1 Abstract

Recently, many companies use containers to run their microservices, since containers could make their hardware resources be used efficiently. For example, GCP(Google Cloud Platform), AWS(Amazon Web Services), and Microsoft Azure are using this technique to separate subscribers' resources and services. However, if the hacker gets privilege escalation of containers, then such attacks would influence the host or the other containers. Therefore, this research would analyze, implement, and protect the container escalation in healthcare data exchange system. The container escalation would be inspired in a container and influence the host or the other containers. The healthcare data exchanging system will take the FHIR(Fast Healthcare Interoperability Resources) [1] to simulate the real-world threat and purpose a secure solution to protect patient's privacy.

2 Motivation

The Container is a virtualization technique to package applications and dependencies to run in an isolated environment. Containers are faster to start-up, lighter in memory/storage usage at run time, and easier to deploy than virtual machines. Because the container shares the kernel with the host OS and other containers and deploys by a configure file.
First, we often used to run a docker container to host our services. For example assignments, , servers and some services in the information security club at NSYSU(National Sun Yat-sen University). But there are some threats to the container technique. Like "Dirty CoW" [2] and "Escape vulnerabilities".
Dirty CoW is a vulnerability in the Linux kernel. It is a local privilege escalation bug that exploits a race condition in the implementation of the copy-on-write mechanism in the kernel's memory-management subsystem [3]. It was founded by Phil Oester. We were 16, the first year we had touched the docker container. We tried to use the Dirty CoW vulnerability to take the root privilege of my Android phone.
Escape vulnerability is a subcategory of sandbox security. At first, security researchers often need a sandbox to help them analyze malware, which prevents the malware influence researcher's host OS. Nowadays, the sandbox not only is used in analyzing, but also used to execute a normal application for an isolated environment. However, if the application could modify the outside resources without the kernel permission. That loses the purpose of isolation. That might cause the information to leak or the kernel is hacked.
Hence, there is a big problem about: "How to make sure my services are isolated and secure?" The author of this paper is the leader of the information security club. He should maintain all the services working perfectly. Moreover, we are an information security club. Therefore, the security and performance issue is the top-priority requirement.
Second, to present the container security, we would take the medical system for example. The medical system is the most famous part internationally. Including face the COVID-19 in Taiwan, we do not have the local COVID-19 case in more than 250 days. [4] However, the medical profession needs to renew the exchanging EHR(Electronic Health Records) system these years. To protect the privacy of patients, and producing a high-performance system to exchange the EHR, we need an easy deployment, effective runtime, and a secure system. We have to do this research in this project.

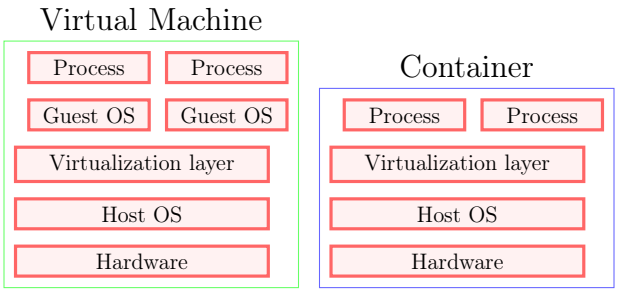


Figure 1: Comparison between containers and virtual machines

3 Related works

This section will focus on (I) concepts (II) containter security, and (III) high-performance server.

3.1 Concepts

3.1.1 Virtual machines and containers

Different names are used to refer to containers in the literature including OS-level virtualization and lightweight virtualization [5]. However, virtual machines are providing the functionality of a physical computer. The virtual machines would take a copy of the guest OS, and execute with the virtualization layer on the host OS. Which difference shows in Figure 1.

3.1.2 FHIR

The FHIR is a standard for healthcare data exchange. The FHIR standard will be used in in Taiwan in the near future [6]. The FHIR will be used to provide the PHR (Personal Healthcare Records) in Taiwan. Therefore we choose the most popular standard "FHIR" for the target.

3.1.3 Linux kernel features

First, there are 4 basic features for the abstract containerization services in the Linux environment. The 4 basic features are: (I) namespace, (II) cgroups, (III) capabilities, and (IV) seccomp.

