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

March 15, 2022

Contents

Abstract

This research proposes a mechanism, forces the system to call a specific policy in the container, is deployed in runtime. This policy is designed for the FHIR healthcare data exchange standard's container, which could guarantee the FHIR server to have only supported behavior and to takes almost zero overhead. Recently, many companies use containers to run their microservices since containers could make more efficient use of their hardware resources as well as 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 [1]. Containers are isolated processes [2] instead of sandboxes [2]. Therefore, if hackers or malicious software could sneak into the container, that would be a new cyber attacking surface in nearly future.

[1]FHIR official: https://www.hl7.org/fhir/

[2]gVisor GitHub: `https://github.com/google/gvisor`

Chapter 1

Introduction

1.1 Container and Linux Kernel

The container is a secondary product of the operating system in the past 20 years. The FreeBSD develops ‘Jails’ in 1999, and the Solaris develops ‘Zones’ in 2004. Linux also took this idea into the Linux kernel, which is named cgroups (2007), the capabilities (2003), and seccomp (2005). However, why the Linux breaks this technology into many parts? This is because they had discussed: ”Why Should a System Administrator Upgrade?” in 2001 [1]. The Linux kernel almost entered the development path of ”upgrade for demand” like Microsoft Windows, and deviated from the original path of ”providing a mechanism but not a strategy” of the original Linux kernel.

While Linux were spreading in various server or distributed system, the Linux community got more pull requests to solved the scalability and virtualization issues [3]. However, they avoided confusion caused by multiple meanings of the term ”container” in the Linux kernel context. In kernel version 2.6.24 (2007) [2], control groups functionality was merged into the mainline, which is designed for an administrator (or administrative daemon) to organize processes into hierarchies of containers; each hierarchy is managed by a subsystem. Moreover, the cgroups was rewrote into cgroups-v2 in Linux kernel 4.5 (2015) [3].

The first and most complete implementation of the Linux container manager was LXC (Linux Containers). It was implemented in 2008 using cgroups and namespaces, and it runs on a single Linux kernel without requiring any patches. LXC provides a new view and imagination of virtualized services without any hypervisor. In 2016, Docker replaced LXC with ”libcontainer”, which was

[1] Version 2.4 of the LINUX KERNEL–Why Should a System Administrator Upgrade? https://www.informit.com/articles/article.aspx?p=20667

[2] Notes from a container: https://lwn.net/Articles/256389/

[3] Control Group v2: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

written in the Go programming language. Docker combined features in a new, more attractive way and made Linux containers popular.

The secondary product of the operating system, containers, offering many advantages: they enable you to "build once, run anywhere." Docker does this by bundling applications with all their dependencies into one package and isolating applications from the rest of the machine on which they're running. Therefore, this research is based on docker container to propose a scheme of healthcare data exchange system's security.

1.2 FHIR

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

1.2.1 RESTful API and Data Structure

REST (Representational State Transfer) is a stateless reliable web API, which is based on HTTP methods to access resources or data via URL parameters and the use of JSON or XML format to transmit queries. Because the RESTful is stateless, the client should keep their information (i.e. cookies) by themself.

FHIR has features: RESTful and data structure, make our research and benchmarks more accurate and reliable. Statelessness is a developer-friendly feature, the developer and the tester would not to design a complex state machine on the server-side or generating test files. And the FHIR takes RESTful as standard. Moreover, FHIR standard declared the 'StructureDefinition' [4]. These structure definitions are used to describe both the content defined in the FHIR specification itself - Resources, data types, the underlying infrastructural types, and also are used to describe how these structures are used in implementations.

1.2.2 Why IBM FHIR server

There are many applications using IBM's FHIR server as the base component of the EHR (Electronic Health Records) system to communicate with the other various databases. Take it for example that the NextCloud's EHR service, Taipei Veterans General Hospital, and AWS Cloud are using the FHIR server in a container for subroutine service. NextCloud is an open-source and self-hosted productivity

[4] FHIR Resource Structure Definition: http://www.hl7.org/fhir/structuredefinition.html

platform for users. Many people caring about their privacy issues distrust the FAAMG (Facebook, Amazon, Apple, Microsoft, Google), so they are using NextCloud to keep their privacy on their own. Therefore, they are eager to have a secure EHR system for their PHR [5].

The benefits of providing IBM FHIR container security in our study are providing secure protection testing, methods, and performance evaluation for FHIR services provided by well-known international company (IBM). This research will provide an important reference for commercial projects for the health information exchange system practiced by medical institutions in Taiwan.

[5] Richard Stallman talks about IoT

Chapter 2

Related Work

2.1 Collecting System Calls

There are several pieces of research to detect intrusions or unexpected behaviors by collecting the system calls methods in runtime [4, 5, 6, 7]. Abed, Clancy, and Levy [4] proposed a real-time host-based intrusion detection system in a container, which is based on system call monitoring. They use the 'strace' command to collect a behavior log to a system call parser. Then use the BoSC (Bag of System Calls) [8] to classify is it a normal behavior in the database.

