보안 컨테이너 가상화 기반 접근 제어

정동화 · 이성규 · 신영상 · 박현철
삼성전자 VD사업부

Access Control using Secured Container-based Virtualization

Dong-hwa Jeong · Sunggyu Lee · Youngsang Shin · Hyuncheol Park
Visual Display Division, Samsung Electronics
E-mail : {dh48.jeong, sg0923.lee, young.s.shin, hcheol.park}@samsung.com

요 약

Container 기반 가상화 환경에서는 가상 실행 환경 들이 호스트 OS를 공유함으로써 기존 가상화가 수반하는 오버헤드를 감소시키고, 가상 실행 환경 간의 isolation을 보장한다. 이로 인해 최근 embedded device와 같은 system 자원이 제한적인 환경에서도 서로 다른 가상 실행 환경 또는 호스트 실행 환경의 자원에 대한 접근을 차단할 수 있는 sandboxing의 목적으로 활발히 연구 및 적용되고 있다. 하지만, 가상 실행 환경들이 공유하는 호스트 OS 및 호스트 실행 환경에 존재하는 보안 취약점이 있을 경우 이를 악용한 공격자가 가상 실행 환경으로의 접근 및 제어를 할 수 있게 되는 보안 위협이 존재하여 이의 방지에 대한 필요성이 증가하였다.

본 논문에서는 가상 실행 환경에 대한 임의 접근 및 비인가 행위를 차단하기 위해 가상 실행 환경 접근 권한 모델을 정의하고 이를 제어하는 Container 접근 제어 기법을 제안한다. 또한, 공격자의 Container 접근 제어 기능 무력화 방지를 위해 커널 드라이버 인증 기법을 제안한다. 제안된 기법은 Linux 커널에 구현 및 테스트되었으며, 가상 실행 환경에 대한 임의 접근 및 비인가 행위 차단 결과를 보인다.

ABSTRACT

Container-based virtualization reduces performance overhead compared with other virtualization technologies and guarantees an isolation of each virtual execution environment. So, it is being studied to block access to host resources or container resources for sandboxing in restricted system resource like embedded devices. However, because security threats which are caused by security vulnerabilities of the host OS or the security issues of the host environment exist, the needs of the technology to prevent an illegal accesses and unauthorized behaviors by malware has to be increased.

In this paper, we define additional access permissions to access a virtual execution environment newly and control them in kernel space to protect attacks from illegal access and unauthorized behaviors by malware and suggest the Container Access Control to control them. Also, we suggest a way to block a loading of unauthenticated kernel driver to disable the Container Access Control running in host OS by malware. We implement and verify proposed technologies on Linux Kernel.

키워드
보안, 가상화, 컨테이너, 접근 제어, namespace

I. 서 론

가상화 기술은 단일의 하드웨어 자원을 논리적으로 독립된 다수 가상머신들이 공유할 수 있게 함으로써 하드웨어 자원의 효율적 사용 및 효과적 관리 목적으로 발전이 가속되어 왔다. 또한, 가상화 기술이 보장하는 이러한 가상머신 간의 독립성은 application 실행 환경 간의 isolation을 제공

함으로써, 보안 관점에서 악의적인 사용자의 각 가상 실행 환경 간 임의 접근 및 비인가 행위 방지 목적으로도 활발히 연구되어 왔다.[1]

기존의 하이퍼바이저 기반 반가상화 및 전가상화 기술의 경우 성능 오버헤드로 인해 embedded device와 같은 system 자원이 제한적인 환경에서는 isolation의 목적을 위한 적용이 제한적이었다. 그러나 최근 가상 실행 환경의 성능 오버헤드를 저감시킨 OS Level 가상화 기술의 발전으로 가상화 기술 기반 isolation에 대한 관심 및 연구가 다시 증가하고 있다. 특히, OS Level 가상화 기술 중 하나인 lxc[2], docker[3] 등과 같은 container 기반 가상화 기술의 경우, 가상 실행 환경 간 호스트 OS를 공유함으로써 평균적으로 5배 이상의 application 동작 성능을 개선하여[4] system 보안을 강화하기 위한 목적으로 널리 연구되고 있다.