Second, there are 3 terminologies of computer science related to container security in the Linux kernel. Which are (V) mmap, (VI) Copy on write, and (VII) Race condition.

And finally, there are 2 I/O Linux system calls that would be considered to enhance the I/O performance. Which are the (VIII) epoll and (IX) io_uring.

namespaces The Linux kernel provides the namespaces to perform the job of isolation and virtualization of system resources for a collection of processes [5]. User namespaces can be nested; that is, each user namespace—except the initial ("root") namespace—has a parent user namespace and can have zero or more child user namespaces. [7] The nested namespace would look like the Figure 2.

cgroups This feature can limit, account for, and isolate the hardware resource usage of a collection of processes [8]. The container could use this feature to set the maximum/minimum usage of hardware resources, which could guarantee processes' resources using reasonability.

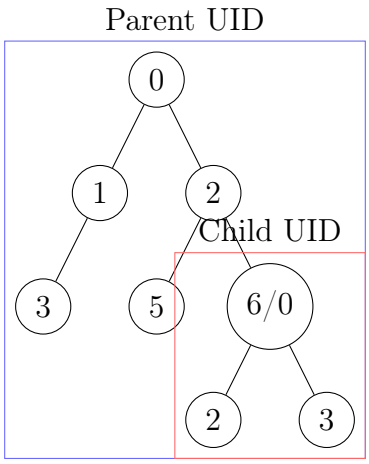


Figure 2: The nested user namespaces

capabilities This feature divides the privileges traditionally associated with superuser into distinct units. To performing permission checks, traditional UNIX implementations distinguish two categories of processes: privileged and unprivileged. Privileged processes would bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials [9].

Take ping command for example. The ping needs to generate and receive ICMP packets, and usually that's done using the "raw sockets" – a feature limited to root only (CAP_NET_RAW). Because it could also be abused to sniff and disrupt other traffic on the system. And we can set the capability to the file to get accessibility to execute the file. Therefore, when we set the CAP_NET_RAW capability to the /bin/ping, we could use the ping command as your user.

seccomp The seccomp feature is that only some specified process could call some specified system calls. We could set a policy of while some file be loaded, and we often use this system call to enforce the whitelisting or blacklisting policy.

mmap This is a system call of mapping files or devices in to memory, which creates a new mapping in the virtual address space of the caller process. Such that the process could operate the instance of file in memory directly. And some libraries are also mapped into the virtual address space to share and handle the function call. Therefore, the processes could take the same view of libraries in it's memory space.

copy on write This mechanism purposes of a resource being duplicated but not modified, it is not necessary to create a new entry. Therefore the kernel can make callers share the same memory resources. The mmap is a system call could inspire this mechanism in above paragraph. When some processes request same memory resource, the kernel supplies the same memory page to callers.

race condition That is processes or threads are racing the same mutable resource. For example: There is an accessible and mutable shared memory which be initialized as 0. And 2 threads or processes sharing that page. Consider one of the tasks is assigning the page full of character 'A'. In the meanwhile, the schedular context switches to the other task, which assigns that page full of 'B'. Then the schedular context switches again to the first task. There is a problem now. What is the page for the first task looked like? It does not meet the expectation for the first task. This is race condition.

epoll The epoll system call is a scalable I/O event notification facility, which similar task to poll and select. Its function is to monitor multiple file descriptors to see whether I/O is possible on any of them. Which uses RB-tree to search the monitored file descriptor.

io_uring The io_uring system call is a new feature in the Linux kernel 5.1. Which is also an asynchronous I/O API, supplying larger throughput and lower latency. This system call has 3 key elements: Submission Queue (SQ), Completion Queue (CQ), and Submission Queue Entries (SQE).

When we operate the io_ring, we should use the mmap to share the memory page from the returned file descriptor by the io_setup. After the page is mmaped, we could share the 3 key elements with the Linux kernel. The Linux kernel will pick up our submission from the SQE and polling in the kernel thread. That is, we have no use for a system call to tell the kernel that we want to have an asynchronous I/O.