The BoSC technique is a frequency-based detection tip. Kang, Fuller, and Honavar [8] defined those distinct system calls in $\{c_1, c_2, \ldots, c_n\}$, For all system call s_i had been called in c_i times. And they use Naïve Bayes classification to deduce if it is unexpected behavior. Then the Abed, Clancy, and Levy give the false positive rate around 2% in $O(S + n_k)$ epochs to the MySQL database [4].

- Epoch Size (S): The total number of system calls in one epoch.
- n_k: It is the size of the database after epoch k.

However, the BoSC is running in user space, even though it is a background service running on the same host kernel. It might have heavy constant time costs of copying data from user to kernel and kernel to user by the 'copy_to_user()' and 'copy_from_user()' calls.

Azab et al. [6, 7] takes a mathematical model to simulate the smart moving target defense for Linux container resiliency. Considering an 'ESCAPE' model is the interaction between attackers and their target containers as a "predator searching for a prey" search game. This search game has 3 modules: behavior monitoring, the checkpoint/restore, and the live migration modules. This model is running on the same host and the same attacking surface because they considered the containers (prey) are running on the same machine with some migration probability.

They show the survival rate in Abed, Clancy, and Levy [4] model for some zero-day vulnerabilities in different types and numbers machines. Azab et al. [6, 7] concluded that an IDS could detect and avoid mobile continually-growing attacks efficiently by the 'ESCAPE' model with collecting system calls.

2.2 Fine-grained Permission Control

The file system access control lists (ACL) was defined in POSIX, which shares a naive and robust permission model [9, 10]. But after 20 years of evolution, in the practical consideration of the Linux operating system design, it can be divided into two permission control mechanisms: (i) POSIX ACL and (ii) seccomp. Traditional permission control is mostly controlled by ACL or similar. Many Linux secure modules (LSM) also use ACLs for file access control[11]. For example, SELinux and AppArmor use such permission settings [12, 13, 14, 15].

Han et al. [12] had proposed an architecture to enforce the access control of image's layers. Because the docker engine does not guarantee the layers could not be modified by the host environment. Therefore, if we give a container privileged permission, it could modify the layers of images. The research [12] is using the LSM's policy table to enforce the access control of the file system in the kernel.

Sun et al. [13] proposed separate the security namespace. Each container can route their operation to different security namespaces for their "comment". Each involved in the security namespace independently makes a security decision, and the operation is allowed only if the policy engine allow.

However, the policy engine has four types of policy conflicts: (I) Parent-Child Conflict, (II) Global-Local Conflict, (III) Lack of Authority, and (IV) Environment does not meet the expectation. The initial security namespace Φ is $\emptyset$. (I, II) will route the policy to $\Phi = \Sigma(\Phi \cap P_i), i \in \mathbb{N}, i < n$. And the (III, IV) is conflicted by the capabilities of that process. Sun et al. [13] give the capabilities higher hierarchy then policy in the policy engine. Therefore all of these conflicts will follow the capability first.

Android sandbox also uses ACL to control SELinux permissions for application registered users. This is called in Android system UID-based discretionary access control (DAC). And after Android 5.0, SELinux is provided to force the execution of DAC [1].

[1] https://source.android.com/security/app-sandbox

2.3 Recently Exploited Vulnerabilities

In this section, we will mention and review some 'High' or 'Critical' vulnerabilities about kernel and containers in CVSS (Common Vulnerability Scoring System). Because container is not a real virtual machine, it is an isolated process.

We ignore the CVE-2020-29389 series (CVE 306). Because those CVEs are not container or kernel's vulnerabilities, those CVEs are issue of image defaults password. Despite those CVEs got 10.0 score, those are small and unimportant vulnerabilities.

2.3.1 Five Stages of Malware

We had been inspired by the quark engine[2], which is an open-source malware scoring system for Android APK files. The quark engine had been developed from the Taiwan Criminal Law's five stages: (i) Determination, (ii) Conspiracy, (iii) Preparation, (iv) Start, (v) Practice.

We also can use these five steps and category to give the malware stage to exploit the vulnerabilities. (i) Base image landing, (ii) Derived image landing, (iii) User landing, (iv) Kernel landing, (v) Escaping. The escaping category is the worst case of container security, because we want a container be a container, it must has zero leakage of capsulation.

Base image landing

This is the most fundamentally basic assumption or guarantee of container security. **inproceedings** proposed the BoSC technique must be $S = \{\emptyset\}$ in this step. By definition, for all container c is an image I in execution, that is $c = E(I)$. E is a function to execute and give container c a description δ and a lifetime status λ. If we are using the docker environment, we can use the command:

```
docker inspect [NAME|ID...]
```

to get the description δ of the container. And we can use

```
docker ps [OPTIONS]
```

to get the lifetime status λ of the container.

$$c = E(I) = \{\delta, \lambda\}$$

[2] https://quark-engine.readthedocs.io/en/latest/

$\lambda \in \{\text{created, running, paused, stopped}\}$ statuses.

<TODO: Graph will be inserted here.>

It is called base image landed, if the BoSC technique $S \neq \{\emptyset\}$, which might be injected some malicious item in the image. It is showed bellow.

<TODO: Graph will be inserted here.>

Derived image landing

It is called derive image landing if some malicious items are inserted into the final layer, while developers are inserting the application(s) and some dependencies into image layers, It could be performed by malicious base, dependencies, libraries, or binaries are inserted into the filesystem. It is often in third-party unknown source image which is integrated and republish by some crackers.