Container 기반 가상화 환경에서는 각각의 가상 실행 환경(container)들을 격리하여 서로 다른 가상 실행 환경 또는 가상 실행 환경을 제외한 호스트 OS 위에서 동작하는 어플리케이션들의 실행 환경인 호스트 실행 환경의 자원에 접근하는 것을 차단함으로써, DAC(discretionary access control), MAC(mandatory access control)와 같은 접근 제어 기술이나 kernel hardening보다 더 뛰어난 application sandboxing 기능을 제공할 수 있다. 이로 인해, 하나의 가상 실행 환경이 외부의 공격을 받더라도 해당 실행 환경만 영향을 받고, 나머지 가상 실행 환경 및 호스트 실행 환경은 보호할 수 있어 system이 받을 수 있는 피해 범위를 최소화할 수 있다. 또한 다양한 권한 분리, 최소 권한 부여, capability 제어, 자원 제한을 적용하여 각 가상 실행 환경에 맞는 보안 권한 제어가 가능하다.[5]

하지만, container 기반 가상화 환경에서는 부하 저감을 위해 호스트 실행 환경과 가상 실행 환경들이 커널을 공유하여 사용하는데, 이때 호스트 실행 환경에서 동작하는 커널인 호스트 OS에 보안 취약점이 존재할 경우 공격자가 이를 악용하여 모든 가상 실행 환경으로의 접근 및 제어를 할 수 있게 되는 보안 위협이 존재한다. 따라서 이러한 공격자의 호스트 OS에 대한 루트 권한 탈취 등의 상황에서도 가상 실행 환경에 대한 임의 접근 및 비인가 행위를 방지할 수 있는 기술이 필요하다.

본 논문에서는 호스트 OS의 루트 권한을 임의로 탈취한 공격자의 가상 실행 환경 접근을 차단하기 위해, 가상 실행 환경 접근 권한을 추가적으로 정의하고, 이를 커널 영역에서 제어하여 불법적인 접근 및 제어를 차단하는 기능을 수행하는 container 접근 제어 기법을 제안 한다. 또한, 호스트 OS 영역에서 동작하는 container 접근 제어 기능의 임의 무력화를 방지하기 위해 kernel driver 인증 기능을 추가하여 container 접근 제어 기능을 무력화하는 공격자의 kernel driver 로딩 차단 기법을 제안한다.

그리고 이 기법들을 Linux 커널에 구현 및 테스트하여, 호스트 OS에 대한 루트 권한을 탈취한 공격자의 가상 실행 환경에 대한 임의 접근 및 비인가 행위를 효과적으로 방지할 수 있음을 보인다.

Ⅱ. 관련 연구

최근 OS Level 가상화 기술은 physical machine의 하드웨어를 가상화하지 않고, 공유된 global kernel resource들을 이용하여 가상 실행 환경을 생성하는 점에서 반가상화 또는 전가상화 기술들과는 다르다. 이로 인해, 복수의 가상 실행 환경들이 호스트 OS를 공유하고 동일한 system interface들을 사용함으로써 훨씬 더 적은 system 자원(cpu, memory, networking...)들을 사용하게 되어, 자원에 제약이 많은 mobile이나 embedded device 등에서도 사용이 가능하게 되었다.[4]

특히, OS Level 가상화 기술 중 하나인 Container 기반 가상화 기술을 이용하여 light-weight 하면서도 서로 격리된 application sandboxing 보안 기능을 제공하는 가상 실행 환경을 생성하여 system의 보안을 향상시키는 연구가 최근 활발히 이루어지고 있다.[1]

[1]에서는 isolation 된 다수의 container들이 동작하는 환경에서 하나의 container를 제어할 수 있는 권한을 가진 공격자가 다른 container에 접근하거나 이를 제어하는 등의 악의적인 행위를 하지 못하도록 system을 잘 보호할 수 있는 보안 모델을 제시하였고, 이를 OS Level 가상화 기술의 기반인 kernel namespace, cgroups, capability, seccomp 기반 동작으로 설명하였다.

그러나 [1]에서는 가상 실행 환경 간의 isolation 모델만을 제시하고 있어 호스트 OS 및 호스트 실행 환경에 존재하는 보안 취약점으로 인해 불법적으로 가상 실행 환경에 접근하거나 이를 제어할 수 있는 보안 위협에 대한 해결이 필요하다.

Ⅲ. Container 기반 가상화 개념 및 특징

OS Level의 Container 기반 가상화 기술은 Xen[6], VMWare[7], KVM[8] 등의 기존 가상화 기술들이 하드웨어 가상화를 기반으로 동작했던 것과는 다르게 global kernel resource(kernel namespace)들을 가상화하여 동작하는 방식이다.

이는 복수의 가상 실행 환경(container)들이 하나의 physical machine 상에서 동작하는 동일한 호스트 커널을 공유하여 사용할 수 있게 함으로써, 하드웨어 기반 가상화 기술 대비 더 적은 system 자원(cpu, memory, networking...) 만으로 가상화 기술 적용이 가능하다.