After the kernel finishes the I/O operation, the kernel will put the index of the finished file descriptor to the completion queue. The user could traverse over the CQ to pick up the finished file descriptor. Which also reduced a system call to tell the kernel we picked it up.

Therefore, we have not required any system call on submission and completion in optimal.

3.2 Containter Security

3.2.1 Study of the Dirty Copy On Write

This paper [10] show the race condition, and the mechanism of "copy on write". "Copy on Write" is "a resource-management technique used in computer programming to efficiently implement a "duplicate" or "copy" operation on modifiable resources." [11] It often inspires when fork() or mmap().

3.2.2 Dirty CoW demo code

Let's analyze the proof of concept (PoC) of dirty CoW. (Oester, 2016) [2] The key to inspiring this vulnerability is the mmaped memory space, which is mapped with the PROT_READ flag. The PROT_READ flag declares the page is read-only.

```
87  f=open(argv[1],O_RDONLY);
88  fstat(f,&st);
89  name=argv[1];
90  map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
```

src/dirtyc0w.c

It creates 2 threads, which would have a race condition of the mmaped memory space, madviseThread and procselfmemThread.

threads in main

```
106 pthread_create(&pth1,NULL,madviseThread,argv[1]);
107 pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
```

src/dirtyc0w.c

In one thread, call a system call "madvise", would make the user thread gain the root privilege to operate the protected page temporarily. And the flag MADV_DONTNEED would tell the kernel: "Do not expect to access it in the near future. [12]" Moreover, this flag might not lead to immediate freeing of pages in the range. The kernel is free to delay free the pages until an appropriate moment. [12]

madviseThread

```
33 void *madviseThread(void *arg)
34 {
35   char *str;
36   str=(char*)arg;
37   int i,c=0;
38   for(i=0;i<100000000;i++)
39   {
40     c+=madvise(map,100,MADV_DONTNEED);
41   }
42   printf("madvise %d\n\n",c);
43 }
```

src/dirtyc0w.c

In another thread, open its memory resource file. This file is a special file, which allows the process reads its memory by itself.
Then, we move the printer of file descriptor of the memory resource file to the mmaped space. And try to write it. But the mmaped space is read-only space. We expected the kernel would create a copy of this space and write the copy [13].
procselfmemThread

```
50 void *procselfmemThread(void *arg)
51 {
52   char *str;
53   str=(char*)arg;
54   int f=open("/proc/self/mem",O_RDWR);
55   int i,c=0;
56   for(i=0;i<100000000;i++) {
57     lseek(f,(uintptr_t) map,SEEK_SET);
58     c+=write(f,str,strlen(str));
59   }
60   printf("procselfmem %d\n\n", c);
61 }
```

src/dirtyc0w.c

But there is a problem! There is another thread that is racing this page with root privilege. If the schedular context switches the madviseThread to procselfmemThread, while the adviseThread is calling the "madvise" system call. It would cause the procselfmemThread to gain the root privilege from madviseThread to control the mmaped file.

3.2.3 Container Security: Issues, Challenges, and the Road Ahead

This paper [5] has derived 4 generalized container security issues: (I) protecting a container from applications inside it, (II) inter-container protection, (III) protecting the host from containers, and (IV) protecting containers from a malicious or semi-honest host. [5]

The (I), (III), and (III) issue could implement the protection by the software based solutions.

For the (I) protecting a container from applications, this paper recommends that we could use the different capabilities and the LSM(Linux secure module). Take CVE-2017-5123 [14] for example. The vulnerability here is the third argument of waitpid() system call didn't ensure that the user-specified pointer points to userspace and not kernel space. Since unprivileged users shouldn't be able to write arbitrarily to kernel memory.

The solution of CVE-2017-5123 without update the Linux kernel is to insert a LSM to the kernel, which monitors the runtime behaviors of system call. If any process use the waitpid() with a pointer point to the kernel as the third argument, the LSM should block the operation and raise a signal to the user.

