Contents

1 Introduction to Clear Containers **7**

1.1 Security Mechanisms in Hardware and Software 8

1.2 Clear Containers Software Architecture 9

1.3 KVM Exit 11

1.4 Virtual Machine (VM) Memory Management 12

1.5 Why Guest Operating System (OS) uses a Direct Access for Files (DAX) driver? 14

1.6 Virtio 16

1.7 A Walk Through 16

2 Threat Modeling Methodology **21**

2.1 Threat Modeling Stages 21

2.1.1 Understanding the adversary's view 21

2.1.2 Caracterizing the security of the system 22

2.1.3 Determining Threats 23

2.2 Classification of Threats 23

2.3 Scale of Rating 24

2.4 Symbology 26

2.5 Choosing what to model 27

3 Threat Modeling of Clear Containers **29**

3.1 Assets 30

3.2 Entry Points and Trust Levels 31

3.3 Characterize the Security of the System 32
3.3.1 Assumptions 33
3.3.2 Data Flow Diagrams 34
3.4 Determine Threats 37
3.5 Mitigations 45
3.5.1 User Process Crossing to Another User Process 45
3.5.2 One process crossing to kernel 45
3.5.3 Protection to Binaries 45
3.5.4 Access Control List 46

4 Results and Conclusions 47
4.1 Results 47
4.2 Conclusions 48
4.3 Future Work 48

A Interesting Source Code 53
A.1 lkvm handle `KVM_EXIT` 53

B Tools 57

C Common Vulnerability Enumarations (CVEs) of Linux Namespace 59

List of Figures

1-1 Intel CPU Rings 8
1-2 Security Mechanisms 10
1-3 Clear Container Software Architecture 11
1-4 Virtualization flow 12
1-5 Main components of virtualization 13
1-6 VM Guest memory map in host memory 14
1-7 Using hard disk as RAM 15
1-8 Shared memory between guest and host 16
1-9 Virtio drivers 17
1-10 Interaction between guest and host 20

2-1 The high-level process of threat modeling 22
2-2 Data flow symbols 26

3-1 Virtual Machine Escapes 30
3-2 Privilege Instructions Space 33
3-3 Data Flow Diagram (DFD) of Clear Containers architecture 35
3-4 DFD of lkvm 36

List of Tables

3.1 Entry points with trust level 31
3.2 DREAD rating for Clear Containers threats 44

4.1 High priority threats with Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege (STRIDE) 47

Chapter 1

Introduction to Clear Containers

Clear Containers [1] is one of the features provided by new Linux distribution called Clear Linux [2]. This distro is oriented to Cloud Computing tuned to manage virtual machines having the best of Intel Architecture technology and performance.

Because Virtual Machines (VMs) are expensive and slow to start, companies are using Linux Containers (LXC) to provide a more efficient alternative, but how much security do they provide to containers?

With performance in mind, Intel is innovating with Clear Containers to improve security of containers by using its Intel hardware Virtualization Technology (VT).

Clear Containers technology allows you to create containers that are completely isolated on your Linux host machine, with almost no performance penalty.

The name Clear Containers might confuse users to think that Clear Containers is similar to Linux Container, but in fact Clear Containers rely on a hypervisor running on a Clear Linux host and having VMs with a small Clear Linux OS, which is a optimized Clear Linux kernel that runs in the VM environment as guest. The software architecture is similar to other tools already in the market that provide full virtualization.

To understand how Intel is achieving its goal, we will provide a short introduction for Linux kernel basics and know the security mechanisms of Linux OS, Intel processors and the hardware VT. This chapter will start with a brief introduction of the security involved and a walk through the processes involved from an application (app)

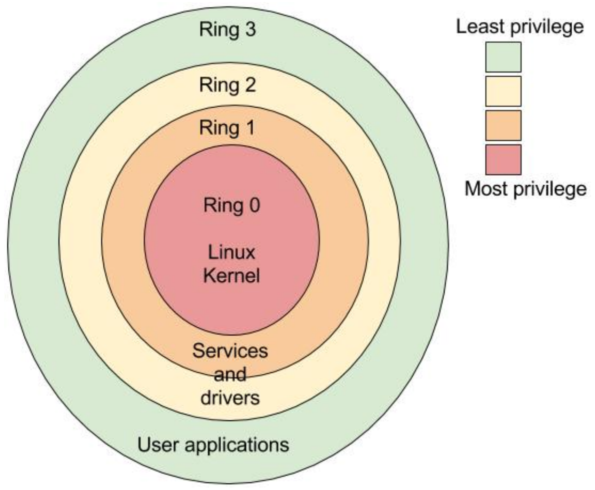

Figure 1-1: Intel CPU Rings

in a Guest OS to the hardware.

1.1 Security Mechanisms in Hardware and Software

Intel processor has four ring levels, numbered 0 (most privileged) to 3 (least privileged) as shown in figure 1-1 to provide boundaries for Linux processes and restrict them access to privilege resources and to execute operations successfully. In ring zero some machine instructions can be executed to interact with privilege hardware like memory or Input and Output (I/O) ports. An attempt to run those instructions in ring 3 causes a general-protection exception.

The Linux kernel uses this security mechanism to implement the kernel-user space isolation, and assign ring zero (the most privilege) to kernel space and allow access to memory locations, peripherals devices, system drivers, and sensitive configuration parameters providing much more dangerous accesses to critical resources, and it the most protected. Ring 1 and ring 2 are not used by the kernel and ring 3 is usually for applications (apps) to which limits the type of memory, peripheral device, and driver access activity. If an app needs more privilege, it can use system calls to request the Kernel to perform the necessary task. If code executing in user mode attempts to send instructions to the Central Processing Unit (CPU) outside of its permission, then CPU will handle this violation as an exception. Instead of your entire system crashes, only that particular app crashes [3].

Intel processors added a feature called VT to support full virtualization [4]. It means that a user should not detect any difference between running in a guest OS or in a host OS.

VT supports the Virtual Machine Extensions (VMX) operation and has two modes of operations: VMX root operation and VMX non-root operation. The Virtual Machine Monitor (VMM) or hypervisor, is a program that creates and runs VMs, it will run in VMX root operation and guest software will run in VMX non-root operation. When the processor is running in VMX non-root and wants to transition into VMX root operation, the hardware handles this transition called VM Entry and returning back to non-root is called VM Exit.

Processor in VMX root operation is similar normal processor execution. The principal difference is a new set of instructions (VMX instructions) and more control registers to handle VMs. In VMX non-root operation, the processor is restricted to some instructions, and attempt to execute privilege machine instructions or events will cause a VM Exit and enter to the VMM. This VM Exit facilitate virtualization by helping the root operation to trap these actions and implement the functionality in software used by hardware in VMX non-root operation. This limitation allows the VMM to retain control of processor and resources, so VT applies restrictions to processes on guest running in ring 0, guest (non-root).

Clear Linux uses the new VMX infrastucture, among with Linux process space to protect virtual machines and host from illegal accesses. Figure 1-2.

1.2 Clear Containers Software Architecture

Now lets put together all components in a stack layer architecture and their corresponding privilege level as shown in figure 1-3. We can see that Clear Containers is made of many software components that work together to handle VMs functionality.

The guest OS running inside a VM is a small Clear Linux that have DAX driver,

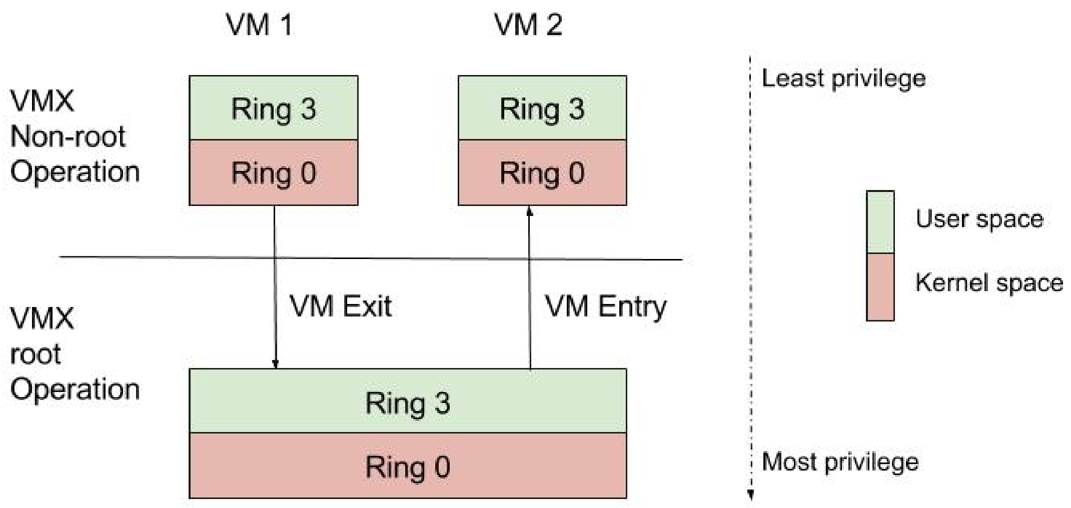

Figure 1-2: Security Mechanisms

and guest libraries to execute user apps. All VMs are running in virtualization non-root operation.

In the user space of the host we have the lightweight hypervisor kvmtool that emulates virtual networks and hardware needed for the VMs. The binary kvmtool is lkvm.

The host OS is a Clear Linux that contains a VMM driver, which is a character device driver called the Linux Virtual Machine Monitor (kvm), that extends to kvm-intel.ko to manage the hardware VT through the device node /dev/kvm.

This host OS contains a modified systemd to reduce the amount of physical memory used by services.

The operations provided by kvm [5] include:

- Virtual machine creation.
- Virtual machine memory allocation.
- Virtual CPU registers read/write.
- Virtual CPU interruption injection.
- Virtual CPU execution.

