

The Container Security in Healthcare Data Exchange System

Chih-Hsuan Yang
National Sun-Yet-San University, Taiwan
Bachelor's degree graduation project
Advisor: Chun-I Fan

2021-06-22

Contents

1	Abstract	4
2	Introduction and Motivation	4
3	Related Works	5
3.1	Preliminaries	6
3.1.1	FHIR	6
3.1.2	Linux Kernel Features	6
3.2	Container Security	9
3.2.1	The Dirty Copy on Write	9
3.2.2	Dirty CoW Demo Code	9
3.2.3	Container Security: Issues, Challenges, and Road Ahead	11
3.3	High-Performance Server	12
3.3.1	PINE: Optimizing Performance Isola- tion in Container Environments	12
3.3.2	‘epoll’ vs. ‘io_uring’ Performance Com- parison	13
4	Methods	13
4.1	Study	13
4.1.1	CVEs and Related Mechanisms	13
4.1.2	FHIR and Related Standards	14
4.1.3	Efficient I/O with ‘io_uring’	14
4.2	Container Security	14
4.2.1	Implementation of a Simple Container	14
4.2.2	Aimming at Vulnerability and Imple- menting PoC	14
4.2.3	Implementing Patch and Pull Request	15
4.3	FHIR System	15
4.3.1	Front-End	15
4.3.2	Back-End	15
4.4	System Combination	16
5	Expected Outcome	16
6	References	17

1 Abstract

Recently, many companies use containers to run their microservices since containers could make their hardware resources be used efficiently. And the newest healthcare data exchange standard FHIR (Fast Healthcare Interoperability Resources) [1] has been implemented in a container by IBM, Microsoft, and firebase. The deployment of FHIR in a container is a trend in the digital world [2]. However, if a 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 FHIR to simulate the real-world threat and purpose soliciting a secure solution to protect patient's privacy.

2 Introduction and 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. This is because that the container shares the kernel with the host OS and other containers and deploys by a configure file [3]. In this way, the information service based on a container platform increases the resource efficiency of the system [4]. First, we often used to run a docker container to host our services, for example: the servers and some services in the information security club at NSYSU(National Sun Yat-sen University). However, there are some threats to the container technique, such as "Dirty CoW" [5] 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 [6]. It was found by Phil Oester. We were 16, the first year we 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 researchers' 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, it will be unable to achieve the purpose of isolation. That might cause the information leakage or the kernel being hacked.

Hence, there is a big problem: "How to make sure my services isolated and secure?" We are leading the information security club at NSYSU. We 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 as an example. Our medical system is the most famous one internationally. When facing COVID-19 in Taiwan, we have not had any local COVID-19 case in more than 250 days [7]. Medical workers usually needs to renew the EHR (Electronic Health Records) system frequently. To protect the privacy of patients and producing a high-performance system to exchange the EHR, an easy-deployment, effective-execution, and secure system is required. It also is the major goal of this project.

3 Related Works

This section will focus on (I) [preliminaries](#) (II) [container security](#), and (III) [high-performance server](#).

3.1 Preliminaries

3.1.1 FHIR

FHIR is a standard for healthcare data exchange. The FHIR standard will be used in Taiwan in the near future [8]. FHIR will be used to provide PHR (Personal Healthcare Records) in Taiwan. Therefore, we choose the most popular standard "FHIR" for the target.

3.1.2 Linux Kernel Features

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

Namespaces The Linux kernel provides the namespaces to perform the job of isolation and virtualization of system resources for a collection of processes [9]. 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 [10].

The nested namespace is shown in Figure 1.

Cgroups This feature can limit, account for, and isolate the hardware resource usage of a collection of processes [11]. The container could use this feature to set the maximum/minimum usage of hardware resources, which could guarantee processes' resources using reasonability.