Those unknown source image could be replaced the normal or official image by some hacks or overlays. It looks fine when user didn't check the image until user create the instance of image, that is container. If the default application trigger malicious part, it would give crackers a chance to take control of the container. It would go to the next step user landing.

User landing

It is the cracker land into the container, no matter it is come from derived image or hacking from the normal micro-application. Crackers might get a shell or execute some malicious binaries by some injections or the other vulnerabilities.

In this step, the cracker could control the normal service to do the unexpected behaviors as normal hacking scenarios. They can drop databases [16] , practice the local file inclusion [17, 18] etc. Take an online judge in container as example: People cloud write some program, compile, and execute on that machine. The cracker could wrote some malicious program or load some shell code in those program, and give the operating system to execute. This is the user landing step.

If crackers could practice a remote code execution (RCE), they might get a sell and promote the privilege to the super-user account in the container. Them can do the same things like the host super-account except for the capabilities in 3.2.2.

Kernel landing

It is the hacker could hack the kernel [19, 20, 21, 22]. While the kernel copy data from user and execute the user-provided malicious pattern or user exploit the kernel vulnerabilities, and let that code executed in kernel mode, that is kernel landing.

It is kernel landing that we will introduce in the following subsection 2.3.2.

Escaping

This is the most critical step of these five steps, because this is the final utility given by the container. Despite the kernel landing is almost control the whole machine, it is the last container insecure issue of breaking the containers. There are three types of escaping: (i) Cgroups, (ii) Namespaces, (iii) Capabilities.

(i) The cgroup escaping showed that Gao et al. [23] break the cgroups' limitation and affect the other container on the same host significantly, and gain some extra resource from the host. (ii) The namespace escaping shows in 2.3.2 demonstration paragraph. The last one, (iii) capability escaping can be overridden the capability after the kernel landing and modify the 'task struct' of the process in kernel.

2.3.2 Case studies

The Dirty CoW

Alam et al. [24] 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 [25]. It is often inspired when 'fork' or 'mmap'.

Mechanism Let's analyze the proof of concept (PoC) of the dirty CoW [24] vulnerability [3]. 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.

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

src/dirtyc0w.c

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

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

[3] https://github.com/dirtycow/dirtycow.github.io/blob/master/dirtyc0w.c

src/dirtyc0w.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.” 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 [4].

```c
void *madviseThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int i,c=0;
  for(i=0;i<100000000;i++)
  {
    c+=madvise(map,100,MADV_DONTNEED);
  }
  printf("madvise %d\n\n",c);
}
```

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 printer 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 [26].

```c
void *procselfmemThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int f=open("/proc/self/mem",O_RDWR);
  int i,c=0;
  for(i=0;i<100000000;i++) {
    lseek(f,(uintptr_t) map,SEEK_SET);
    c+=write(f,str,strlen(str));
  }
```

[4] https://man7.org/linux/man-pages/man2/madvise.2.html

```
  printf("procselfmem %d\n\n", c);
}
```

src/dirtyc0w.c

However, 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.

Demo

```
user@ubuntu:~$ uname -a; id
Linux ubuntu 3.16.0-23-generic #31-Ubuntu SMP Tue Oct 21 17:56:17 UTC 2014 x86_64 x86_64 x86_64 GNU/
Linux
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),112(li
bvirtd),113(lpadmin),114(sambashare)
user@ubuntu:~$ ./cowroot
DirtyCow root privilege escalation
Backing up /usr/bin/passwd to /tmp/bak
Size of binary: 51128
Racing, this may take a while..
thread stopped
/usr/bin/passwd overwritten
Popping root shell.
Don't forget to restore /tmp/bak
thread stopped
root@ubuntu:/home/user# id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),112(libvi
rtd),113(lpadmin),114(sambashare)
root@ubuntu:/home/user# _
```

CVE-2016-8655 series

We will introduce the series vulnerabilities related to CVE-2016-8655[5], which are CVE-2017-7308[6] and CVE-2020-14386[7]. These vulnerabilities are related to the bugs in net/packet/af_packet.c in the kernel. These series vulnerability is rely on the capability of CAP_NET_RAW[8], which is a capability that can "use RAW and PACKET sockets and bind to any address for transparent proxying" in Linux. And we will introduce Linux capabilities at 3.2.2.

CVE-2016-8655 and CVE-2017-7308 They are that there exists a race condition probability to race the unauthorized data inside packet_set_ring() and packet_setsockopt(). When we are using the PACKET_RX_RING option on the setsockopt(), and if the version of the packet socket is TPACKET_V3. Then we can race the init_prb_bdqc() and swap(rb-¿pg_vec, pg_vec) in packet_set_ring() with the

[5] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655
[6] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7308
[7] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14386
[8] https://linux.die.net/man/7/capabilities

spin lock rb_queue-¿lock. However, when the socket was closed and called kfree() of the struct packet_sock. It causes a use-after-free on a kernel timer object that can be exploited by various attacks on the SLAB allocator in setsockopt() [9] [10].

They are critical vulnerabilities that can impact all Linux distributions' kernel being built from 2011 to 2016. We can use these vulnerabilities to land on the kernel in containers, such that the container would be controlled by crackers.