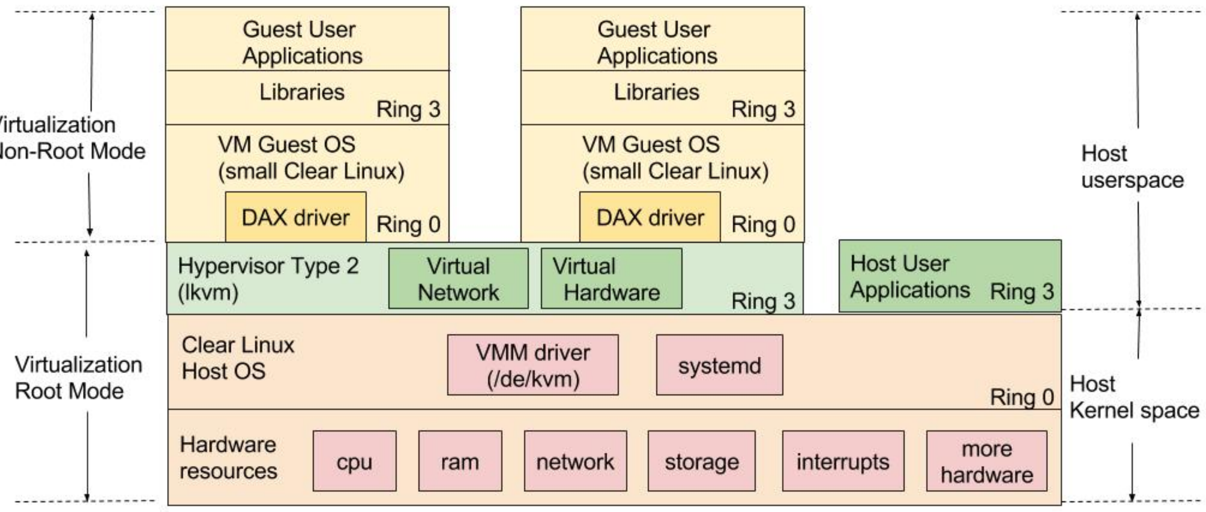

Figure 1-3: Clear Container Software Architecture

So lkvm in user space and kvm in kernel space work together to manage VMs from user apps

1.3 KVM Exit

Transition from non-root operation to root operation generates a KVM Exit, and from the figure 1-3 we can say that it happens when a VM is running and sends control to lkvm. There are many reason to generate a KVM Exit, which can be execution of privilege machine instructions, exceptions, interrupts, access to CPU or Model-Specific Registers (MSRs), Extended Page Table (EPT) violations, access to I/O ports or debug instructions.

The VT hardware traps this KVM Exit, and the kernel driver kvm reads this value and sends the reason to lkvm. lkvm should be able to handle all these types of exits (Appendix A.1) to handle or emulate the operation requested by the VM. When operation is done, the lkvm will transition (KVM Entry) to VM to resume the execution.

As the system can be running multiple VMs, a switch context should be saved and loaded registers EPT used for virtualization. This context contains the pages that the VM has access or belongs to regions and host kernel uses structures Virtual

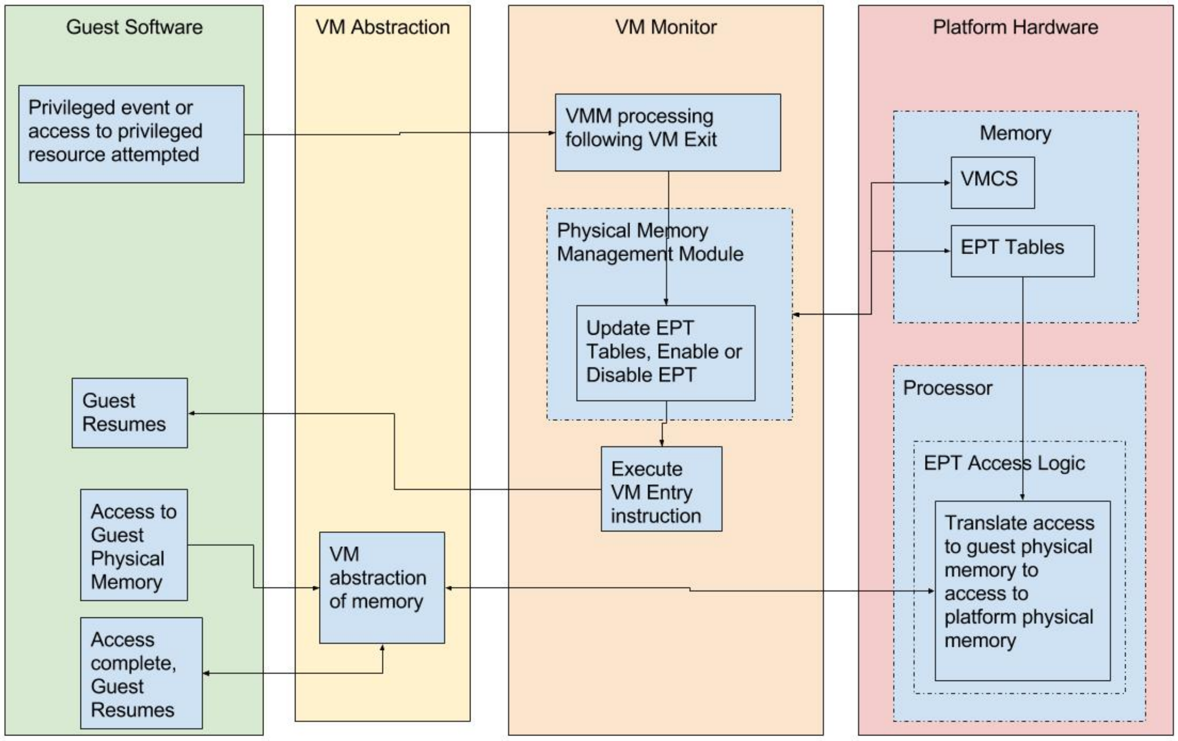
Guest Software
VM Abstraction
VM Monitor
Platform Hardware
Privileged event or access to privileged resource attempted
VMM processing following VM Exit
Memory
VMCS
EPT Tables
Physical Memory Management Module
Update EPT Tables, Enable or Disable EPT
Processor
Guest Resumes
Execute VM Entry instruction
EPT Access Logic
Access to Guest Physical Memory
VM abstraction of memory
Translate access to guest physical memory to access to platform physical memory
Access complete, Guest Resumes

Figure 1-4: Virtualization flow

Machine Control Structure (VMCS) and EPT Tables to saves information in host memory. Figure 1-4

1.4 VM Memory Management

In normal operation of Intel processors, the Linux Kernel uses CR3 register to point to the base address of Page Directory Table (PDT), plus structures Local Descriptor Table (LDT) and Global Descriptor Table (GDT), and Memory Management Unit (MMU) hardware to translate a linear addresses of a process to physical addresses in Random Access Memory (RAM). Similar hardware is needed to translate from a linear address of a guest process to physical address of VM and then convert to a physical address in RAM (this last conversion is done by the MMU).

VT provides the EPT logic hardware and the structures EPT tables and VMCS as seen in figure 1-5 to convert from linear address of guest process to physical address of VM. This help to maintain separated RAM physical address space for each virtual

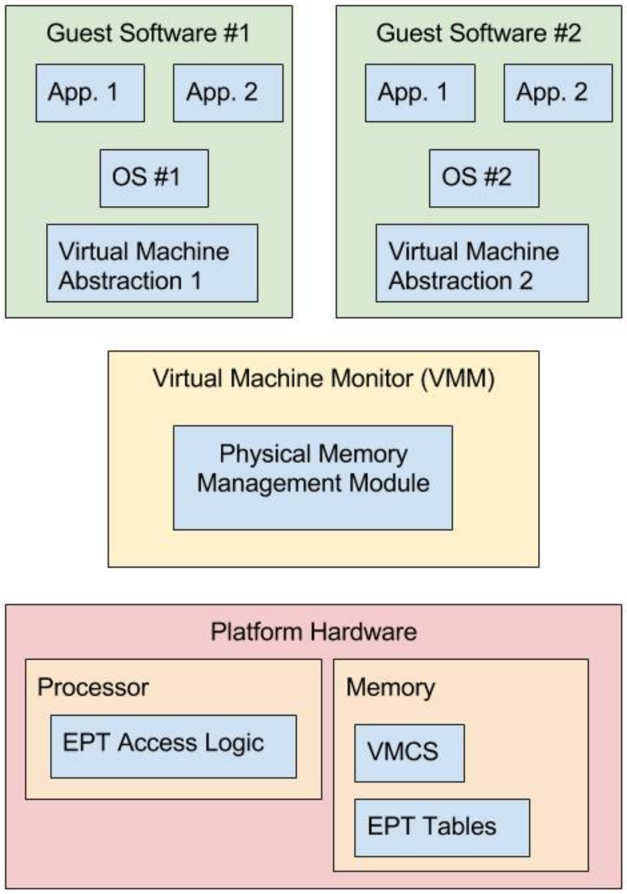

Figure 1-5: Main components of virtualization

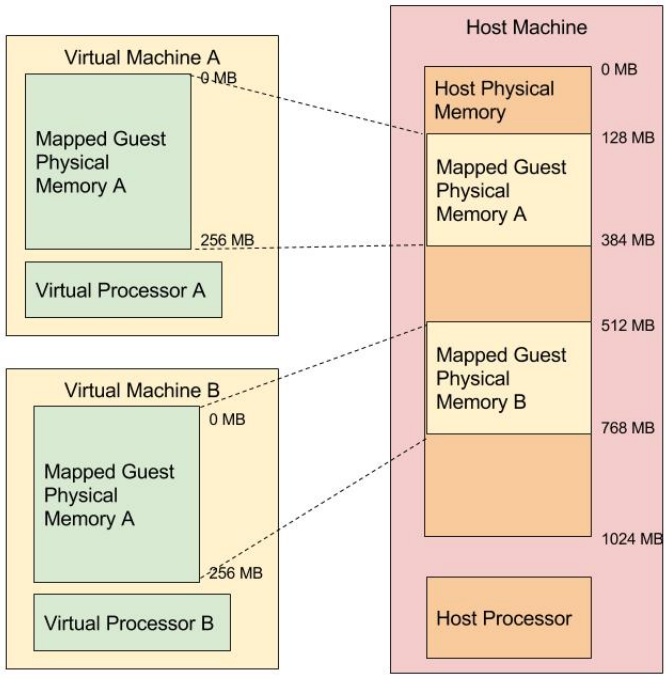

Figure 1-6: VM Guest memory map in host memory

machine. This happens because a lkvm creates user process to represent Virtual CPU (VCPU) and execute all tasks of the VM.

The structures EPT and VMCS to translate VM to physical address of host are in ram owned by kernel. In 1-6 we can see the isolation between virtual machines address spaces memory assigned for each VM.

