including `dmesg`, and `/proc/kallsyms` etc.) set as inaccessible for the container tenants. Third, vulnerabilities in the underlying kernel system should be patched as soon as possible, which will increase the difficulty for the attackers to seek out a usable exploit. Fourth, it is better to use the kernel images of different versions in the virtual machines from the ones in the containers, if the service provider allows the tenants to apply for both virtual machines and containers. This can prevent the attackers from crafting a feasible ROP chain for bypassing SMEP, and also raise the bar to achieve container escape. We have supplied these suggestions to the cloud service providers.

It is more challenging to achieve privilege escalation on the public cloud platforms than on local Docker platforms, since the lack of available exploits on the specific underlying kernel systems. However, according to our research, tens of memory corruption vulnerabilities are published each year, and there is

a lag for the vulnerabilities to be patched. Therefore, by continuously collecting the emerging exploits and tracking the updating states of the cloud container services' underlying kernel systems, it is possible for attackers to obtain ROOT privilege from inside the containers. Security of the cloud container services heavily depends on whether a vulnerable kernel system is updated in time and how long is the lag. However, the problem has not been well studied yet. Also, more persuasive evaluation could be obtained if the memory corruption vulnerability dataset could be greatly enlarged. We leave them as our future work.

6 Related Work

There are a line of research works on the security of the container mechanism. For example, M. Ali Babar et al. [16] compared the security between the containers based on different OS-level virtualization implementations, i.e., Rkt, Docker and LXD. Thanh Bui et al. [19] compared the architecture between hypervisor-based virtualization and container-based virtualization briefly, and they mainly analyzed the docker internal security. XinLin et al. [27] used the vulnerabilities to measure the security of the local container, and they analyzed the influence of different capabilities on container vulnerability exploitation. Reshetova et al. [37] theoretically analyzed the security of different OS-level virtualization implementations, i.e., FreeBSD Jails, Linux-VServer, Solaris Zones, OpenVZ, LxC and Cells. Z Jian et al. [25] summarized two approaches to achieve container escape and evaluated the proposed defense tool with 11 CVE vulnerabilities. A. Martin et al. [29] classified the vulnerabilities of the container to five categories and performed a vulnerability assessment based on the security architecture and use cases of Docker. A. Mouat et al. [33] provided an overview of some container vulnerabilities, such as kernel exploits, container breakouts and secret leakage. Different from these works, we focus more on the security of the remote cloud containers, which is more complicated and varies on different cloud container platforms.

Many researchers also investigate the security of cloud container orchestration tools [18,34]. Alexander et al. proposed a method to detect the container environment [26], i.e., comparing the number of processes returned by the *sysinfo()* system call and the "ps -ef" command. Xing Gao et al. [22] proposed that the leaked host information will seriously threaten the security of the cloud server. They also introduced a leakage channel detection method based on the context and listed the leakage channels.

7 Conclusion

Cloud container service is widely used, so its security is particularly important. In this paper, we provide a concrete method to evaluate the security of cloud container services. We also perform a detailed evaluation of five in-service cloud container services, i.e., whether the user can achieve privilege escalation

from inside the cloud containers, and the possibility to achieve the cloud container escaping. We find some incorrect configurations in them (e.g., two cloud container services have assigned ROOT privilege to their container tenants by default). Moreover, the KASLR mechanism could be successfully bypassed on all five cloud containers. However, even after obtaining ROOT privilege in a container, attackers still can hardly escape from the container on the public cloud platforms. Finally, we give some suggestions to improve the security of the cloud container services.

Acknowledgment. We thank the anonymous reviewers for their insightful comments on improving our work. This work is partially supported by National Key R&D Program of China under Award No. 2018YFB0804402, the National Natural Science Foundation of China under GA No. 61802398, the National Cryptography Development Fund under Award No. MMJJ20180222 and the NSF grant CNS-1815650.