CVE-2020-14386 It is a combination of CVE-2016-8655 and CVE-2017-7308 above. Despite people patch those vulnerabilities, there exist an arithmetic overflow, because the variable of netoff is an offset of ethernet header which is only stored in an unsigned short. Crackers can produce an arithmetic overflow when they have the CAP_NET_RAW capability, which value must be smaller than INT_MAX, but receive a larger value than the size of a block and write beyond the bounds of a frame buffer[11].

Or Cohen submitted the patch[12] to fix this CVE-2020-14386, and this patch is integrated in Linux 5.8. This vulnerability is also a kernel level bug that can gain root privileges from unprivileged processes. Therefore, cracker could use this vulnerability to get the privilege to escape from containers.

People notice that it is impossible to do any protection if the kernel have vulnerabilities that container has the capability to ask kernel to execute malicious code directly. Despite they make the kernel up-to-date, there also have some probability that cracker could exploit the kernel and brake the container. Because, containers are just isolated processes, they are using the shared kernel as the host. When this bug is published, google's gVisor said "Hey, we are immunity to this vulnerability." [13] Because the gVisor implement their own network stack in their gVisor sandbox by the go language. They do not ask for these supports from kernel.

runC exploits

This sub-subsection would introduce some exploits for the runC engine. RunC is an abbreviation of "run container", which is an instance of host OS's process and the parent process of a container environment.

[9] https://github.com/torvalds/linux/blob/f6fb8f100b807378fda19e83e5ac6828b638603a/net/packet/af_packet.c#L3690

[10] https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html

[11] https://www.openwall.com/lists/oss-security/2020/09/03/3

[12] https://github.com/torvalds/linux/commit/acf69c946233259ab4d64f8869d4037a198c7f06

[13] https://cloud.google.com/blog/products/containers-kubernetes/how-gvisor-protects-google-cloud-services-from-cve-2020-14386

CVE-2019-5736

CVE-2021-30465

2.4 A Minimal Cross-platform Container in Linux

A kernel level virtualization, which is so called as a container, is constructed by two features: hardware limitation, namespace limitation. We can use the ‘mount’ with tag of cgroup system call in Linux to create an association set of parameters for hierarchy subsystems [14]. We use the ‘clone’ or ‘unshare’ to manipulate the task_struct in kernel.

We give the container all the hardware usage to our mini-container, and execute from a thread of function ’run’.

```c
static inline pid_t loader(char *argv[])
{
    return clone(run, c_stkptr + STK_SIZE,
                 CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWPID | SIGCHLD, argv);
}
```

src/lc/cont.c

We use the clone [15] with CLONE_NEWNS flag to start in a new mount namespace, initialing with a copy of the namespace of the parent. Then, we use chroot to limit the child process’s root directory to our ”rootfs”.

```c
static void isol()
{
    unshare(CLONE_FILES | CLONE_FS | CLONE_SYSVSEM | CLONE_NEWCGROUP);
    sethostname("container", 10);
#ifdef ROOTFS
    if (chroot(STRINGIZE_VALUE_OF(ROOTFS)))
        perror("chroot error");
#else
    if (chroot(STRINGIZE_VALUE_OF(rootfs)))
        perror("chroot error");
#endif
    printf("In container PID: %ld\n", (long) getpid());
}
```

[14] https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html
[15] https://man7.org/linux/man-pages/man2/clone.2.html