For the (II) inter-container protection, this paper recommends that we could use the LSM, namespaces, and cgroups to limit the container. Take CVE-2016-8655 [15] for example. This vulnerability is a bug in net/packet/af_packet.c. We often use the CAP_NET_RAW namespace in the container to make unprivileged user could use some privileged net-util commands. The bug is there exist a race condition probability to race the unauthorized data inside packet_set_ring() and packet_setsockopt() [16]. Such that, there is a chance to modify the socket version to TPACKET_V1 before the packet_set_ring function. However, which would be kfree the timer in the TPACKET_V1. We can take the timer, which is used after free, to control the SLAB adopter to write the st_uid by itself [17].

For the (III) protecting the host from containers issue. Take the Dirty CoW vulnerability for example, which is exploitation from the Linux kernel, the vulnerability could change the victim container to a privileged container. Therefore, we should protect the host from the container, which belongs to type (III) threat in this paper.

3.3 High-performance Server

This section will study some I/O performance and caching issues. Because the medical data exchange system demands the stringent specification of the response time. The I/O is the most often causing the bottleneck in the low latency required system. To support a high-performance system, we can design a module to control the throughput intelligently, and use the cache-friendly architecture to minimize the latency.

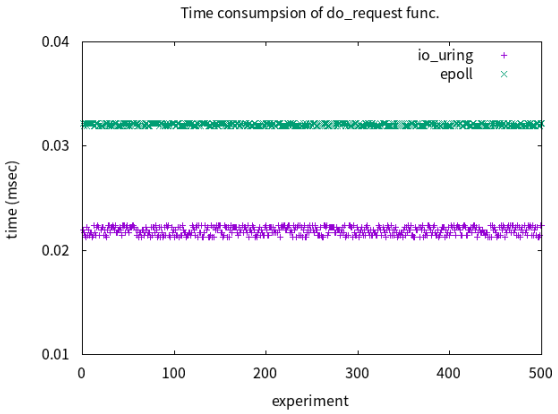
3.3.1 PINE: Optimizing Performance Isolation in Container Environments

This paper [18] introduce a high throughput and low latency module to control the I/O streams. Which implement a module to accord calculated optimization parameters, and check if the process throughput is satisfied or the 99.9% throughput is satisfied.

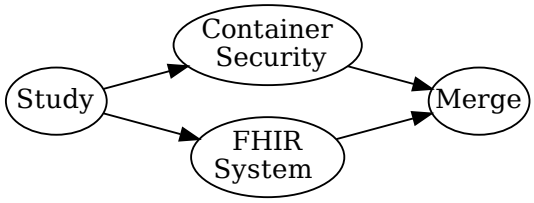
That is if the throughput reached the bottleneck, then the model would extend the bandwidth by the cgroup. The latency evaluation is more difficult than throughput evaluation. This paper [18] used the statistical data to calculate the 99.9% of latency is in the margin of error in 3 standard deviations, if not the module will raise the priority of the I/O queue.

3.3.2 The epoll vs. io_uring performance comparison

The healthcare data exchange system would request for "small data" frequently. Hence, the low latency has the priority over the high throughput in this project. We often use the system call: "epoll" for asynchronous I/O with many files descriptors, rather than sequential accessing those file descriptors in this scenario. The figure 3a [19] shows the io_uring could do request in fewer time significantly.



(a) The comparison of do_request function.



(b) The mapReduce model in this project

Figure 3

4 Methods

This project would use the MapReduce model. As shown in Figure 3b.

4.1 Study

4.1.1 Study CVEs and related mechanisms

The Linux kernel is a monolithic kernel, which is over 28 million lines of code now(2020). There are many mechanisms to solve real-world situations. Study those CVEs' related mechanism in the kernel, might have more chance to find new vulnerabilities.

This project will study several container vulnerabilities for example: CVE-2016-8655 [15], CVE-2016-9962 [20], and CVE-2020-14386 [21].

And study some kernel exploit techniques [22] because the container shares the kernel. If I could exploit the kernel in the suffering container, it might have more chance to influence the other containers or host.

4.1.2 Study FHIR and related standards

This project will implement the FHIR [1] data exchange system to demonstrate the container security risk. Hence, we should study FHIR standard, JSON format(RFC7159), XML format(RFC4825), and RESTful APIs.