또한, Linux 커널에서 제공하는 kernel namespac

e를 기반으로 cgroups, capability, seccomp, apparmor, smack, selinux 등의 kernel security feature들을 이용하여 가상 실행 환경에 대한 보안을 강화하고 있다.[9]

따라서 container 기반 가상화는 system 자원이 상대적으로 제한적인 embedded device에서도 가상 실행 환경을 효율적으로 동작시키고, 해당 가상 실행 환경들의 보안을 강화시키는데 활용될 수 있게 되었다.

OS Level 가상화 기술을 보안 강화 목적으로 적용할 경우, 각각의 가상 실행 환경들을 격리하여 서로 다른 가상 실행 환경에 접근하거나 host 실행 환경에 접근하는 것을 차단하여 application sandboxing 기능을 제공함으로써, 하나의 가상 실행 환경에 보안 문제가 발생하더라도 전체 system이 영향을 받지 않아 system의 보안을 더 강화시킬 수가 있다.

또한, 가상 실행 환경마다 각자 환경에 필요한 system 자원이나 권한만을 가질 수 있도록 제한하여 권한 분리 및 최소 권한 부여 등을 통해, system에 끼칠 수 있는 보안 문제에 대한 영향을 최소화 시킬 수 있도록 제어가 가능하다.

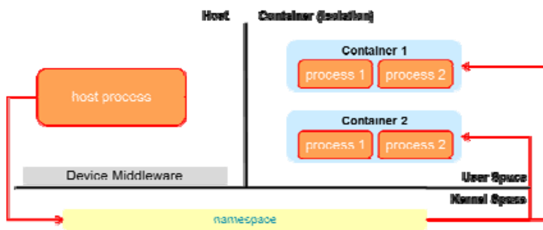

그림 1. 기존 가상 실행 환경

그러나 그림 1에서와 같이 OS Level 가상화 기술 환경에서는 복수의 가상 실행 환경들이 호스트 OS를 공유해야 하는 제약사항으로 인해 각 가상 실행 환경들이 호스트 실행 환경과 동일한 커널을 사용해야한다. 따라서 공유되는 호스트 OS에 존재하는 보안 취약점으로 인해 서로 다른 가상 실행 환경 또는 호스트 실행 환경으로의 불법적인 접근 및 제어와 같은 악의적인 행위를 할 수 있는 보안 문제점이 존재한다.

또한, 근본적으로 호스트 실행 환경에서는 system의 모든 가상 실행 환경을 제어할 수 있고, 자원에도 접근이 가능하기 때문에, 호스트 실행 환경에 존재하는 보안 취약점으로 인해 앞선 경우와 동일한 문제가 발생할 수 있다.

Ⅳ. Protection Container 설계 및 구현

본 장에서는 각 실행 환경에 공유되어 사용되는 커널 및 호스트 실행 환경의 취약점으로 인해 불법적인 가상 실행 환경으로의 접근 및 제어 문제를 해결하기 위한 Protection Container의 설계와 구현에 대해 설명한다. 그림 2에서는 Protection Container의 설계와 동작 개념을 보인다.

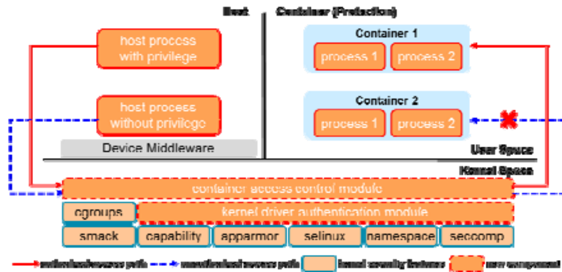

그림 2. Protection Container 구조

그림 2에서 Container Access Control Module은 불법적인 가상 실행환경으로의 접근 및 제어를 차단하기 위한 기능인데, 호스트 실행 환경에서 동작하는 process 또는 커널 드라이버가 namespace가 분리된 영역의 자원에 접근하거나, 분리된 namespace를 제어하려고 할 때, 이에 대한 권한이 존재하는지 확인하여, 권한이 없는 process 및 커널 드라이버의 접근을 차단한다.

분리된 namespace 영역에 접근할 수 있는 권한은 binary 서명 인증 방식을 이용하며, 접근 권한이 필요한 경우 해당 binary에 대한 전체 hash 값을 생성하고, 이를 private key로 암호화한 인증코드를 생성하고, 이 인증코드를 binary에 추가해둔다. 이후 해당 binary가 실행될 때, Container Access Control Module은 namespace 분리 영역으로의 접근여부를 판단하기 위해 접근을 시도한 process 및 커널 드라이버의 hash 값을 다시 생성하고, 파일에서 읽어 들인 인증코드를 public key로 복호화하여 hash 값이 동일한지 확인한다. 확인 결과 hash 값이 동일한 경우 접근 권한을 가지고 있는 process로 인증하고, 접근을 허용한다.