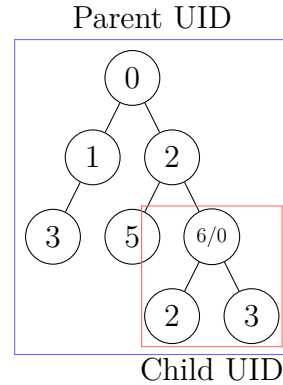


Figure 1: The nested user namespaces

Capabilities This feature divides the privileges traditionally associated with superuser into distinct units. To perform 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 processes' credentials [12].

Take the ping command as an example. The ping needs to generate and receive ICMP packets and usually is generated by "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 systems. Moreover, 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 execute the ping command as the users.

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

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

‘mmap’ This is a system call of mapping files or devices into memory, which creates a new mapping in the virtual address space of the caller process such that the process could operate the instance of a 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 its memory space.

Copy on Write This mechanism purposes that a resource is duplicated but not modified, and 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 that could inspire this mechanism in the above paragraph. When

some processes request the same memory resource, the kernel supplies the same memory page to the callers.

Race Condition This means that processes or threads are racing the same mutable resource. For example: There is an accessible and mutable shared memory which is initialized as zero. And two threads or processes are sharing that page. Consider that one of the tasks is assigning the page full of character 'A'. In the meanwhile, the scheduler context switches to the other task, which assigns that page full of 'B'. Then the scheduler 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.

Finally, there are two I/O Linux system calls that would be considered to enhance the I/O performance, which are the (VIII) `epoll` and (IX) `io_uring`.

'`epoll`' The '`epoll`' system call is a scalable I/O event notification facility. Its function is to monitor multiple file descriptors to see whether I/O is possibly on any of them, which uses RB-tree to search the monitored file descriptors.

'`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 three key elements: Submission Queue (SQ), Completion Queue (CQ), and Submission Queue Entries (SQE).

When we operate the `io_uring`, we should use the `mmap` to share the memory page from the returned file descriptor by the `io_setup`. After the page is `mmap`ed, we could share the three 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, it is useless 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 descriptors, which also reduced a system call to tell the kernel that we picked it up.

Therefore, we do not require any system call on submission and completion in optimal cases.

3.2 Container Security

3.2.1 The Dirty Copy on Write

Delwar Alam, et al. [13] showed 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 [14]. It is often inspired when 'fork' or 'mmap'.

3.2.2 Dirty CoW Demo Code

Let's analyze the proof of concept (PoC) of the dirty CoW [5] vulnerability. The key of inspiring this vulnerability is the mmaped memory space, which is mapped with the PROT_READ flag. The PROT_READ flag declares that 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/dirtycow.c

It creates two threads, which would have a race condition of the mmaped memory space, [madviseThread](#) and [proccselfmemThread](#).

threads in main

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

src/dirtycow.c

In one thread, issuing 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 [15]." 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 [15].

`madviseThread`

```

33 void *madviseThread(void *arg)
34 {
35     char *str;
36     str=(char*)arg;
37     int i,c=0;
38     for(i=0;i<1000000000;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 to read its memory by itself.

Then, we move the pointer of file descriptor of the memory resource file to the mmaped space. And we try to write it. But the mmaped space is read-only. We expected that the kernel would create a copy of this space and write the copy [16].

`proccselfmemThread`

```

50 void *proccselfmemThread(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<1000000000;i++) {
57         lseek(f,(uintptr_t) map,SEEK_SET);
58         c+=write(f,str,strlen(str));
59     }

```