1.5 Why Guest OS uses a DAX driver?

In Linux when using block devices, the page cache is usually used to buffer reads and writes to files. It is also used for performance reasons to avoid waiting reading for information from Hard disk, which is slower, and write to an intermediate buffer that provide the pages which are mapped into userspace by a call to mmap. For devices like memory, the page cache pages is unnecessary and no copies of the original storage is needed. The DAX code removes the extra copy by performing reads and writes directly to the storage device. For file mappings, the storage device is mapped directly into userspace. To access ram memory as hard disk or Non-Volatile Memory (NVM),

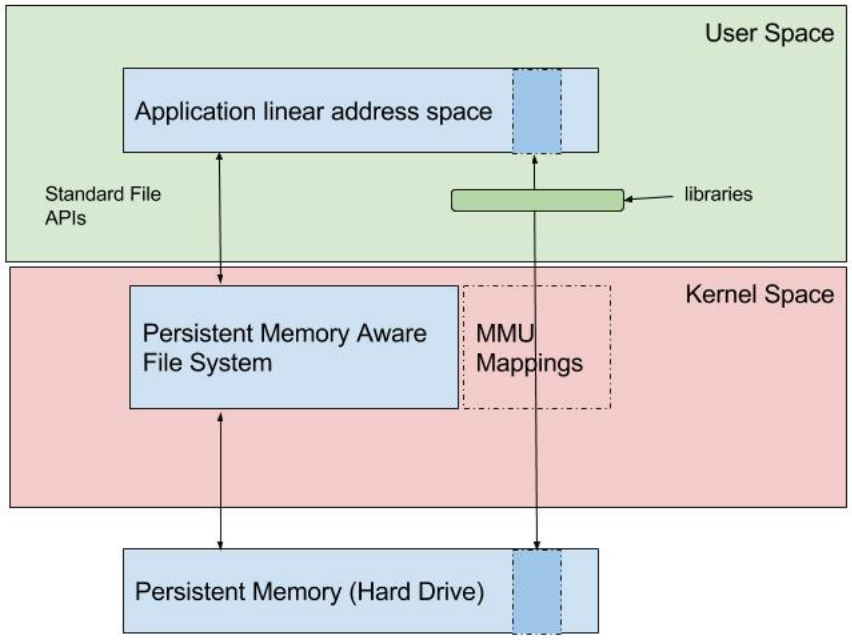
User Space
Application linear address space
Standard File APIs
libraries
Kernel Space
Persistent Memory Aware File System
MMU Mappings
Persistent Memory (Hard Drive)

Figure 1-7: Using hard disk as RAM

a dax driver is needed because ram memory uses page frame number and hard disk doesn't.

Clear Containers uses a DAX driver to reduce the time when creating VMs. The guest Operating Systems (OSs) use an uncompressed Small Clear Linux located in the hard disk (File System (FS)) and then DAX does a memory map of the image. As in figure 1-7, this bypasses the cache buffer in the kernel and cuts the time by not decompressing the image. All this functionality is implemented in the host and when the guest reads or writes to ram address corresponding to the memory mapped, then the Kernel will redirect to hard disk.

Because the Linux image was mapped in ram, now the guest OS needs a driver to access host ram and use it as NVM, and execute from there. When accessing the image, the guest OS will go to VM ram, and it will redirect the access to host ram which points to hard disk where the image file is located. This method is like using many pointer one after another to finally point to the physical file. Figure 1-8

And because guest OS will execute accesses to ram addresses directly, it is con-

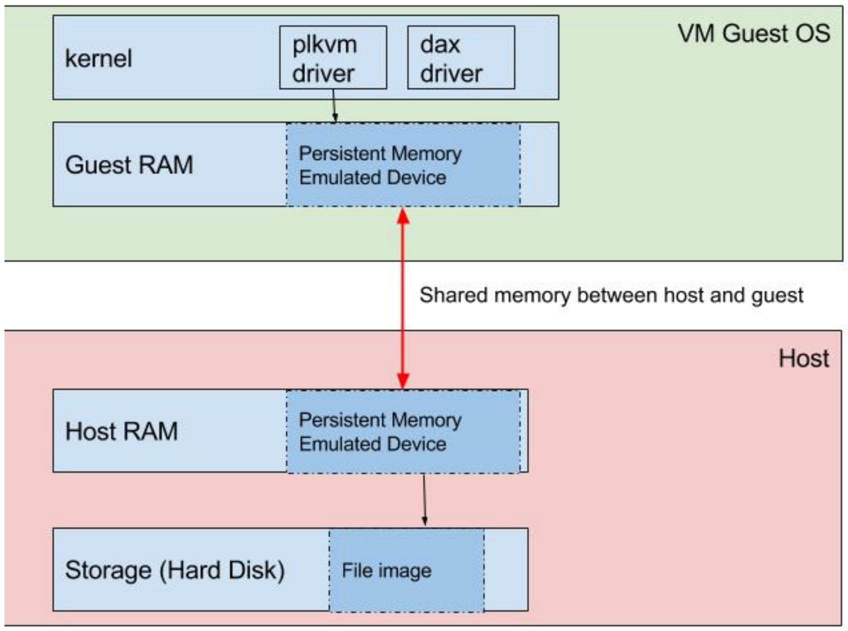
VM Guest OS
kernel
plkvm driver
dax driver
Guest RAM
Persistent Memory Emulated Device
Shared memory between host and guest
Host
Host RAM
Persistent Memory Emulated Device
Storage (Hard Disk)
File image

Figure 1-8: Shared memory between guest and host

sidered privileged and the lkvm must implement code to emulate accesses to physical ram, but in fact it will point to image file.

1.6 Virtio

Because traps generated by KVM Exits can be time consuming when switching from guest to host and vice-versa, virtio devices provide a solution. Virtio is a kernel module that implements direct Hardware (HW) communication without having a KVM Exit. Guests that want to communicate with hardware devices can do it through virtio. Guest implements the front-end and host the back-end. This means that guest knows it is running in a virtual environment. Figure 1-9.

1.7 A Walk Through

This section explains step by step the process when a user needs to create a new virtual machine and execute a user guest application.

1. A host user opens a Linux console, then execute lkvm pass command line ar-

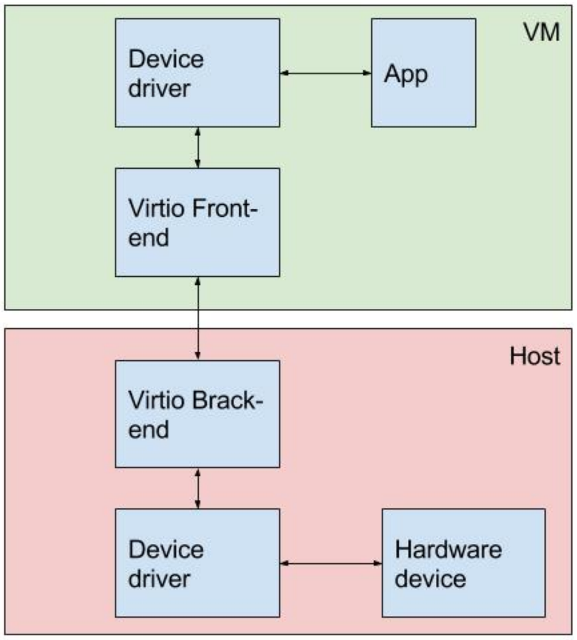

Figure 1-9: Virtio drivers

guments to create a new VMs with the number of Central Processing Units (CPUs), ram memory, etc.

2. lkvm starts execution and it first need to check if the processor supports hardware virtualization (this task can be done using cpuid instruction).

3. lkvm will open the decompressed image Small Clear Linux and map it to the VM physical address space.

4. lkvm will open the device node /dev/kvm (kernel) and request the version of the Application Programming Interfaces (APIs) supported (ioctl `KVM_GET_API_VERSION`) in kvm and possibly check available extensions (ioctl `KVM_CHECK_EXTENSION`) to know what functionality can be used.

5. kvm will execute ioctl and return the information requested by lkvm.

6. lkvm creates a new virtual machine using ioctl `KVM_CREATE_VM`

7. kvm executes the ioctl and returns a handler to lkvm. This handler is used to manage and configure the VM.

8. lkvm reserves ram memory using the function mmap() and will assign the memory to the VM (ioctl `KVM_SET_USER_MEMORY_REGION`) which will represent the size ram physical memory available for the guest.

9. kvm configures VMCS ready to convert guest linear addresses to ram. The memory is passed as parameters in the `kvm_userpace_memory_region` structure.

   And because there is an already mapped memory is used on the host with the Small Clear Linux uncompressed, the VM can have access to these binary in the guest physical memory.

10. lkvm creates a virtual CPU using the ioctl `KVM_CREATE_VCPU` which needs to allocate memory for the structure `kvm_run` that contains all the information of virtual CPU that is used when `VM_EXIT` to keep processor states. This structure contains information `kvm_run` registers, flags, control registers. lkvm then configures initial values in registers in Virtual CPUs (VCPUs) using `KVM_GET_SREGS`.

11. kvm configures registers and structures. Virtual Machine, virtual CPUs, and ram memory mapped are all initialized. At some point, register will point to the first instruction of the Linux image.

12. lkvm can start running the VM Guest using the ioctl `KVM_RUN` passing as parameter the VCPU to run. Due to DAX mapped the uncompressed small Clear Linux image the code can start executing the operating system kernel.

13. KVM Entry, the VM starts execution of the Small Clear Linux and lkvm waits for a KVM Exit in a loop.

14. A guest user app might try to write to specific memory port using a system call.

15. The guest OS executes interrupt 80 (guest can handle this) and call the appropriate system call.

16. The guest will open the device driver to communicate with the network and use the Front-end virtio.

17. The guest back-end will fill required queue for the back-end.

18. lkvm back-end virtio will catch handle the task requested.

19. lkvm back-end open and send to the network driver.

20. back-end receives results from network driver and send to front-end virtio.

21. guest user receives the information.

22. another guest app wants to open serial port and open the device node

23. The guest OS kernel writes and executes privileged instructions to write to memory addresses.

24. KVM Exit, lkvm receives the reason and should handle this appropriately in one of the cases. ioctl(`KVM_RUN`) returns a value containing the reason of KVM Exit.