src/lc/cont.c

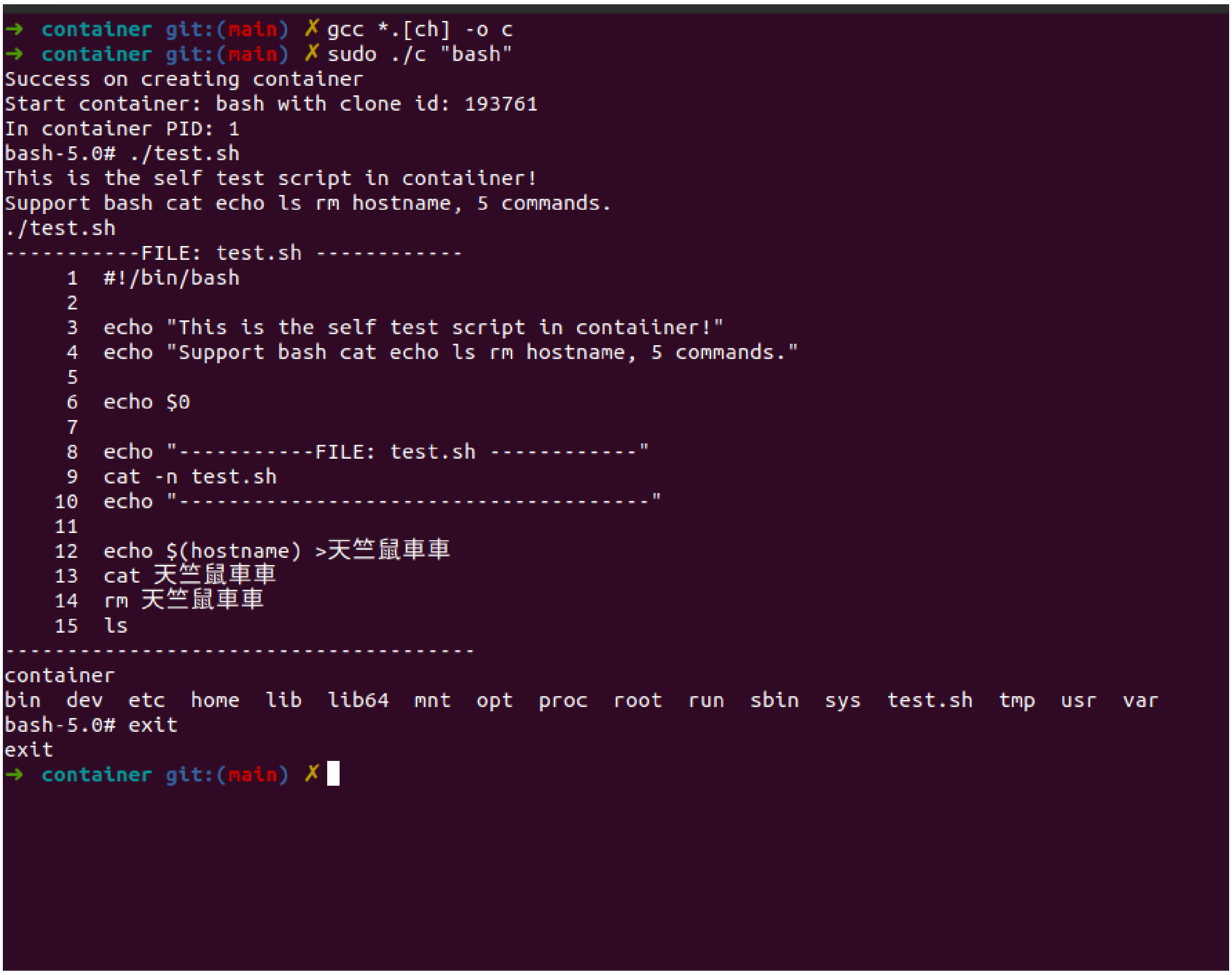
So we start the first program in the container, which would be executed in the our-designed container.

```c
static int run(void *argv)
{
    char **arg = (char **) argv;
    isol();
    chdir("/");

    int ret = execvp(arg[0], arg);
    if (ret)
        printf("%s in container\n", strerror(errno));

    return ret;
}
```

src/lc/cont.c



But there is a problem here. That is the program could not be loaded normally while the kernel try to load the dynamic libraries in to memory, which is depended by the binary program. This is the reason why we need an immutable base file system layer to support the container image.

Suppose we build the minimal container on self machine, we can assume the CPU architecture is the same. Therefore we can copy the dependencies to ”rootfs” directly.

```
#----------create root fs----------
echo "Creating rootfs"
mkdir $rootfs
for i in ${root_dirs[@]}; do
    mkdir $rootfs/$i
done
echo

#----------Copy commands----------
for app in ${support_list[@]}; do
    echo "Copying $app from $(which $app) to $rootfs/usr/bin/"
    cp $(which $app) $rootfs/bin/
done
echo

libs=()
#----------Copy lib----------
for app in ${support_list[@]}; do
    echo "Add $(which $app | xargs ldd | grep '\(\(\/usr\)\?\/lib[^\ ]\+\)' -o |
     tr '\n' ' ')for $app"
    for l in $(which $app | xargs ldd | grep '\(\(\/usr\)\?\/lib[^\ ]\+\)' -o |
    tr '\n' ' '); do
        if [[ ! " ${libs[@]} " =~ " $l " ]]; then
            libs+=("$l")
        fi
    done
done
echo
echo ${libs[@]}

for l in ${libs[@]}; do
    echo "Copying lib"
    cp -f $l "$rootfs$l"
    if [[ $? != 0 ]]; then
```

```
        mkdir -p "$rootfs$1" # the end of $1 is file name, not the dir name
        rmdir "$rootfs$1"
        cp -f $1 "$rootfs$1"
    fi
done
```

src/lc/build.sh

2.5 Virtual Environment Performance Benchmark

There is a trend of applications are developed or deployed into microservice in a virtual environment since 2008. And the performance benchmark of applications in the virtual environment becomes more and more critical.

Therefore, there are many pieces of research shows how to evaluate the performance when using containers or the other virtual infrastructures[27, 28, 29, 30]. They are comparing the throughput, latency, and QoS for memory IO, or cryptography algorithms calculating costs.

Young et al. [30] showed the gVisor costs: $2.2\times$ system call overhead, $2.5\times$ memory allocation latency, and $216\times$ **slower** than raw system on complex file opening. And Kozhirbayev and Sinnott [28] showed that I/O times have more disadvantages of latency and throughput, which is compared to container and native machines.

Chapter 3

Preliminary

3.1 Container's Components

3.1.1 Namespaces

3.1.2 Cgroups

3.1.3 Seccomp

3.2 Programs in Execution

3.2.1 The task_struct in Kernel

3.2.2 Capabilities

3.3 Sandbox Security

3.3.1 User Mode Linux

gVisor

3.3.2 Virtual Machines

3.4 The (e)BPF

Chapter 4

Proposed Scheme

It is the programmer's responsibility to write complete unit and integration tests. We extend the definition Test-Driven Development (TDD), which is not only red, green, and refactor, but also "Test Do what's Designed".