```

60 |     printf("procmem %d\n\n", c);
61 | }

```

src/dirtycow.c

However, there is a problem! There is another thread that is racing this page with root privilege. If the scheduler context switches the `adviseThread` to `procmemThread` while the `adviseThread` is calling the "madvise" system call, it would cause the `procmemThread` to gain the root privilege from `adviseThread` to control the mmaped file.

3.2.3 Container Security: Issues, Challenges, and Road Ahead

Sari Sultan et al. [9] have derived four 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 [9].

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

For the (I) issue, Sari Sultan et al. [9] recommends that we could use the different capabilities and the LSM (Linux secure module). Take CVE-2017-5123 [17] as an example. The vulnerability here is the third argument of 'waitpid' system call which didn't ensure that the user-specified pointer points to user space and not kernel space since unprivileged users shouldn't be able to write arbitrarily to kernel memory.

The solution of CVE-2017-5123 without updating the Linux kernel is to insert an LSM to the kernel, which monitors the runtime behaviors of system calls. If any process uses 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) issue [9] recommends that we could use the LSM, namespaces, and cgroups to limit the container. Take CVE-2016-8655 [18, 19] as an 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 be able

to use some privileged net-util commands. The bug is that there exists a race condition probability to race the unauthorized data inside `packet_set_ring()` and `packet_setsockopt()` [20] such that there is a chance to modify the socket version to `TPACKET_V1` before the `packet_set_ring` function. However, it 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 [21].

For the (III) issue, we take the Dirty CoW vulnerability as an example, which is exploitation from the Linux kernel. The vulnerability could change the victim container to be a privileged container. Therefore, we should protect the host from the container, which belongs to type (III) threat.

3.3 High-Performance Server

This section will study some I/O performance and caching issues because the healthcare data exchange system demands the stringent specification of the response time. The I/O is the most often one 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.

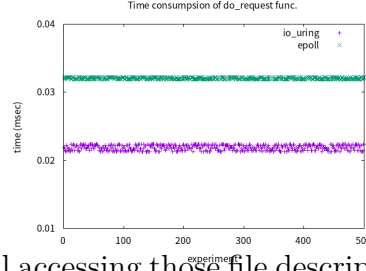
3.3.1 PINE: Optimizing Performance Isolation in Container Environments

Youhuizi Li et al. [22] introduced a high throughput and low latency module to control the I/O streams. It implements a module to accord calculated optimization parameters, and checks if the process throughput is satisfied or the 99.9% throughput is satisfied.

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. Youhuizi Li et al. [22] evaluated if the latencies of 99.9% of the data are all within three standard deviations of the acceptable latency. If not, the module will raise the priority of the I/O queue.

3.3.2 ‘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 file descriptors, rather than sequential accessing those file descriptors in this scenario. Figure 2 [28] shows that the io_uring could do request in fewer time significantly.



4 Methods

This project would use the MapReduce model which is shown in Figure 3.

4.1 Study

4.1.1 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. Studying those CVEs' related mechanism in the kernel might have more chance to find new vulnerabilities.

This project will study several container vulnerabilities such as CVE-2016-8655 [18], CVE-2016-9962 [24], and CVE-2020-14386 [25].

And we will study some kernel exploit techniques [26] because the container shares the kernel. If we could exploit the kernel in the suffering container, it might have more chance to influence the other containers or host.

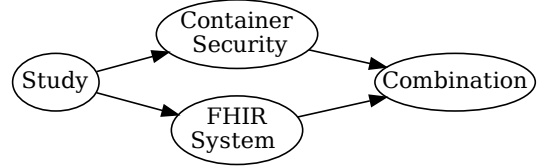


Figure 3: The mapReduce model in this project

4.1.2 FHIR and Related Standards

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

4.1.3 Efficient I/O with ‘io_uring’

This is a new asynchronous I/O API in Linux kernel 5.1 [27]. 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.

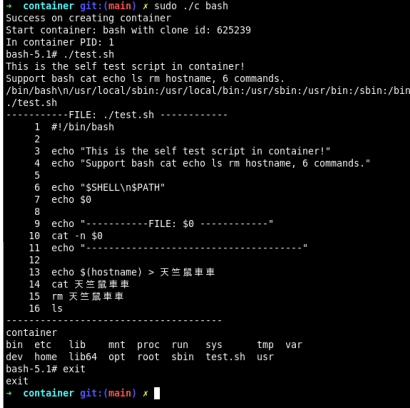
4.2 Container Security

4.2.1 Implementation of a Simple Container

The Linux kernel supplies 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, which is shown in Figure 4.

4.2.2 Aiming at Vulnerability and Implementing PoC

This project would find a vulnerability of privilege escalation in the container and affect other containers. We would use the fuzzing technique, which is listed from the vulnerabilities that may happen, to continue researching the security of the Linux kernel in order to detect other known vulnerabilities of