    Figure 1-10

25. lkvm emulates the write to memory.

26. Execute again ioctl(`KVM_RUN`)

27. Steps 1 to 26 can be repeated in another lkvm app to create another VMs and each one with its own physical address space. Figure 1-6

Clear Linux host is running Kernel Shared Memory (KSM) to search for memory pages with the same content in every physical memory of VM, so that VMs can reference to a single page in host physical memory ram. If any virtual machine alter the content of a sharing page, the host kernel will create a new page for that VM.

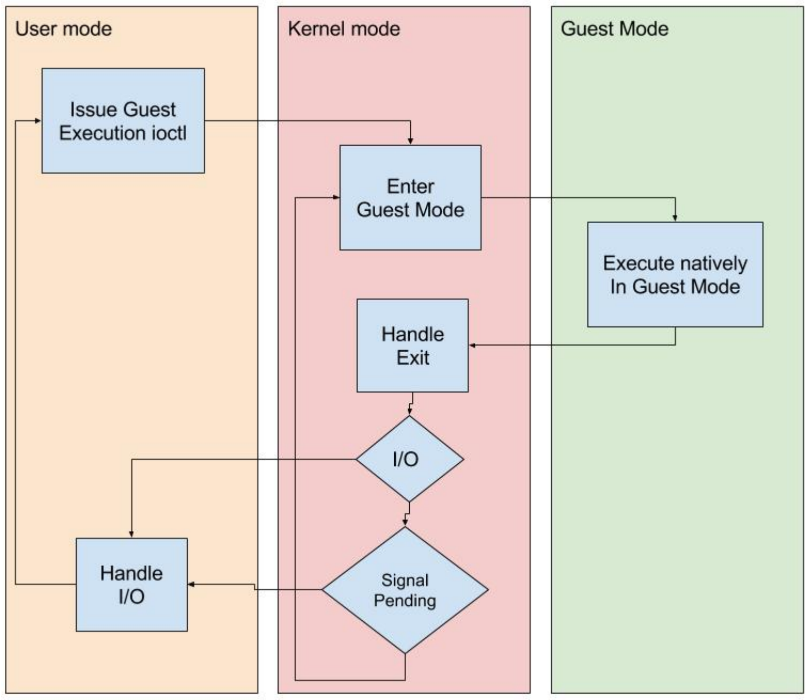

Figure 1-10: Interaction between guest and host

Chapter 2

Threat Modeling Methodology

Threat modeling is a methodology to assess and document security risks in a system or application. It helps development teams to identify both the security strengths and weaknesses of a system and can be the base for investigating potential threats, testing and find vulnerabilities. [6] [7]

Although it is impossible to assign a quantifiable security rating to a system, threat modeling provides a basis for characterizing relative security.

2.1 Threat Modeling Stages

Threat modeling has three stages: Understanding the adversary's view, Characterize the security of the system and Determine threats. Figure 2-1

2.1.1 Understanding the adversary's view

Threat modeling takes an outside-in as a black box approach because such an approach is more close to the attackers that don't have access to source code or architecture information. This models the adversary's view, and therefore understands the adversary's view by enumerating entry points and assets, as well as cross-referencing them with the trust levels.

Entry points are any interface where data or controls can receive or send between

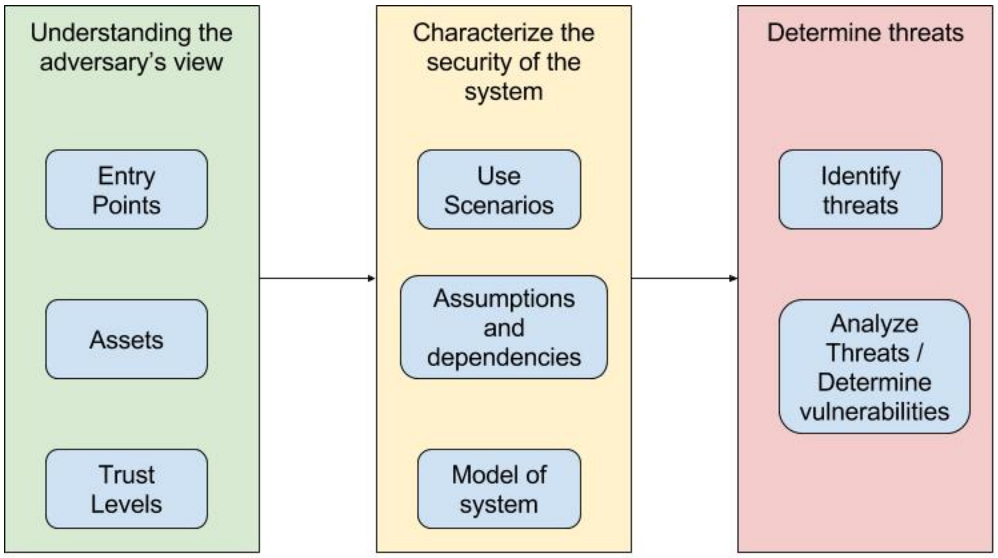
Understanding the adversary's view
Entry Points
Assets
Trust Levels
Characterize the security of the system
Use Scenarios
Assumptions and dependencies
Model of system
Determine threats
Identify threats
Analyze Threats / Determine vulnerabilities

Figure 2-1: The high-level process of threat modeling

the system being modeled and another system. This entry points are how an adversary can interface with the component a attack the system such as open sockets, Remote Procedure Call (RPC) interfaces, Web services interfaces, and data being read from the file system. Entry points should be listed regardless of the privilege level required to interact with them.

Assets are resources, components or system that an adversary might try to modify, steal, or otherwise access or manipulate.

Trust levels are the privileges assigned or credentials needed by external entities to access this component.

Trusted Computing Base (TCB) is a set of all hardware, firmware, and/or software components that are critical to the security of the system, in the sense that bugs ocurring inside the TCB might jeopardize the security properties of the entire system.

2.1.2 Caracterizing the security of the system

This stage is more like white box which involves limiting the threat model, gathering information about dependencies that are critical to security, and understanding how the system works internally.

Define scenarios this shows how the component or system will be used and how can misused.

Identify assumptions and dependencies collects information like external and internal dependencies, security notes, and implementation assumptions how system should work.

Model the system with Data Flow Diagrams (DFDs) visual representations of how a system process data and to understand the actions performed at a given component.

2.1.3 Determining Threats

Describes all the potential attacks in the system.

Identify threats determines how an adversary might try to affect an asset. Using DFDs can help to know what the adversary might try to do from an external entity to manipulate an asset at a given entry point, how to reach the asset, what path to follow and which trust levels are needed. Having this threats, classify them with STRIDE and rate is each threat as high, medium or low in a scale for evaluation.

Analyze threats determine whether threats are mitigated already or not. Using diagrams like threat trees, decompose a threat into individual, and testable conditions.

2.2 Classification of Threats

Once threats are identified, they should be categorized. The model commonly used for software is STRIDE. This is a classification of the effects of realizing threat:

- **Spoofing** allows an adversary to use others credentials to pose as another user, component, or other system.
- **Tampering** is the modification of data within the system.
- **Repudiation** is when the attacker did evil and the victim has no way to proof who did it.

- **Information Disclosure** is the exposure of protected data to a user not allowed access to that data.
- **Denial Of Service** attack the system to stop providing services of the system.
- **Elevation of Privilege** is when process/entity gains more privilege that it have.

2.3 Scale of Rating

Damage, Reproducibility, Exploitability, Affected users, and Discoverability (DREAD) is a classification scheme for quantifying, comparing and prioritizing the amount of risk presented by each evaluated threat. The DREAD acronym is formed from the first letter of each category below. DREAD modeling influences the thinking behind setting the risk rating, and is also used directly to sort the risks. The DREAD heuristic isn't a rigorous classification and doesn't mean that it isn't useful. It is useful in helping people focus the debate about what to do about some specific problem. As shown below, is used to compute a risk value, which is an average of all five categories.

`Risk_DREAD` = (DAMAGE + REPRODUCIBILITY + EXPLOITABILITY + AFFECTED USERS + DISCOVERABILITY) / 5

The calculation always produces a number between 0 and 10; the higher the number, the more serious the risk. The severity can be given by the damage, reproducibility and affected users, because they tend to remain static, while the other two factors (exploitability and discoverability) varies a lot depending on the situation.

Here are how to quantify the DREAD categories.

Damage

This has to be judged in the context of the system that provides virtualization. If a threat exploit occurs, how much damage will be caused? How bad would an attack be?

0 = Nothing.

5 = VM compromised.

10 = Host compromised.

Reproducibility

How easy is it to reproduce the attack? How well it works once you trigger the attack.

0 = Very hard or impossible to reproduce.

5 = Few steps executed correctly under a condition

10 = Always, no need for special conditions.

Exploitability

How much work takes to launch the attack? What is needed to exploit this threat?

0 = No exploitability.

5 = Skilled programmer.

10 = Automated/script kiddie.

Affected Users

How many users will be affected or impacted?

0 = None.

5 = Only VM under attack.

10 = All users. (Even host or other VMs).

Discoverability

How easy is it to discover the threat?

0 = Requires intimate knowledge of internal workings of hardware and software, architecture or bugs already in the system.

5 = Can figure it out by guessing or by monitoring network traces, code review, or other technique.

10 = Published.

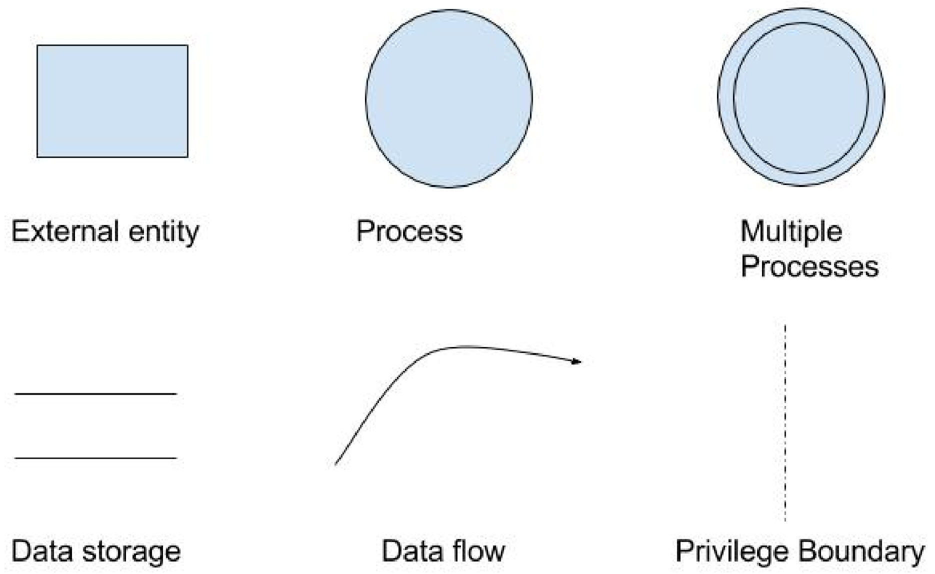

Figure 2-2: Data flow symbols

2.4 Symbology

Data flow diagramming focuses on data as it moves through the system and the transforms that are applied to that data. Six basic shapes are used in DFDs for threat modeling as shown in figure 2-2.