4.1 Workflow

In short, our proposal is generating a perfectly fittable mask layer which is coupled with the healthcare data exchange system in build time.

We proposed a CI/CD workflow to guarantee the runtime enforcement of policies in figure 4.1. Each block of the workflow will be described in the following subsection.

Because of the CI/CD workflow, we can rolling update all the features or fixing vulnerabilities, such that, the software would be released secure eventually. Linus Torvalds said[1] : "The only real solution to security is to admit that bugs happen, and then mitigate them by having multiple layers." And our layer is enforced in kernel space, therefore, there are no existing other attacks that can be inflected in the user program except for the kernel exploit.

4.1.1 Scan Base Image

We scan all the layers which construct the image of the container recursively. All containers are images in execution, that is we can treat the container as an image in runtime. Therefore, the layers of image construction have to be trusted.

For a general image I_i which has been constructed in n layers L_i, $\forall i \leq n$, $n \in \mathbb{N}$, we can use

[1] https://www.youtube.com/watch?v=5CIL54-KKz0

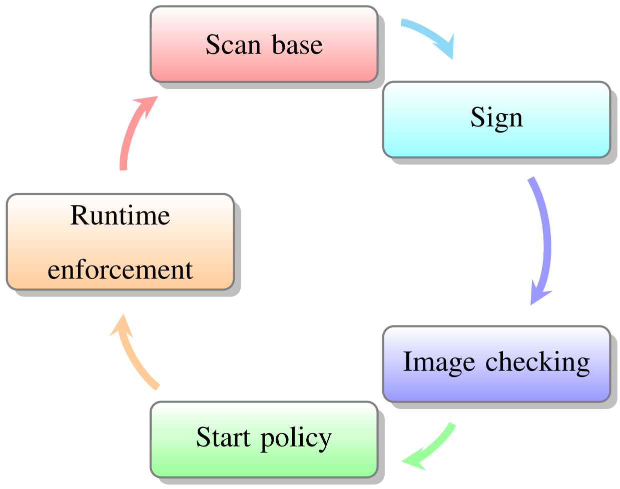


Figure 4.1: Contiguous Integration and Contiguous Deployment

the spotbugs[2] or the other bug-scanning tools to ensure that the software is a bugless program. The bugless program p_i is in the layer L_i which construct the I_i

4.1.2 Building and Signing

We will execute the developer's unit tests and the integration test in the build time. We catch all the system calls s_i by the BoSC[8] method, and generate the $S = \{s_1, s_2, \ldots s_i \ldots s_n\}$ set from the program's n system calls, $S \subseteq \mathbb{S}$, the $\mathbb{S}$ is all the system calls that the kernel supported. We wrote a driver to parse the S into a whitelist filter of seccomp's policy P.

Through the workflow above, the L_i's security is almost surely enough. Then we sign our certificate C and the policy R to the image I_i, which is constructed by those trusted layers L_i into $\hat{I}_i$. That is $\hat{I}_i = C(P \oplus \Sigma_{\forall i} L_i)$.

4.1.3 Check Image and Policy

When we deploy the $\hat{I}_i$ into an active machine, we have to check the C of $\hat{I}_i$ is valid for signer's trusted verification server.

The verification server can check the certificate C's integrity and encrypt those checking results by the server's private key P_{VK} to the active machine. The active machine will also check the certificate C' from the verification server bidirectionally.

And we register our policy P into the active machine's kernel to limit the $\hat{I}_i$ launched by the user in runtime, that is the container.

[2] https://spotbugs.github.io/

4.1.4 Enforce the Policy

The kernel of the active machine can help us to guarantee the policy P is enforced in kernel space. Since the container is launched by the user, the policy P has been invoked in each system calls of the container. Because the policy P is a whitelist, all of the other system calls which do not belong to the signed container's application would send a permission denied signal from the kernel.

4.2 Rolling Updates

The rolling update is a trend of software engineering products, which is also named agile software development. Eric S. Raymond formulated the Linus's law in *The Cathedral and the Bazaar*[31]. We give enough eyeballs and layers, all bugs or vulnerabilities are shallow in our healthcare data exchange system. Therefore the container can be secure eventually.

Chapter 5

Analysis and Benchmark

5.1 Analysis

5.1.1 Attacking Surface

5.1.2 Time Consuming

5.1.3 Statistics

Figure 5.1.3 is the FHIR server's all system calls in BoSC[8] and the number of called times.

5.2 Benchmark

5.2.1 Latency

Figure 5.2.1 is the concurrent processes transporting time difference in container and virtual machine.

5.2.2 Throughput

TBD...