4.1.3 Efficient I/O with io_uring

[23] This is a new asynchronous I/O API in Linux kernel 5.1. POSIX has aio_read() and aio_write() to satisfy asynchronous I/O, however, the implementation of those is most often lackluster and performance is poor. We will study and implement the new asynchronous I/O API: io_uring in this project to optimize the performance of the FHIR data exchange system.

4.2 Container Security

4.2.1 Implement a simple container

The Linux kernel supply some system calls to clone a process(also in thread) in their namespace and group. We could implement a simple container by ourselves so that we can make a list of vulnerabilities that may happen.

```
âđIJ  container git:(main) âIJŮ sudo ./c bash
Success on creating container
Start container: bash with clone id: 375696
In container PID: 1
bash-5.0# ./test.sh
This is the self test script in container!
Support bash cat echo ls rm hostname tree, 7 commands.
/bin/bash
./test.sh
-------FILE: ./test.sh --------
     1  #!/bin/bash
     2
     3  echo "This is the self test script in container!"
     4  echo "Support bash cat echo ls rm hostname tree, 7 commands."
     5
     6  echo "$SHELL"
     7  echo $0
     8
     9  echo "-------FILE: $0 --------"
    10  cat -n $0
    11  echo "----------------------"
    12
    13  echo $(hostname) > åďľçńžéijăèżŁèżŁ
    14  cat åďľçńžéijăèżŁèżŁ
    15  rm åďľçńžéijăèżŁèżŁ
    16  ls
----------------------
container
bin  dev  etc  home  lib  lib64  mnt  opt  proc  root  run  sbin  sys  test.
   sh  tmp  usr  var
bash-5.0# exit
exit
âđIJ  container git:(main) âIJŮ
```

src/lc_out.txt

4.2.2 List secure details of the simple container

In my rough opinion, there are 5 types of container security risks, (I) Host OS risks, (II) Orchestration system risks, (III) Container runtime risks, (IV) Registry risks, (V) Images risks. In this stage, we should research the details of those risks, and purpose some solutions.

I. Host OS risks
   - Improper user permission
   - Kernel vulnerabilities

II. Orchestration system risks
   - Unbounded domain access
   - Weak credentials
   - Mismanaged inter-container network traffic
   - Mixed of workload sensitivity levels

III. Container runtime risks
   - Runtime software vulnerabilities
   - Unbounded network access from containers
   - Insecure container runtime configurations

IV. Registry risks
   - Insecure connections to registries

- Old images in registries

V. Images risks
- Image vulnerabilities
- Embedded malware or secrets

4.2.3 Aim a vulnerability and implement the PoC

After listing the risks. This project would find a vulnerability of privilege escalation in the container and affect other containers.

4.2.4 Implement the patch and pull request

Being a security researcher, we cannot just only exploit the software, but also give patches to the maintainer. Make the container technique more secure.

4.3 FHIR system

4.3.1 Front-end

This project would be designed a user-friendly interface. Make it easy to get data for the patient and the patient's family. And make the exchange of patient's data between different medical center confidential, integral, and available.

The interface would be designed as a website, which makes every user could access on different platforms.

4.3.2 Back-end

This project would use the container technique at the back-end. Would isolate different services in a different container. We would also design an access controller for variadic requests, and design a high-performance kernel module to speed up the I/O and caching.

Access controller The access controller would be implemented as a kernel module. Because malicious user has less probability to break the Linux kernel if this module has no bugs. The access controller would use the whitelisting method to enforce the accessibility policy. It would reserve the essential system calls in the container, and discard all unused system calls.

High performance server This project would implement a kernel module for the high performance server. Which could hook the I/O system calls from the web server. The module would replace the normal I/O calls with asynchronous I/O, which could enhance the concurrency performance to provide high throughput.

Moreover, this project would design an efficient algorithm to predict and cache the container's application data. To reduce the latency, while backend server requiring data.

4.4 Merge

The last step is combining the FHIR system and container security. This project would demonstrate an escalation of normal containers(ie. Docker) to steal patients' information from the webserver. However, our container can detect and prevent a malicious user from escaping the container of the web server.