- **Process** represents a task in the system that processes data or perform some actions based on the data.
- **Multiple processes** is made up of sub processes. Should be used whenever the process is further broken down in a lower-level DFD.
- **External entity** represents an interactor that exists outside the system being modeled and which interacts with the system at an entry point.
- **Data store** represents a repository for data such as registers, file system, or database where data is saved or retrieved.
- **Data flow** represents data being transferred between other elements.
- **Privilege boundary** represents the boundary between nodes that have different privilege levels.

2.5 Choosing what to model

Doing a threat model of the right features is critical. Threat modeling every feature or component is not always possible because time constraints. Because systems can be huge and have many interactions with other components like data bases, OSs, other programs, drivers, etc. the high priority are components that will receive user-supplied data or that interact with the user. Then focus on important components like security mechanisms, parsers, component with very bad reputation in security.

Chapter 3

Threat Modeling of Clear Containers

Many users think they are more secure having a VMM, but it is just another layer to find bugs! Not to mention that goal of Intel is performance.

An important goal of software security is to ensure sensitive data owned by a program must be exclusively accessible by that program. In this context, sensitive data includes, but is not limited to, the confidentiality, integrity, and availability. Much of the security relies in the security architecture of the Operating System of the host which dictates the types of communication authority, the mechanisms to isolate resources and properly enforce access control, auditing, and determining what subjects and objects reside in specific domains, each resource has to be clearly separated from one another.

Some general threat models has been done in the past [8], but need an update and check which ones apply to this project and this chapter shows the result of the Threat Modeling methodology applied to Clear Containers.

This chapter follows each stage accordingly figure 2-1, but this project will not include the proposed mitigations to threats marked as high in the final result.

3.1 Assets

Having access to confidential information is the most valuable assets when escaping from a Guest VM.

- Confidential information in VMs.
- VMCS Structures.
- VM Escape figure 3-1.
- Denial Of Service (DoS) of VMs.
- The Small Clear Linux.
- Host hardware Resources.
- Network traffic.

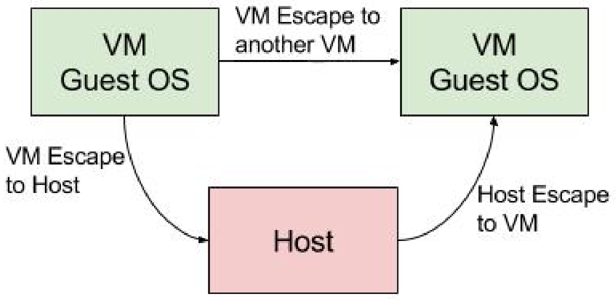

Figure 3-1: Virtual Machine Escapes

3.2 Entry Points and Trust Levels

The table 3.1 shows places where control is transferred to some components of the system from external or internal interactors.

The attacker usually take advantage of bad interfaces that do not well handle input parameters, passing crafted values that might corrupt the program, or execute arbitrary code [9].

No.	Entry point	Trust level
1	system libraries for lkvm	root operation
2	lkvm events for emulated hardware	non-root operation, user
3	lkvm traps	non-root operation, user
3	lkvm virtio back-end	non-root operation, user
4	shared folder	non-root operation, user
5	DAX simulation	non-root operation, user
6	lkvm command line parameters	root operation, user
7	virtual network	non-root operation, user
8	Connections from physical network	unauthenticated user
9	/dev/kvm APIs	root operation, user
10	Hardware that operate independently of the system	root operation, kernel
11	Improperly sanitized shared HW response	root operation, kernel
12	Shared L3 cache	root, kernel

Table 3.1: Entry points with trust level

3.3 Characterize the Security of the System

From the Clear Linux website, it says that Clear Containers has done some changes to the following components:

- Patches applied to kvmtool.
- Optimizations in the kernel.
- Replaced malloc with `Kmalloc_trim()` in systemd.
- A DAX driver was added to the Small Clear Linux guest to enable faster filesystem accesses.
- KSM is enabled in the Clear Linux host.
- Optimizations of core user space for minimal memory consumption.

Intel customized kvmtool, systemd, and a Small Clear Linux packages to improve performance, but not all changes affects security.

From the patches applied in systemd, the only interesting is the one that replaces malloc by `malloc_trim` to return heap memory to the OS.

The small Clear Linux implement a DAX driver, and because the owner of a VM has root access he can install or remove drivers from guest OS. This is another entry point for emulated hardware.

Patches applied to kvmtool implement the emulation of DAX that interface with the DAX Driver on the OS Guest, this add support to map the uncompressed Small Clear Linux. A new thread was created to check unlinked files. And Customization to Dynamic Host Configuration Protocol (DHCP) to not read the Ethernet header and also handle situations with no Internet Protocol (IP) address.

From within all the patches applied to Clear Containers and prioritizing, the only interesting changes to check are the ones applied to kvmtool.

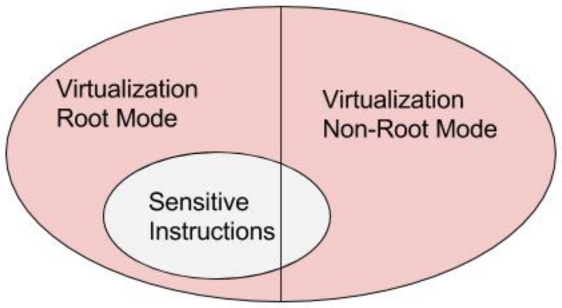

Figure 3-2: Privilege Instructions Space

3.3.1 Assumptions

The security of the system is based on some trusted entities, If those entities have no security, then the system has no meaning.

- VT hardware is secure. Having a good hardware design able to trap privilege instructions, and the lkvm driver having a good configuration to enable those traps.
- The uncompressed small Clear Linux image is in a folder with protections.
- The Intel processor is 64-bit in protected mode.
- The host machine is in a secure area, with restricted access.
- The host has secure configurations.
- The VM owner has root access.
- In development phases, Security-Enhanced Linux (SELinux) is not used, however is planned to be activated in production.

3.3.2 Data Flow Diagrams

The figure 3-3 and figure 3-4 shows a context data flow diagrams that is drawn for a Clear Containers system. It shows the participants who will interact with system, called the external entities (Physical Network). In figure 3-3 Guest VM, Physical Network, lkvm, Clear Linux Host OS, Systemd, VT hardware, kvm, physical memory, and physical hard disk are the entities who will interact with the system. In between the entities, there are data flow (connectors) that indicate the information exchange.

From the chapter 1.7, we are now able to understand figure 3-3 and figure 3-4.

As we can see the network interface is the less trusted entity because it is connected to the internet world and data is send to VM or host. lkvm, which is in the middle of the system, receives and manage most of the data sent from VMs. Systemd is part of the system, but it was just modified to manage memory from services and it does not contribute to the security of the system. kvm is in kernel space in the host and it only receives information from lkvm and VT hardware.

The Small Clear Linux images is stored in the hard disk which is one of the entry points to VMs.

Because lkvm receives most the entry points, is the most vulnerable component as we can see in figure 3-3.

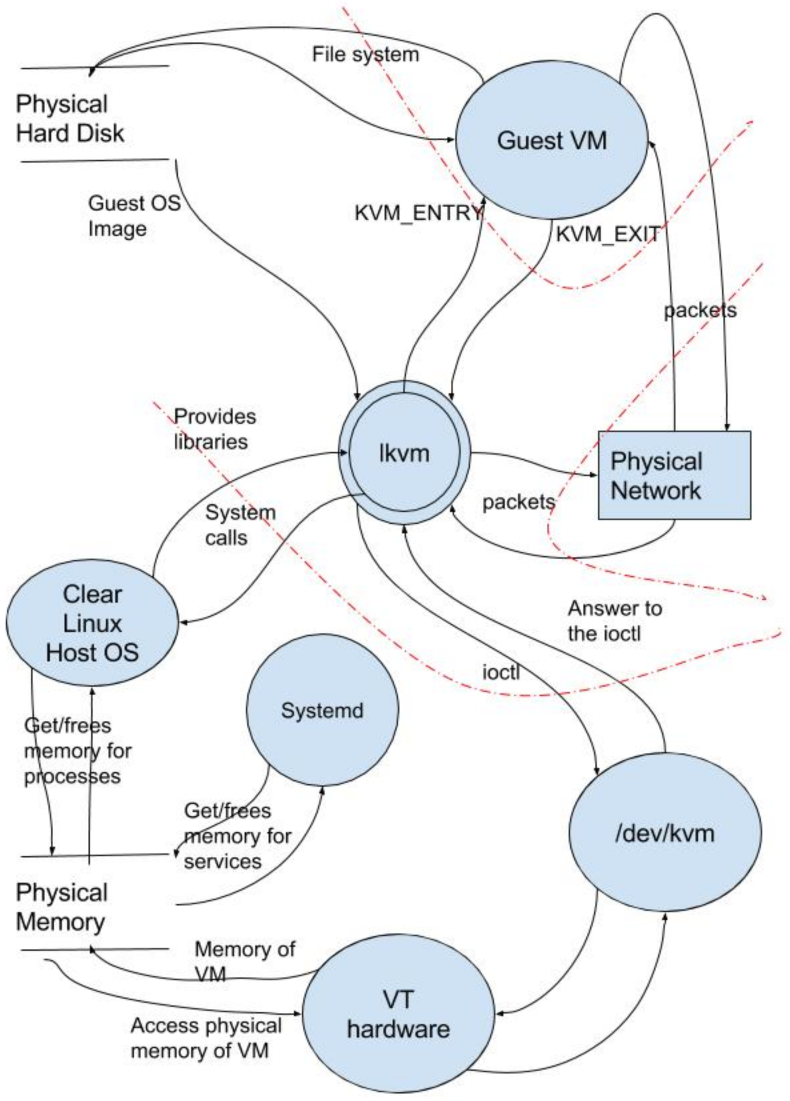

Figure 3-3: DFD of Clear Containers architecture

In figure 3-4, the main entities of lkvm are DAX, Virtio back-end, virtual network, KVM exit handler, and a threat to clean files.