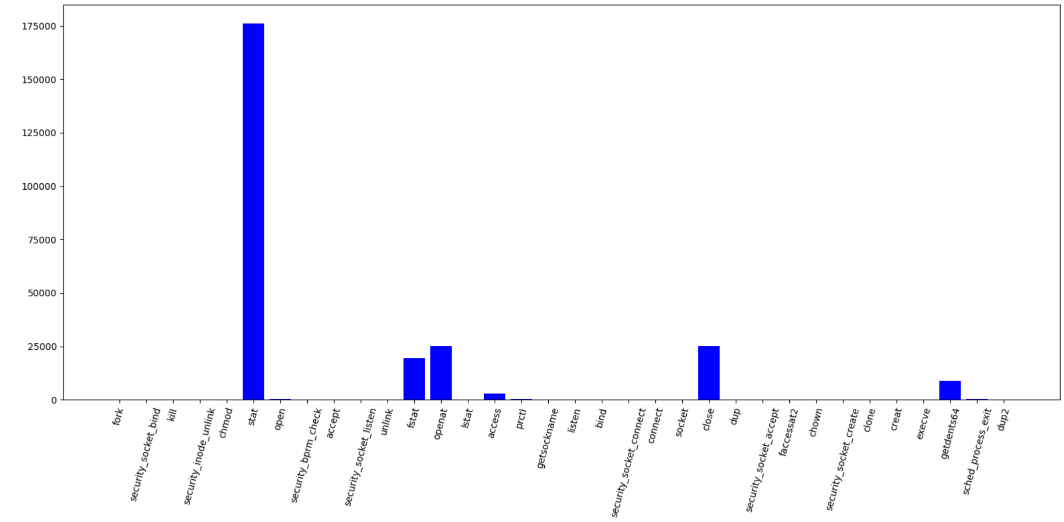


Figure 5.1: All the system calls which the FHIR called times

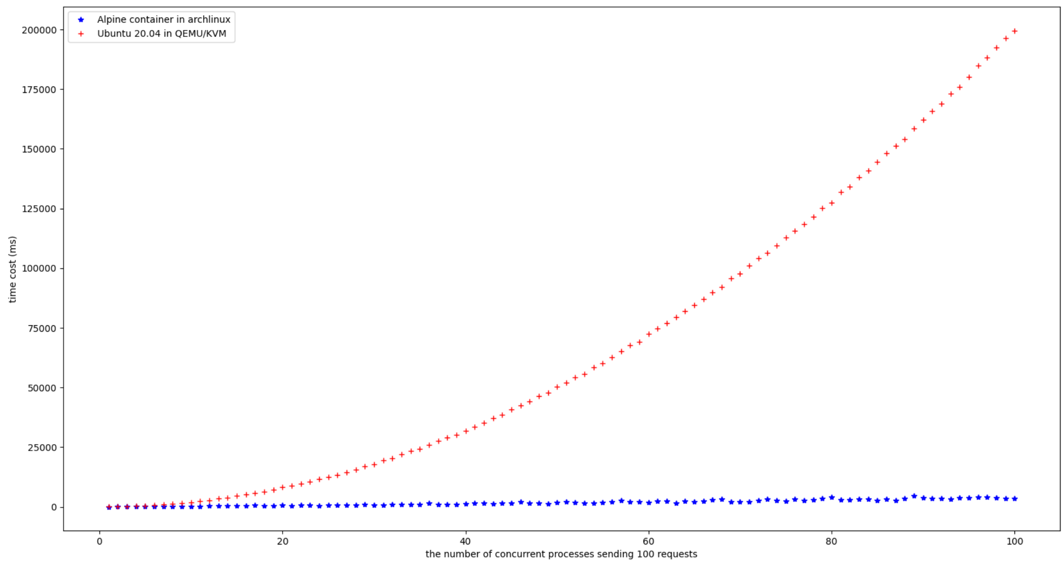


Figure 5.2: Concurrent processes transporting time

Chapter 6

Conclusion

We can see the comparison results in virtual machine and container are significantly indifferent order of time-consuming. There is no exist the gVisor's result is because the gVisor was not able to launch the IBM/FHIR server system, which is the target in our research. We also expect the gVisor might run faster significantly than the virtual machine, however, our target cannot be launched successfully in gVisor's sandbox.

We thought there might have been some race condition bugs via JWE(JAVA Web Engine) in gVisor. We found the IBM/FHIR server return an error code 141 while it launching. However, we did the same configuration in Docker with our policy and raw gVisor. Therefore, we thought the gVisor did not do well to supports all system calls.

And the time complexity of the virtual machine is significantly different from the container. We propose a hypothesis of the time complexity of the virtual machine, because there are more page fault events and limited by the throughput of virtual machine device driver[2, 29].

6.1 Better Architecture

6.2 Future Machine Learning in Kernel

Reference

[1] Arif Ahmed and Guillaume 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.

[2] Ian Goldberg et al. "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker". In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*. SSYM'96. San Jose, California: USENIX Association, 1996, p. 1.

[3] Silas Boyd-Wickizer et al. "An Analysis of Linux Scalability to Many Cores". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. URL: https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores.

[4] Amr S. Abed, Charles Clancy, and David S. Levy. "Intrusion Detection System for Applications Using Linux Containers". In: *Proceedings of the 11th International Workshop on Security and Trust Management - Volume 9331*. STM 2015. Vienna, Austria: Springer-Verlag, 2015, pp. 123–135. ISBN: 9783319248578. DOI: 10.1007/978-3-319-24858-5_8. URL: https://doi.org/10.1007/978-3-319-24858-5_8.