그림 2에서 Kernel Driver Authentication Module은 악의적인 행위를 수행하는 커널 드라이버로부터 Container Access Control Module이 무력화되는 것을 방지하기 위한 기능으로, 커널 드라이버가 로딩 될 때 인증을 통해 허용되지 않은 커널 드라이버가 동작하는 것을 막음으로써, 불법적인 가상 실행 환경으로의 접근 및 제어를 위한 공격 시도를 할 수 없도록 차단한다.

커널 드라이버의 인증은 앞서 설명한 것과 동일한 binary 서명 인증 방식을 이용하였다.

본 논문에서는 제시한 Protection Container의 유효성 검증을 위해 Linux 커널에 Container Access Control Module과 Kernel Driver Authentication Module을 구현하고 테스트하였다.

그림 3 및 그림 4에서는 lxc tool과 pid 1번 process의 namespace id 확인을 통해 container 접근 권한을 가진 shell을 이용하여 namespace가 다른

container 영역에 shell 실행이 가능한 것과 권한이 없는 shell에서 container 영역에 shell 실행이 실패하는 것을 확인하였다.

그림 3에서 권한을 가진 shell을 이용하여 container에 접근이 가능함을 확인하기 위해 container 내부의 pid 1번 process와 host의 pid 1번 process의 ipc, mnt, net, pid, uts namespace id 값들을 비교하여 id 값이 서로 다름을 확인함으로써, container로 접근이 가능함을 확인하였다.

```
sh-3.2# id
uid=0(root) gid=0(root)
sh-3.2#
sh-3.2#
sh-3.2# lxc-ls
container
sh-3.2#
sh-3.2#
sh-3.2# lxc-attach -n container
bash-3.2# id
uid=0(root) gid=0(root)
bash-3.2#
bash-3.2#
bash-3.2# ls -al /proc/1/ns
total 0
dr-x--x--x 2 root root 0 2017-08-17 18:13 .
dr-xr-xr-x 8 root root 0 2017-08-17 18:12
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 ipc -> ipc:[4026532563]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 mnt -> mnt:[4026532561]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 net -> net:[4026532566]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 pid -> pid:[4026532564]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 uts -> uts:[4026532562]
bash-3.2#
bash-3.2#
bash-3.2# exit
exit
sh-3.2#
sh-3.2#
sh-3.2# ls -al /proc/1/ns
total 0
dr-x--x--x 2 root root 0 2017-08-17 18:13 .
dr-xr-xr-x 8 root root 0 1970-01-01 09:00
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 net -> net:[4026531905]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 uts -> uts:[4026531838]
sh-3.2#
```

그림 3. 권한을 소유한 경우의 Container 접근

그림 4에서 접근 권한이 없는 shell에 대해서 동일한 방식으로 ipc, mnt, net, pid, uts namespace의 id 값들을 비교하여 id 값들이 변경되지 않고 동일함을 확인하였으며, 이를 통해 권한이 없는 shell을 이용해 container에 접근 할 경우 접근이 되지 않는 것을 확인하였다.

```
sh-3.2# id
uid=0(root) gid=0(root)
sh-3.2#
sh-3.2#
sh-3.2# ls -al /proc/1/ns
total 0
dr-x--x--x 2 root root 0 2017-08-17 18:13 .
dr-xr-xr-x 8 root root 0 1970-01-01 09:00
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 net -> net:[4026531905]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 uts -> uts:[4026531838]
sh-3.2#
sh-3.2#
sh-3.2# lxc-attach -n container
sh-3.2#
sh-3.2#
sh-3.2# ls -al /proc/1/ns
total 0
dr-x--x--x 2 root root 0 2017-08-17 18:13 .
dr-xr-xr-x 8 root root 0 1970-01-01 09:00
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 net -> net:[4026531905]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 2017-08-17 18:13 uts -> uts:[4026531838]
sh-3.2#
```

그림 4. 권한이 없는 경우 Container 접근 차단

그림 5에서는 container 접근 권한을 가진 shell에서 container의 root file system path 하위에 존재하는 자원에 접근이 가능하다는 것을 보였으며, 그림 6에서 container 접근 권한이 없는 shell에서는 동일한 자원에 접근이 차단되는 것을 확인하였다.

```
sh-3.2# id
uid=0(root) gid=0(root)
sh-3.2#
sh-3.2#
sh-3.2# cd /zone/container/rootfs
sh-3.2# ls
bin        etc   lib    mnt  proc  sbin  srv             usr
boot  dev  home  media  opt  run         sys  tmp        var
sh-3.2#
sh-3.2#
sh-3.2# ls -al /zone
total 0
drwxr-xr-x 1 root root  4 2017-07-13 18:22 .
drwxr-xr-x 1 root root 25 2017-07-13 19:09 ..
drwxr-xr-x 1 root root  2 2017-08-17 18:12 container
sh-3.2#
```