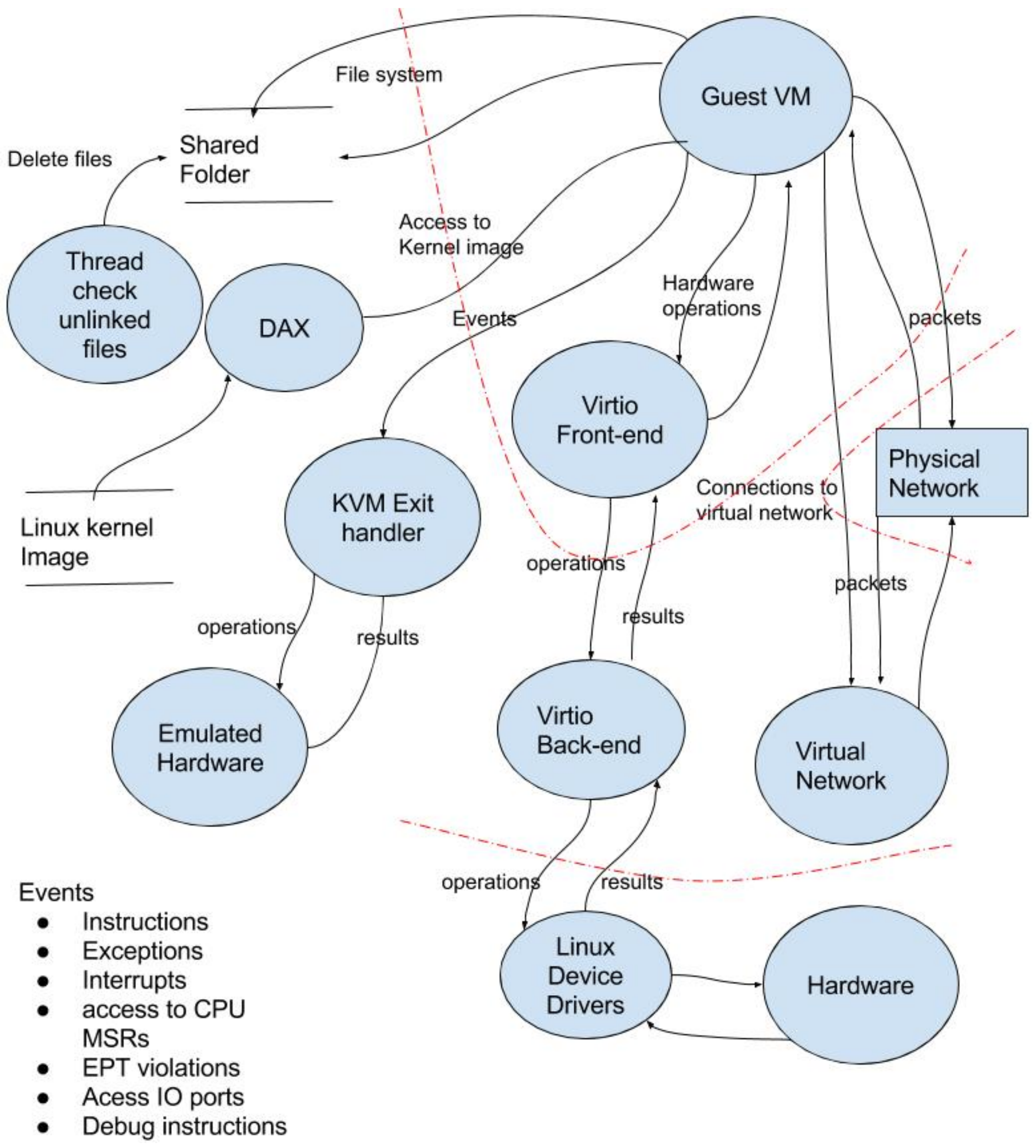

Figure 3-4: DFD of lkvm

3.4 Determine Threats

Every threat is evaluated using DREAD, where the first value will be damage, the second is reproducibility, the third is exploitability, the fourth is affected users and the last one is discoverability

Threats	D	R	E	A	D	Risk
1 - Attack virtio back-end.	10	5	5	10	5	**7**
A malicious guest might try to send crafted instructions to the back-end virtio, to execute code in host. **Risk reason:** Mitigation: Virtio back-end has a parser to process commands from the virtio front-end. Impact: An issue in the parser might allow a VM to execute code in host as a Linux user and do more evil.						
2 - Attack virtualized networks.	10	5	5	10	5	**7**
Because an attacker has access to source code implementation of lkvm, the attacker might find security issues in network implementation that can be exploited in virtual networks using crafted packages to escape to host [10]. **Risk Reason:** Mitigation: Not all VMs has access to virtual networks and lkvm sends packages to Linux devices that handles routing. Impact: An issue in the parser might allow a VM to execute code in host as a Linux user and do more evil.						

Continuation of table						
Threats	**D**	**R**	**E**	**A**	**D**	**Risk**
3 - Vulnerabilities in emulated hardware.	10	5	5	10	5	**7**
An attacker in a VM might try to use hardware (i8042, pci-shmem, rtc, serial, vesa or new hardware) that will be emulated by source with programming issues. **Risk Reason:** Mitigation: lkvm validates inputs from VMs that represents accesses to hardware. Impact: An issue in the parser might allow a VM to execute code in the host as a Linux user and do more evil.						
4 - Using guest DAX driver to access other virtual address in the lkvm process space.	5	5	5	5	5	**5**
A DAX driver in the guest OS might try to access ram memory out scope and lkvm emulates this to provide the Linux binary image. **Risk reason:** Mitigation: lkvm validates address ranges that only belongs to the VM. Impact: A bad implementation of the emulation might give access to other memory regions on VM process space.						

Continuation of table						
Threats	**D**	**R**	**E**	**A**	**D**	**Risk**
5 - Attack drivers shared by many guests.	10	5	5	10	5	**7**
An attacker might try to use hardware handled by virtio that manages requests from different VMs and corrupt the system. **Risk reason:** Mitigation: The back-end virtio handles and validates all driver requests and then send tasks to hardware. Impact: If host contains bad quality Linux kernel drivers, then an attacker might try create a race-condition and cause DoS.						
6 - Exploiting common vulnerabilities in all Guest OSs.	5	5	5	5	5	**5**
Because all VM guest OSs use the same un Small Clear Linux image, an attacker might find a vulnerability in the image and exploit it in all guest OSs. **Risk reason:** Mitigations: Enable Linux Kernel protections and binary protections. Impact: Only the VM under attack gets compromised.						
7 - Overlap in the memory managed by kvm or lkvm.	10	5	5	10	10	**8**
Sharing page memories of between VMs, might confuse the VMM to think it has more memory available, and allocate used pages to a new VM. **Risk reason:** Mitigations: the host Linux and lkvm handles ram memory using mmap, they will not allow overlaps. Impact: Access to pages with confidential information from other VMs, but attacker might not know when it is exploitable.						

Continuation of table						
Threats	**D**	**R**	**E**	**A**	**D**	**Risk**
8 - Attack guest OS network services.	5	5	5	5	10	**6**
This is the same as having a physical machine. An attacker might try to exploit a vulnerability in any of the services listening ports. **Risk reason:** Mitigation: Each of the services validates inputs from network. Impact: Only affects the VM under attack.						
9 - Starvation of hardware resources.	5	5	5	5	5	**5**
A guest VM might create lot of hardware operations directly in the host driver like network bandwidth without limits. **Risk reason:** Mitigation: Each VM is handled as a user process so it has a time slice, and cannot consume all resources. Impact: DoS on VMs that needs hardware operations.						
10 - Leak processes address space layout from other VMs.	0	5	5	0	5	**3**
When using KSM, an attacker can try to guess the address-space layout of processes running on other VMs.[11]. **Risk reason:** Mitigation: This attack only works if only two VMs are running, and will only return information on shared memory, like read-only code segment. Impact: A malicious VM might detect apps and their versions running on other VMs.						

Continuation of table						
Threats	**D**	**R**	**E**	**A**	**D**	**Risk**
11 - Exhaust host ram.	0	0	10	10	10	**6**
A user attacker in the host might execute simple programs to consume ram memory and causing to lower the performance of VMs. **Risk reason:** Mitigation: Host Linux memory has over-commit handling modes. Impact: An attacker might cause DoS.						
12 - Steal VM	10	5	5	10	5	**7**
Copying and pasting the content of VMs to get access to confidential information. **Risk reason:** Mitigation: Host Linux uses Access Control List (ACL) to set permissions in folders containing the VMs. Impact: If VMs users do not encrypt their information, the attacker might still them.						
13 - Use rare functionality in kvm driver.	10	5	5	10	5	**7**
An attacker might exploit a vulnerability in functionality not well validated, and not trapped by VMM. Therefore, instruction will be executed with ring 0 privilege. For example 16-bit binaries that uses i8086 instruction set. **Risk reason:** Mitigation: lkvm implementation should not allow to access these rare features. Impact: An attacker might execute virtual functionality as a exploit chain.						

Continuation of table						
Threats	**D**	**R**	**E**	**A**	**D**	**Risk**
14 - /dev/kvm APIs not validating parameters.	10	10	5	10	5	**8**
Because a user app can use kvm APIs, an attacker might send malformed parameters to /dev/kvm to find issues and exploit them to gain kernel privilege. **Risk reason:** Mitigation: kvm validates input parameters. Impact: Attacker might exploit and execute code in root host.						
15 - Creating VMs not using mmap private	10	5	5	10	5	**7**
An attacker might fool to create VMs that will keep changes made in the shared Linux image. **Risk reason:** Mitigation: This is controlled by the lkvm and only the Linux host can manage it. Impact: If bad implementation in lkvm, it will exploit other VMs sharing not using private.						
16 - Use virtio to delete files in host	10	10	5	10	10	**9**
An attacker might corrupt the thread (virtio 9p) that checks unlinked files to deleted other files or crash lkvm. **Risk reason:** Mitigation: lkvm validates names of files to delete. Impact: If bad implementation in lkvm, attacker might crash lkvm or access other files in the host filesystem.						