References

1. Apparmor security profiles for docker. https://docs.docker.com/engine/security/apparmor/
2. The linux kernel archives. https://www.kernel.org
3. National vulnerability database. https://nvd.nist.gov/
4. Supervisor mode execution prevention. https://en.wikipedia.org/wiki/Control_register#SMEP
5. Cve-2016-2384 (2016). https://xairy.github.io/blog/2016/cve-2016-2384
6. Cgroup_namespaces-overview of linux cgroup namespaces (2017). https://www.man7.org/linux/man-pages/man7/cgroup_namespaces.7.html
7. Aws fargate (2018). https://aws.amazon.com/fargate
8. Overview of linux capabilities (2018). http://man7.org/linux/man-pages/man7/capabilities.7.html
9. Overview of linux namespaces (2018). http://man7.org/linux/man-pages/man7/namespaces.7.html
10. Seccomp security profiles for docker (2018). https://docs.docker.com/engine/security/seccomp/
11. What is docker (2018). https://www.docker.com/what-docker
12. Google kubernetes engine (2019). https://cloud.google.com/kubernetes-engine/
13. Intel transactional synchronization extensions (intel tsx) overview (2019). https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview
14. Red hat bugzilla (2019). https://bugzilla.redhat.com/
15. Authors, T.K.: Production-grade container orchestration (2018). https://kubernetes.io/
16. Babar, M.A., Ramsey, B.: Understanding container isolation mechanisms for building security-sensitive private cloud. Technical Report, CREST, University of Adelaide, Adelaide, Australia (2017)
17. Barham, P., et al.: Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. **37**(5), 164–177 (2003)
18. Bernstein, D.: Containers and cloud: from LXC to docker to kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)

19. Bui, T.: Analysis of docker security. CoRR abs/1501.02967 (2015)
20. Corbet, J.: Supervisor mode access prevention (2012). https://lwn.net/Articles/517475/
21. Edge, J.: Kernel address space layout randomization (2013), https://lwn.net/Articles/569635/
22. Gao, X., Gu, Z., Kayaalp, M., Pendarakis, D., Wang, H.: Containerleaks: emerging security threats of information leakages in container clouds. In: 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26–29, 2017. pp. 237–248 (2017)
23. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: 2013 IEEE Symposium on Security and Privacy, pp. 191–205, May 2013
24. Jang, Y., Lee, S., Kim, T.: Breaking kernel address space layout randomization with intel TSX. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016, pp. 380–392 (2016)
25. Jian, Z., Chen, L.: 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, pp. 142–146 (2017)
26. Kedrowitsch, A., Yao, D.D., Wang, G., Cameron, K.: A first look: Using linux containers for deceptive honeypots. In: Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense, pp. 15–22. ACM (2017)
27. Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., Zhou, Q.: A measurement study on linux container security: attacks and countermeasures. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 418–429. ACSAC 2018, ACM, New York, NY, USA (2018)
28. Ltd, C.: Lxc introduction (2018). https://linuxcontainers.org/lxc/introduction/
29. Martin, A., Raponi, S., Combe, T., Pietro, R.D.: Docker ecosystem - vulnerability analysis. Comput. Commun. **122**, 30–43 (2018)
30. McCarty, B.: Selinux: Nsa's open source security enhanced linux, vol. 238. O'Reilly (2005). http://www.oreilly.de/catalog/selinux/index.html
31. Ángel Mendoza, M.: Vulnerabilities reached a historic peak in 2017 (2018). https://www.welivesecurity.com/2018/02/05/vulnerabilities-reached-historic-peak-2017/
32. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
33. Mouat, A.: Docker security using containers safely in production (2015). https://www.oreilly.com/content/docker-security/
34. Pahl, C., Brogi, A., Soldani, J., Jamshidi, P.: Cloud container technologies: a state-of-the-art review. In: IEEE Transactions on Cloud Computing (2017)
35. Qumranet, A., Qumranet, Y., Qumranet, D., Qumranet, U., Liguori, A.: KVM: the linux virtual machine monitor. In: Proceedings Linux Symposium, vol. 15 (2007)
36. Reports, H.C.R.: Containers as a service market research report - global forecast 2023 (2019). https://www.marketresearchfuture.com/reports/containers-as-a-service-market-4611
37. Reshetova, E., Karhunen, J., Nyman, T., Asokan, N.: Security of OS-Level virtualization technologies. In: Bernsmed, K., Fischer-Hübner, S. (eds.) NordSec 2014. LNCS, vol. 8788, pp. 77–93. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11599-3_5
38. Sconway: Kubernetes continues to move from development to production (2017). https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/

39. SecurityFocus: Securityfocus (2019). https://www.securityfocus.com/
40. Stoler, N.: How i hacked play-with-docker and remotely ran code on the host (2019). https://www.cyberark.com/threat-research-blog/how-i-hacked-play-with-docker-and-remotely-ran-code-on-the-host/
41. Vertical, horizontal data: Inventory of cyber security vulnerabilities in 2017: The number of vulnerabilities has grown unprecedentedly and may occur at all levels (2017), https://news.zoneidc.com/679.html