```
container git:(main) ✗ sudo ./c bash
Success on creating container
Start container: bash with clone id: 625239
In container PID: 1
bash-5.1# ./test.sh
This is the self test script in container!
Support bash cat echo ls rm hostname, 6 commands.
/bin/bash\nusr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
./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, 6 commands."
5
6 echo "$SHELL\n$PATH"
7 echo $0
8
9 echo "-----FILE: $0 -----"
10 cat -n $0
11 echo "-----"
12
13 echo $(hostname) > 天竺鼠草草
14 cat 天竺鼠草草
15 rm 天竺鼠草草
16 ls
-----
container
bin etc lib mnt proc run sys tmp var
dev home lib64 opt root sbin test.sh usr
bash-5.1# exit
exit
container git:(main) ✗
```

Figure 4: The implementation of a simple container

various types, as well as zero-day vulnerabilities [28].

4.2.3 Implementing Patch and Pull Request

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

4.3 FHIR System

4.3.1 Front-End

This project would design a user-friendly interface. We will make it easy to get data for the patient and the patient's family. And the exchange of patient's data between different medical center will be 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. It 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 a 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 [29]. 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 `io_uring`, which could enhance the concurrency performance to provide low latency and enhance the 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 System Combination

The last step is combining the FHIR system and container security. This project would demonstrate an escalation of normal containers (i.e., Docker) to steal patients' information from the web server. However, our containers 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 deploy a secure and high-performance FHIR healthcare data exchange system in containers. We will demonstrate a container escalation and purpose an efficient mechanism to protect the patients' privacy. Many companies publish their FHIR system in containers, that make much medical organization easy to deploy. However, the medical organization might not have a professional security software engineer to maintain the performance issues and patients' privacy issues.

Therefore, this project would aim at a secure and high-performance solution for the medical organization. We hope this could help them face the future cyber challenges.

This project has two parts: the FHIR system and container security.

Container Security We would implement a vulnerability PoC, patch, and demonstration for container escalation since the container technique is the trend of these microservices. The healthcare data exchanging center must use containers for quick deployment and management. Additionally, the information security issue becomes more and more important these days. To prevent the cracker from breaking the healthcare data center and keep the patients' information is critical today. We hope that this research could make the container technique more secure, and provide the healthcare data exchange system more reliable.

FHIR System We would implement the healthcare data exchange system with the FHIR standard. The implementation would provide a high-performance service for anyone, anywhere, and anytime. The fifth-generation mobile telecommunications has been launched, and the sixth-generation mobile telecommunications is coming. More IoT devices will collect our healthcare data for medical demands, needing a low latency server to digest the data. We hope that this project could implement a high-performance FHIR standard system, which makes data exchange effectively.

Finally, this project would combine these two parts into a container of a secure and high-performance FHIR system. It is expected that this project could play the model role of the healthcare data exchange system in Taiwan.

6 References