Continuation of table						
Threats	**D**	**R**	**E**	**A**	**D**	**Risk**
17 - Install malicious kernel driver in host.	10	5	5	10	5	**7**
An attacker might find a vulnerability in the host to fool the users and try to install a malicious kernel module to manipulate VMCS structures to get access to pages belonging to specific guest [12]. **Risk reason:** Mitigation: Host Linux uses ACL to not allow any user to install drivers. Impact: Full control of the system.						
18 - Attacking binaries in the infrastructure.	10	5	5	10	0	**6**
Binary installation if affected by the security mechanism provided by Linux, so an attacker will try to modify binaries, hijack libraries, inject exploits from resource with weak permissions. **Risk reason:** Mitigation: Host Linux uses binaries protections and ACL. Impact: The attacker might install malware on the host Linux to later do evil on VMs.						
19 - Hyperjacking	10	0	0	10	0	**4**
Manipulate boot files to convert a host OS into a VM. **Risk reason:** Mitigation: Host Linux uses ACL to protect boot files. Impact: Gain control of the whole system.						

Continuation of table						
Threats	**D**	**R**	**E**	**A**	**D**	**Risk**
20 - Malicious trusted hardware.	10	0	0	10	0	**4**
An attacker with physical access, might install hardware devices that work independently of the host OS can attack buses, registers or other hardware connected [13]. **Risk reason:** Mitigation: Out of scope. Impact: compromise of host machine.						
21 - Cold boot attack.	10	0	0	10	0	**4**
Attacker with physical access to the host, can reboot and extract information from ram. **Risk reason:** Mitigation: Out of scope. Impact: Steal unencrypted information in ram.						
22 - Cache side-channel attack [14] [15]	10	0	0	10	0	**4**
Attacker interfere with victim on a host and exfiltrate sensitive information. **Risk reason:** Mitigation: needs further research. Impact: Steal cryptographic keys, but it is difficult to guest from which VM is belonging the information.						

Table 3.2: DREAD rating for Clear Containers threats

3.5 Mitigations

3.5.1 User Process Crossing to Another User Process

Two processes running in ring 3 are arranged in a linear address of 4 GB. Where instructions like ljmp (jump to other segments) is allowed to execute, and since the CPU is unable to identify which Code Segment (CS) belongs to processes, the Kernel selects segment descriptors from the current LDT and Task State Segment (TSS). No matter how the operands are written, it is unable to cross the current process code because a process only has access to segments in LDT mapped that are mapped to the physical memory used by this process.

Similar technique applies for linear address space of guest process to try to access another VM.

3.5.2 One process crossing to kernel

The user process privilege level is 3, and the kernel privilege level is 0. Trying to access the kernel memory mapped in physical memory, a user needs to access to GDT, LDT and TSS of the kernel process, but is not possible because it needs to set CR3 register to point to a fake/created PDT but that cannot be possible because the instruction to load CR3 with a value is privileged in ring 0.

Similar technique applies to protect guest process to access host process by protecting privileged instruction that sets values to register EPT.

3.5.3 Protection to Binaries

Address Space Layout Randomization (ASLR) changes the address of shared libraries and code to prevent from exploiting vulnerabilities in binaries running in guest and host.

Relocation Read-Only (RELRO) protects start-up to allow linker to load and link functions and mark them as read-only.

Position Independent Executable (PIE) ASLR being forced on every process, not every memory area is randomized for all executables. The code segment (or text segment; .text) of the main binary is located at random locations only if the executable has been compiled as a Position Independent Executable (PIE).

Canary prevent an attacker to exploit buffer overflows by adding special value just before the saved return address on the stack, which checks the value, is an effective way to detect stack has not been overwritten.

No Execute (NX) prohibit execution of code stored in memory for data.

RPATH is a hard-code list of directories that may contain shared libraries needed by the app.

3.5.4 Access Control List

Linux provides with user and group permissions to access files. This prohibit unauthorized user from reading, executing or writing files.

Chapter 4

Results and Conclusions

4.1 Results

The table 4.1 shows the issues that affect the security of the system. If we divide 1-10 in three parts we get high is from 10-7, medium is 6-3, and low 2-1

Threat	**Risk**	**Rating**	**Type**
Threat 1 - Attack virtio back-end	7	High	E
Threat 2 - Attack virtualized networks	7	High	E
Threat 3 - Vulnerabilities in emulated hardware	7	High	E
Threat 5 - Attack drivers shared by many guests	7	High	D
Threat 7 - Overlap in the memory managed by kvm or lkvm	8	High	I
Threat 12 - Steal VM	7	High	I
Threat 13 - Use rare functionality in kvm driver	7	High	E
Threat 14 - /dev/kvm apis not validating parameters	8	High	E
Threat 15 - Creating VMs not using nmap private	7	High	T
Threat 16 - Use virtio to delete files in host	9	High	E,D
Threat 17 - Install malicious kernel driver in host	7	High	I

Table 4.1: High priority threats with STRIDE

4.2 Conclusions

The Clear Containers architecture proposes a new way to address security and performance using hardware VT, to mitigate the security problems of LXC and the threat modeling presented in this help to understand how it is done.

The output can be used to continue the Security Development Lifecycle (SDL) process, so security validators and developers can now to better the focus on emulated hardware, DAX, virtio back-end and harden the host, to protect the uncompressed Linux image.

We can compare threats in Clear Containers with the list of security issues in LXC [16](Appendix C) in the history. Security flaws in namespaces allow a host user to gain privilege escalation, or VM escapes from guest. Issues related to the use of namespaces are similar to emulated hardware and the problems of isolation are solved with the structures VMCS of VT.

4.3 Future Work

The scope is limited, someone else can continue detailing data flow diagrams. This limits the scope to the results in table 3.2

Taking the output of the threat model and to continue with the SDL process to create threat trees for penetration testing.

These are some of the tasks to find bugs:

- Execute static analysis tools on lkvm source code.
- Secure code review of lkvm to find programming mistakes in the memory management and hardware emulation.
- Create fuzzers to write to memory address, I/O ports, and all the hardware to find vulnerabilities in lkvm.
- Secure code review of the Intel kvm implementation to find weird functionality.
- Patch and fix bugs found in source code.

List of Acronyms

ACL	Access Control List
API	Application Programming Interface
APIs	Application Programming Interfaces
app	application
apps	applications
ASLR	Address Space Layout Randomization
CPU	Central Processing Unit
CPUs	Central Processing Units
CS	Code Segment
CVEs	Common Vulnerability Enumarations
DAX	Direct Access for Files
DFD	Data Flow Diagram
DFDs	Data Flow Diagrams
DHCP	Dynamic Host Configuration Protocol
DoS	Denial Of Service
DREAD	Damage, Reproducibility, Exploitability, Affected users, and Discoverability
EPT	Extended Page Table
FS	File System
GDT	Global Descriptor Table

HW	Hardware
I/O	Input and Output
IP	Internet Protocol
KSM	Kernel Shared Memory
kvm	the Linux Virtual Machine Monitor
LDT	Local Descriptor Table
LXC	Linux Containers
MMU	Memory Management Unit
MSRs	Model-Specific Registers
NVD	National Vulnerability Database
NVM	Non-Volatile Memory
NX	No Execute
OS	Operating System
OSs	Operating Systems
PDT	Page Directory Table
PIE	Position Independent Executable
SDL	Security Development Lifecycle
RAM	Random Access Memory
RELRO	Relocation Read-Only
RPC	Remote Procedure Call
SELinux	Security-Enhanced Linux

STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege
TCB	Trusted Computing Base
TSS	Task State Segment
VE	Virtual Environment
VCPU	Virtual CPU
VCPUs	Virtual CPUs
VMCS	Virtual Machine Control Structure
VM	Virtual Machine
VMs	Virtual Machines
VMM	Virtual Machine Monitor
VMX	Virtual Machine Extensions
VT	Virtualization Technology

Appendix A

Interesting Source Code

A.1 lkvm handle KVM_EXIT

Function in file kvmcpu.c shows how to handle different KVM_EXITs.

```
int kvm_cpu__start(struct kvm_cpu *cpu)
{
  sigset_t sigset;

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGALRM);

  pthread_sigmask(SIG_BLOCK, &sigset, NULL);

  signal(SIGKVMEXIT, kvm_cpu_signal_handler);
  signal(SIGKVMPAUSE, kvm_cpu_signal_handler);

  kvm_cpu__reset_vcpu(cpu);

  if (cpu->kvm->cfg.single_step)
    kvm_cpu__enable_singlestep(cpu);

  while (cpu->is_running) {
    if (cpu->paused) {
      kvm__notify_paused();
```

```
        cpu->paused = 0;
    }

    if (cpu->needs_nmi) {
        kvm_cpu__arch_nmi(cpu);
        cpu->needs_nmi = 0;
    }

    kvm_cpu__run(cpu);

    switch (cpu->kvm_run->exit_reason) {
    case KVM_EXIT_UNKNOWN:
        break;
    case KVM_EXIT_DEBUG:
        kvm_cpu__show_registers(cpu);
        kvm_cpu__show_code(cpu);
        break;
    case KVM_EXIT_IO: {
        bool ret;

        ret = kvm_cpu__emulate_io(cpu,
                cpu->kvm_run->io.port,
                (u8 *)cpu->kvm_run +
                cpu->kvm_run->io.data_offset,
                cpu->kvm_run->io.direction,
                cpu->kvm_run->io.size,
                cpu->kvm_run->io.count);

        if (!ret)
            goto panic_kvm;
        break;
    }
    case KVM_EXIT_MMIO: {
        bool ret;

        /*
```

```
         * If we had MMIO exit, coalesced ring should be processed
         * *before* processing the exit itself
         */
        kvm_cpu__handle_coalesced_mmio(cpu);

        ret = kvm_cpu__emulate_mmio(cpu,
                    cpu->kvm_run->mmio.phys_addr,
                    cpu->kvm_run->mmio.data,
                    cpu->kvm_run->mmio.len,
                    cpu->kvm_run->mmio.is_write);

        if (!ret)
            goto panic_kvm;
        break;
    }
    case KVM_EXIT_INTR:
        if (cpu->is_running)
            break;
        goto exit_kvm;
    case KVM_EXIT_SHUTDOWN:
        goto exit_kvm;
    case KVM_EXIT_SYSTEM_EVENT:
        /*
         * Print the type of system event and
         * treat all system events as shutdown request.
         */
        switch (cpu->kvm_run->system_event.type) {
        default:
            pr_warning("unknown system event type %d",
                    cpu->kvm_run->system_event.type);
            /* fall through for now */
        case KVM_SYSTEM_EVENT_RESET:
            /* Fall through for now */
        case KVM_SYSTEM_EVENT_SHUTDOWN:
            /*
             * Ensure that all VCPUs are torn down,
         * regardless of which CPU generated the event.
         */
        kvm__reboot(cpu->kvm);
        goto exit_kvm;
      };
      break;
    default: {
      bool ret;

      ret = kvm_cpu__handle_exit(cpu);
      if (!ret)
        goto panic_kvm;
      break;
    }
    }
    kvm_cpu__handle_coalesced_mmio(cpu);
  }

exit_kvm:
  return 0;

panic_kvm:
  return 1;
}
```