Furthermore, the performance issue is also the key point in this project. We would provide a high-performance FHIR system for the real world requirement. Therefore, we would also improve the I/O performance at the final stage of this project.

5 Expected Outcome

This project will implement a high performance FHIR medical data exchange system to demonstrate the container escalation. We want to deploy the FHIR system in containers. The container technique cloud isolates the environments between different containers. Apart from that, it is faster to start-up, lighter in memory/storage usage at run time, and easier to deploy than virtual machines.

Therefore, this project has 2 parts: FHIR system and container security.

Container security We would implement a vulnerability PoC, patch, and demonstration for container escalation. Hope this research could make the container technique more secure, and provide the medical data exchange system more reliable.

FHIR system We would implement the medical data exchange system with the FHIR standard. The implementation considers the most two important issues: security issues and performance issues. Hope this project could be the medical data exchange system model role of implementation in Taiwan.

6 References

[1] HL7. *FHIR homepage*. URL: https://www.hl7.org/fhir/.

[2] Phil Oester. *Dirty CoW CVE-2016-5195*. URL: https://dirtycow.ninja/.

[3] Wikipedia. *Dirty CoW*. URL: https://en.wikipedia.org/wiki/Dirty_COW.

[4] James Griffiths Chandler Thornton. *Taiwan reports first local Covid-19 case in more than 250 days*. URL: https://edition.cnn.com/2020/12/22/asia/taiwan-coronavirus-intl-hnk/index.html.

[5] Tassos Dimitriou Sari Sultan Imtiaz Ahmad. "Container Security: Issues, Challenges, and the Road Ahead". In: *IEEE Access* 7.18620110 (2019).

[6] Wang. *FHIR news in ithome*. URL: https://www.ithome.com.tw/news/140579.

[7] GNU. *Manpage of user namespaces*. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html.

[8] Wikipedia. *cgroups*. URL: https://en.wikipedia.org/wiki/Cgroups.

[9] GNU. *Manpage of capabilities*. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html.

[10] Tanjila Farah Delwar Alam Moniruz Zaman. "Study of the Dirty Copy On Write, A Linux Kernel Memory Allocation Vulnerability". In: 2017. URL: https://ieeexplore.ieee.org/abstract/document/7530217.

[11] Wikipedia. *Copy-on-write*. URL: https://en.wikipedia.org/wiki/Copy-on-write.

[12] GNU. *Manpage of madvise*. URL: https://www.man7.org/linux/man-pages/man2/madvise.2.html.

[13] Babak D. Beheshti A.P. Saleel Mohamed Nazeer. "Linux kernel OS local root exploit". In: 2017. URL: https://ieeexplore.ieee.org/document/8001953.

[14] Federico Bento. *CVE-2017-5123*. URL: https://reverse.put.as/2017/11/07/exploiting-cve-2017-5123/.

[15] Inc. Red Hat. *CVE-2016-8655*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655.

[16] Philip Pettersson. *CVE-2016-8655 Linux af_packet.c race condition*. URL: https://lwn.net/Articles/708319/.

[17] Philip Pettersson. *CVE-2016-8655 PoC*. URL: https://www.exploit-db.com/exploits/40871.

[18] Congfeng Jiang Youhuizi Li Jiancheng Zhang. "PINE: Optimizing Performance Isolation in Container Environments". In: *IEEE Access* 7.18526707 (2019).

[19] shanvia. *Epoll vs. io_uring*. URL: https://hackmd.io/WEifn3YsQ0CBIWZXSCSOzw?view.

[20] MITRE Corporation. *CVE-2016-9962*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9962.

[21] Inc. Red Hat. *CVE-2020-14386*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14386.

[22] xairy. *Linux Kernel Exploitation*. URL: https://github.com/xairy/linux-kernel-exploitation.

[23] Jonathan Corbet. *Ringing in a new asynchronous I/O API*. URL: https://lwn.net/Articles/776703/.

7 Academic Advisor

- Give advises of security issues.
- Give advises of the front-end of user interface.
- Introduce the vision of medical data exchanging system.
- Organize this research to a complete structure.
- Extend to a formal paper, and publish.