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

- [2] A. Ahmed and G. Pierre. “Docker Container Deployment in Fog Computing Infrastructures”. In: *2018 IEEE International Conference on Edge Computing (EDGE)*. 2018, pp. 1–8. DOI: [10.1109/EDGE.2018.00008](https://doi.org/10.1109/EDGE.2018.00008).
- [3] S. Wu et al. “Container-Based Cloud Platform for Mobile Computation Offloading”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 123–132. DOI: [10.1109/IPDPS.2017.47](https://doi.org/10.1109/IPDPS.2017.47).
- [4] C. C. Spoiala et al. “Performance comparison of a WebRTC server on Docker versus virtual machine”. In: *2016 International Conference on Development and Application Systems (DAS)*. 2016, pp. 295–298. DOI: [10.1109/DAAS.2016.7492590](https://doi.org/10.1109/DAAS.2016.7492590).
- [5] Phil Oester. *Dirty CoW CVE-2016-5195*. URL: <https://dirtycow.ninja/>.
- [6] Wikipedia. *Dirty CoW*. URL: https://en.wikipedia.org/wiki/Dirty_COW.
- [7] 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>.
- [8] Wang. *FHIR news in ithome*. URL: <https://www.ithome.com.tw/news/140579>.
- [9] Tassos Dimitriou Sari Sultan Imtiaz Ahmad. “Container Security: Issues, Challenges, and the Road Ahead”. In: *IEEE Access* 7.18620110 (2019).
- [10] GNU. *Manpage of user namespaces*. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html.
- [11] Wikipedia. *cgroups*. URL: <https://en.wikipedia.org/wiki/Cgroups>.
- [12] GNU. *Manpage of capabilities*. URL: <https://man7.org/linux/man-pages/man7/capabilities.7.html>.

- [13] 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>.
- [14] Wikipedia. *Copy-on-write*. URL: <https://en.wikipedia.org/wiki/Copy-on-write>.
- [15] GNU. *Manpage of madvise*. URL: <https://www.man7.org/linux/man-pages/man2/madvise.2.html>.
- [16] Babak D. Beheshti A.P. Saleel Mohamed Nazeer. “Linux kernel OS local root exploit”. In: 2017. URL: <https://ieeexplore.ieee.org/document/8001953>.
- [17] Federico Bento. *CVE-2017-5123*. URL: <https://reverse.put.as/2017/11/07/exploiting-cve-2017-5123/>.
- [18] Inc. Red Hat. *CVE-2016-8655*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655>.
- [19] Xin Lin et al. “A Measurement Study on Linux Container Security: Attacks and Countermeasures”. In: AC-SAC ’18. San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 418–429. ISBN: 9781450365697. DOI: [10.1145/3274694.3274720](https://doi.org/10.1145/3274694.3274720). URL: <https://doi.org/10.1145/3274694.3274720>.
- [20] Philip Pettersson. *CVE-2016-8655 Linux af_packet.c race condition*. URL: <https://lwn.net/Articles/708319/>.
- [21] Philip Pettersson. *CVE-2016-8655 PoC*. URL: <https://www.exploit-db.com/exploits/40871>.
- [22] Congfeng Jiang Youhuizi Li Jiancheng Zhang. “PINE: Optimizing Performance Isolation in Container Environments”. In: *IEEE Access* 7.18526707 (2019).
- [23] shanvia. *Epoll vs. io_uring*. URL: <https://hackmd.io/WEIfn3YsQ0CBIWZXSCSOzw?view>.
- [24] MITRE Corporation. *CVE-2016-9962*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9962>.
- [25] Inc. Red Hat. *CVE-2020-14386*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14386>.

- [26] xairy. *Linux Kernel Exploitation*. URL: <https://github.com/xairy/linux-kernel-exploitation>.
- [27] Jonathan Corbet. *Ringling in a new asynchronous I/O API*. URL: <https://lwn.net/Articles/776703/>.
- [28] E.V. Sharlaev P.A. Teplyuk A.G. Yakunin. “Study of Security Flaws in the Linux Kernel by Fuzzing”. In: 2020. URL: <https://ieeexplore.ieee.org/document/9271516>.
- [29] Sung-Taek Lee Sung-Hwa Han Hoo-Ki Lee. “Container Image Access Control Architecture to Protect Applications”. In: *IEEE Access* 8.19980335 (2020).

7 Academic Advisor

- Give advices of security issues.
- Give advices of the front-end of user interface.
- Introduce the vision of healthcare data exchanging system.
- Organize this research to a complete structure.
- Extend the result to a formal paper, and submit it to a conference or a journal.