Appendix B

Tools

When asking for memory, linux might just hand out a reservation for memory, but nothing will be allocated until the memory is accessed. To disable this behavior echo 2 > /proc/sys/vm/`over-commit_memory`

```c
#include <stdlib.h>
#include <string.h>

int main()
{
  while(1) {
    void *m = malloc(1024*1024);
    memset(m,0,1024*1024);
  }
  return 0;
}
```

Fuzzing ports in the Guest OS.

```c
#include <sys/io.h>

int main ()
{
  int value;
  int address = 0;
  for(;;) {
    address++;
    value++;

    iopl(3);
    outl(value, address);
  }
  return 0;
}
```

Appendix C

CVEs of Linux Namespace

Taken from National Vulnerability Database (NVD)

- CVE-2015-1344

  Summary: The `do_write_pids` function in lxcfs.c in LXCFS before 0.12 does not properly check permissions, which allows local users to gain privileges by writing a pid to the tasks file. Published: 12/7/2015 3:59:01 PM

- CVE-2015-1342

  Summary: LXCFS before 0.12 does not properly enforce directory escapes, which might allow local users to gain privileges by (1) querying or (2) updating a cgroup. Published: 12/7/2015 3:59:00 PM

- CVE-2015-8222

  Summary: The lxd-unix.socket systemd unit file in the Ubuntu lxd package before 0.20-0ubuntu4.1 uses world-readable permissions for /var/lib/lxd/unix.socket, which allows local users to gain privileges via unspecified vectors. Published: 11/17/2015 10:59:24 AM

- CVE-2015-1335

  Summary: lxc-start in lxc before 1.0.8 and 1.1.x before 1.1.4 allows local container administrators to escape AppArmor confinement via a symlink attack on a (1) mount target or (2) bind mount source. Published: 10/1/2015 4:59:00 PM

- CVE-2015-1334

  Summary: attach.c in LXC 1.1.2 and earlier uses the proc filesystem in a container, which allows local container users to escape AppArmor or SELinux confinement by mounting a proc filesystem with a crafted (1) AppArmor profile or (2) SELinux label. Published: 8/12/2015 10:59:05 AM

- CVE-2015-1331

  Summary: lxclock.c in LXC 1.1.2 and earlier allows local users to create arbitrary files via a symlink attack on /run/lock/lxc/*. Published: 8/12/2015 10:59:03 AM

- CVE-2013-6456

  Summary: The LXC driver (lxc/`lxc_driver`.c) in libvirt 1.0.1 through 1.2.1 allows local users to (1) delete arbitrary host devices via the virDomainDeviceDetach API and a symlink attack on /dev in the container; (2) create arbitrary nodes (mknod) via the virDomainDeviceAttach API and a symlink attack on /dev in the container; and cause a denial of service (shutdown or reboot host OS) via the (3) virDomainShutdown or (4) virDomainReboot API and a symlink attack on /dev/initctl in the container, related to "paths under /proc/$PID/root" and the virInitctlSetRunLevel function. Published: 4/15/2014 7:55:08 PM

- CVE-2013-6441

  Summary: The lxc-sshd template (templates/**lxc-sshd**.in) in LXC before 1.0.0.beta2 uses read-write permissions when mounting /sbin/init, which allows local users to gain privileges by modifying the init file. Published: 2/14/2014 10:55:05 AM

- CVE-2013-6436

  Summary: The lxcDomainGetMemoryParameters method in `lxc/lxc_driver.c` in libvirt 1.0.5 through 1.2.0 does not properly check the status of LXC guests when reading memory tunables, which allows local users to cause a denial of service (NULL pointer dereference and libvirtd crash) via a guest in the shutdown status, as demonstrated by the "virsh memtune" command. Published: 1/7/2014 2:55:06 PM

- CVE-2011-4080

  Summary: The `sysrq_sysctl_handler` function in kernel/sysctl.c in the Linux kernel before 2.6.39 does not require the `CAP_SYS_ADMIN` capability to modify the `dmesg_restrict` value, which allows local users to bypass intended access restrictions and read the kernel ring buffer by leveraging root privileges, as demonstrated by a root user in a Linux Containers (aka LXC) environment. Published: 5/24/2012 7:55:02 PM

- CVE-2014-9717

  Summary: fs/namespace.c in the Linux kernel before 4.0.2 processes `MNT_DETACH` umount2 system calls without verifying that the `MNT_LOCKED` flag is unset, which allows local users to bypass intended access restrictions and navigate to filesystem locations beneath a mount by calling umount2 within a user namespace. Published: 5/2/2016 6:59:06 AM

- CVE-2015-2925

  Summary: The `prepend_path` function in fs/dcache.c in the Linux kernel before 4.2.4 does not properly handle rename actions inside a bind mount, which allows local users to bypass an intended container protection mechanism by renaming a directory, related to a "double-chroot attack." Published: 11/16/2015 6:59:00 AM

- CVE-2015-6927

  Summary: vzctl before 4.9.4 determines the Virtual Environment (VE) layout based on the presence of root.hdd/DiskDescriptor.xml in the VE private directory, which allows local simfs container (CT) root users to change the root password for arbitrary ploop containers, as demonstrated by a symlink attack on the ploop container root.hdd file and then access a control panel. Published: 9/28/2015 4:59:09 PM

- CVE-2012-2882

  Summary: FFmpeg, as used in Google Chrome before 22.0.1229.79, does not properly handle OGG containers, which allows remote attackers to cause a denial of service or possibly have unspecified other impact via unknown vectors, related to a "wild pointer" issue. Published: 9/26/2012 6:56:04 AM

- CVE-2007-2074

  Summary: Certain programs in containers in ScramDisk 4 Linux before 1.0-1 execute with SUID permissions, which allows local users to gain privileges via mounted containers. Published: 4/17/2007 11:19:00 PM

- CVE-2013-6456

  Summary: The LXC driver (lxc/`lxc_driver.c`) in libvirt 1.0.1 through 1.2.1 allows local users to (1) delete arbitrary host devices via the virDomainDeviceDettach Application Programming Interface (API) and a symlink attack on /dev in the container; (2) create arbitrary nodes (mknod) via the virDomainDeviceAttach API and a symlink attack on /dev in the container; and cause a denial of service (shutdown or reboot host OS) via the (3) virDomainShutdown or (4) virDomainReboot API and a symlink attack on /dev/initctl in the container, related to "paths under /proc/$PID/root" and the virInitctlSetRunLevel function. Published: 2014-04-15

Bibliography

[1] Arjan van de Ven. An introduction to clear containers, 2015. URL `https://lwn.net/Articles/644675/`.

[2] Intel. Clear containers. URL `https://clearlinux.org/`.

[3] Lixiang Yang. *The Art of Linux Kernel Design: Illustrating the Operating System Design Principle and Implementation.* Auerbach Publications, Boston, MA, USA, 1 edition, 2014. ISBN 1466518030, 9781466518032.

[4] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325384-058US. April 2016.

[5] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, June 2007. URL `http://linux-security.cn/ebooks/ols2007/OLS2007-Proceedings-V1.pdf`.

[6] Frank Swiderski and Window Snyder. *Threat Modeling.* Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619913.

[7] J. Di and S. Smith. A hardware threat modeling concept for trustable integrated circuits. In *Region 5 Technical Conference, 2007 IEEE*, pages 354–357, April 2007. doi: 10.1109/TPSD.2007.4380353.

[8] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. volume 45, pages 17:1–17:39, New York, NY, USA, March 2013. ACM. doi: 10.1145/2431211.2431216. URL `http://doi.acm.org/10.1145/2431211.2431216`.

[9] Y. Wen, J. Zhao, G. Zhao, H. Chen, and D. Wang. A survey of virtualization technologies focusing on untrusted code execution. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 378–383, July 2012. doi: 10.1109/IMIS.2012.92.

[10] Nelson Elhage. Virtunoid: A kvm guest -> host privilege escalation exploit. Black Hat, 2011. URL `https://github.com/nelhage/virtunoid`.

[11] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the*

2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), DSN '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-6471-3. doi: 10.1109/DSN.2013.6575349. URL `http://dx.doi.org/10.1109/DSN.2013.6575349`.

[12] Prashant Dewan, David Durham, Hormuzd Khosravi, Men Long, and Gayathri Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 828–835, San Diego, CA, USA, 2008. Society for Computer Simulation International. ISBN 1-56555-319-5. URL `http://dl.acm.org/citation.cfm?id=1400549.1400685`.

[13] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software attacks against Intel(R) VT-d technology. Invisible Things Lab, April 2011.

[14] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith. A new prime and probe cache side-channel attack for cloud computing. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/-DASC/PICOM), 2015 IEEE International Conference on*, pages 1718–1724, Oct 2015. doi: 10.1109/CIT/IUCC/DASC/PICOM.2015.259.

[15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, US, May 2015.

[16] US government. National vulneravility database. URL `https://nvd.nist.gov/`.