그림 5. 권한을 소유한 경우의 자원 접근

```
sh-3.2# cd /zone/container/rootfs
sh: cd: /zone/container/rootfs: Not a directory
sh-3.2#
sh-3.2#
sh-3.2# ls -al /zone
ls: cannot access /zone/container: Operation not permitted
total 0
drwxr-xr-x 1 root root  4 2017-07-13 18:22 .
drwxr-xr-x 1 root root 25 2017-07-13 19:09 ..
d????????? ? ?    ?     ?                ? container
sh-3.2#
```

그림 6. 권한이 없는 경우의 자원 접근 차단

그림 7 및 그림 8에서는 인증된 커널 드라이버가 정상적으로 로딩 되는 것과 인증되지 않은 커널 드라이버가 차단되는 것을 보였다.

```
sh-3.2# insmod /lib/modules/linux/kernel/container.ko
sh-3.2# lsmod | grep container
container              78870  0

sh-3.2#
```

그림 7. 인증된 커널 드라이버 로딩

```
sh-3.2# insmod /lib/modules/linux/kernel/attack.ko
insmod: ERROR: could not insert module /lib/modules/linux/
kernel/attack.ko: Operation not permitted
sh-3.2#
```

그림 8. 인증되지 않은 커널 드라이버의 차단

위의 테스트 결과를 통해 Container Access Control Module이 호스트 실행 환경에서 동작하는 process(root, user 권한)가 가상 실행 환경의 자원에 접근하거나 제어를 하지 못하는 것을 확인하였고, binary 서명 인증 방식을 이용해 권한을 추가해준 process만이 접근 및 제어가 가능한 것을 확인하였다.

또한, Kernel Driver Authentication Module이 인증되지 않은 커널 드라이버의 로딩을 차단하고 인증된 커널 드라이버만 로딩을 허용하는 것을 확인하였다.

V. 결 론

OS Level 가상화 기반 Application Sandboxing 기법은 서로 다른 가상 실행 환경에 대한 접근을 차단함으로써, 높은 수준의 isolation을 보장할 수 있는 장점이 있지만, 공유되어 사용되는 커널 및 호스트 실행 환경에 존재하는 보안 취약점을 악

용하여 가상 실행 환경에 대한 임의 접근이나 비인가 제어 취약점이 존재한다.

본 논문에서는 가상 실행 환경에 대한 접근 제어 기법을 제안하여 허용되지 않은 process가 가상 실행 환경에 불법적으로 접근하거나 이를 제어하는 것을 차단하고, 호스트 OS의 루트 권한을 탈취한 공격자가 가상 실행 환경에 대한 접근제어 기법을 임의로 무력화 할 수 없도록 커널 드라이버 서명 및 인증기법을 제안하였다. 또한 이를 Linux 커널에 실제 구현 및 테스트하여 가상 실행 환경에 대한 접근 제어기법과 커널 드라이버 서명 및 인증기법이 적용된 Protection Container의 효과적 동작 결과를 보였다.

향후에는 가상 실행 환경 및 호스트 실행 환경 간에 IPC namespace의 제어범위에 포함되지 않는 다양한 종류의 IPC mechanism들을 사용하여 통신하였을 때, 어떤 보안 취약점이 발생할 가능성이 있는지를 분석하고 각 IPC 통신을 안전하게 접근 제어하기 위한 연구를 더 진행하고자 한다.

참고문헌

[1] K. Bernsmed and S. Fischer-Hubner(Eds.), Security of OS-Level Virtualization Technologies, 2014.
[2] Linux Containers, https://linuxcontainers.org/
[3] docker, https://www.docker.com
[4] Ghaziabad, Performance comparison between Linux Containers and Virtual Machines, International Conference on Advances in Computers Engineering and Applications (ICACEA) IMS Engineering College, 2015.
[5] Aaron Grattafiori, Understanding and Hardening Linux Containers, NCC Group White paper, 2016.
[6] Xen Project, https://www.xenproject.org
[7] VMWare, https://www.vmware.com/kr.html
[8] KVM, https://www.linux-kvm.org/page/Main_Page
[9] Thanh Bui, Analysis of Docker Security, arXiv:1501.02967v1, Cornell University, 2015.