[5] José Flora. "Improving the Security of Microservice Systems by Detecting and Tolerating Intrusions". In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2020, pp. 131–134. DOI: 10.1109/ISSREW51248.2020.00051.

[6] Mohamed Azab et al. "Smart Moving Target Defense for Linux Container Resiliency". In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. 2016, pp. 122–130. DOI: 10.1109/CIC.2016.028.

[7] Mohamed Azab et al. "Toward Smart Moving Target Defense for Linux Container Resiliency". In: *2016 IEEE 41st Conference on Local Computer Networks (LCN)*. 2016, pp. 619–622. DOI: 10.1109/LCN.2016.106.

[8] Dae-Ki Kang, D. Fuller, and V. Honavar. "Learning classifiers for misuse and anomaly detection using a bag of system calls representation". In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*. 2005, pp. 118–125. DOI: 10.1109/IAW.2005.1495942.

[9] Andreas Grünbacher. "POSIX Access Control Lists on Linux". In: *USENIX Annual Technical Conference, FREENIX Track*. 2003.

[10] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 689–703. ISBN: 9781931971331.

[11] Stephen Dale Smalley, Chris Vance, and Wayne Salamon. "Implementing SELinux as a Linux Security Module". In: 2003.

[12] Sung-Hwa Han et al. "Container Image Access Control Architecture to Protect Applications". In: *IEEE Access* 8 (2020), pp. 162012–162021. DOI: 10.1109/ACCESS.2020.3021044.

[13] Yuqiong Sun et al. “Security Namespace: Making Linux Security Frameworks Available to Containers”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1423–1439. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/sun.

[14] Doug Kilpatrick, Wayne Salamon, and Chris Vance. “Securing The X Window System With SELinux”. In: 2003.

[15] Luis Franco, Tony Sahama, and Peter Croll. “Security Enhanced Linux to enforce Mandatory Access Control in Health Information Systems”. In: *Health Data and Knowledge Management 2008*. Ed. by P Yu P et al. Australia: Australian Computer Society, 2008, pp. 27–33. URL: https://eprints.qut.edu.au/30563/.

[16] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. “A classification of SQL-injection attacks and countermeasures”. In: *Proceedings of the IEEE international symposium on secure software engineering*. Vol. 1. IEEE. 2006, pp. 13–15.

[17] Md Maruf Hassan et al. “SAISAN: An automated Local File Inclusion vulnerability detection model”. In: *International Journal of Engineering & Technology* 7.2-3 (2018), p. 4.

[18] Michael E Whitman and Herbert J Mattord. *Principles of information security*. Cengage learning, 2011.

[19] Alessio Gaspar and Clark Godwin. “Root-kits & loadable kernel modules: exploiting the Linux kernel for fun and (educational) profit”. In: *Journal of Computing Sciences in Colleges* 22.2 (2006), pp. 244–250.

[20] Hoa Khanh Dam et al. “Automatic feature learning for predicting vulnerable software components”. In: *IEEE Transactions on Software Engineering* (2018).

[21] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. “Vulnerability prediction models: A case study on the linux kernel”. In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2016, pp. 1–10.

[22] Serguei A. Mokhov, Marc-André Laverdière, and Djamel Benredjem. “Taxonomy of Linux Kernel Vulnerability Solutions”. In: *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*. Ed. by Magued Iskander. Dordrecht: Springer Netherlands, 2008, pp. 485–493. ISBN: 978-1-4020-8739-4.

[23] Xing Gao et al. “Houdini’s Escape: Breaking the Resource Rein of Linux Control Groups”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1073–1086. ISBN: 9781450367479. DOI: 10.1145/3319535.3354227. URL: https://doi.org/10.1145/3319535.3354227.

[24] Delwar Alam et al. “Study of the Dirty Copy on Write, a Linux Kernel memory allocation vulnerability”. In: *2017 International Conference on Consumer Electronics and Devices (ICCED)*. 2017, pp. 40–45. DOI: 10.1109/ICCED.2017.8019988.

[25] Hong Lan and Xuan Wang. “Research and Design of Concurrent Web Server on Linux System”. In: *2012 International Conference on Computer Science and Service System*. 2012, pp. 734–737. DOI: 10.1109/CSSS.2012.188.

[26] A.P. Saleel, Mohamed Nazeer, and Babak D. Beheshti. “Linux kernel OS local root exploit”. In: *2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. 2017, pp. 1–5. DOI: 10.1109/LISAT.2017.8001953.

[27] Marcelo Amaral et al. “Performance Evaluation of Microservices Architectures Using Containers”. In: *2015 IEEE 14th International Symposium on Network Computing and Applications*. 2015, pp. 27–34. DOI: 10.1109/NCA.2015.49.

[28] Zhanibek Kozhirbayev and Richard O. Sinnott. "A performance comparison of container-based technologies for the Cloud". In: *Future Generation Computer Systems* 68 (2017), pp. 175–182. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.08.025. URL: https://www.sciencedirect.com/science/article/pii/S0167739X16303041.

[29] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[30] Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/young.

[31] Eric Steven Raymond. *The Cathedral and the Bazaar*. O'Reilly Media, Inc., 2002. ISBN: 9780596001087.