A Comparison of Kernel Memory Protection for Docker Containers Across Host Operating Systems

Li Dai
dali0802@go.stcloudstate.edu

A Comparison of Kernel Memory Protection for Docker Containers Across Host Operating Systems

by

Li Dai

A Thesis

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science in

Information Assurance

May 2020

Abstract

Object oriented programming concepts have been widely adopted by the modern design of enterprise applications, which relies on heap memory mapping, and re-use of pre-coded class libraries. Computing resource sharing such as containerization, is a popular way to effectively reduce operation overhead by enlarging the scale of kernel accessibility among distributed computer systems. Thus, proper isolation between processes, containers and host operating systems is a critical task to assure system wide information security. This is a study designed to compare kernel level memory management and protection effectiveness for Docker container systems maintained on top of Ubuntu Linux and Microsoft Windows as the host operating system. Literature research aims to study the fundamentals of kernel memory management designs, policies and modules in place for enforcement. As well as container architectures based on the variation of the host operating systems. The experimental design focuses on whether the discovery of unauthorized access is possible between containers, kernel spaces and file systems. Research results are targeted to determine a better approach for securing Docker container system implementations and code deployment.

Table of Contents

List of Tables

List of Figures

Chapter I: Introduction

Introduction

Traditionally, enterprise applications ran on their own operating system and at times, standalone physical servers. While the separation of software on specific hardware fulfills not only reliability, but also basic data security requirements, this method also generates a huge computing resource waste, especially when utilizing today's hardware capabilities ("IBM Cloud Education", 2019). Varieties of virtualization technologies have been researched and developed since the 1960s (Douglis & Krieger, 2013), to enable the possibility of multi-tenancy for applications running on singular hardware. Computing capital sharing among distributed systems is becoming more popular for many reasons such as, immense resource overhead, reducing high utility costs, and cloud adoption strategies just to name a few. Virtual zones, virtual machines, and containers are some of the ways that allow systems to utilize hardware resources more efficiently together.

With the greater effectiveness of computer hardware capacity distribution, such as with memory in a shared virtualized environment, security problems and threats that comes with it are sometimes overlooked (Zahedi, 2014). Typical virtual machines may use software-based memory virtualization to share memory resources from a physical host machine ("vSphere Documentation", n.d.). The Docker engine applies LXC-like namespaces to share and separate memory allocations for each container run on the host, and because of this design, its memory content protection relies heavily on the host operating system's kernel memory management operations. This thesis will review the current research in this discipline, create appropriate test-beds to compare the differences on how kernel memory management and protection is provided to Docker containers among various host operating systems, and conclude with results to show better security approaches when deciding on which host operating system to use for new Docker container system design and implementation.

Problem Statement

The greater computing resource sharing is employed, the more it can lead to a variety of security concerns and problems. In containerized systems, kernel resources like system memory, are often shared in between one another. A particular problem rises on how to properly separate the host operating system level shared kernel memory space. And to provide information assurance, in case of any individual container or host machine itself is compromised, while efficiently providing sufficient memory allocation spaces for the container engine daemon.

Nature and Significance of the Problem

Kernel memory has been an attack surface for many reasons. Advanced memory level attacks can cause significant service damage to applications or entire computing environments. Different base operating systems adopt diversified kernel level memory management techniques, therefor the attack prevention provided will also vary. Poorly designed or implemented memory protection mechanisms can directly lead to vulnerable systems and security threats.

Objective of the Research

This thesis will explore fundamental kernel memory management functions and policies, discover methodologies which different container host operating systems utilize, compare the security solutions provided to possible memory vulnerabilities and conclude with an answer for the potential secure combination while deploying with Docker containers using current technology.

Research Questions

1. How does kernel memory management work on Microsoft Windows or Ubuntu Linux?
2. Can a misused Docker container become a tool for unauthorized permission escalation? How do reactions differ between operating systems?
3. Are memory mappings of running Docker containers readable or writeable by sidecar containers? How do reactions differ between operating systems?
4. Can kernel memory management tools be misused by non-root users within a Docker container for hacking? How are kernel reactions different?

Definition of Terms

Kernel: Kernel is the core of a computer operating system software, which oversees every connection between software and hardware. It is one of the first portions to start up during boot and is the primary handler of all system resources. (Israeli & Feitelson, 2010)

Virtualization: According to VMware.com, it is: “the process of creating a software-based, or virtual, representation of something, such as virtual applications, servers, storage and networks.” (“Virtualization”, n.d.)

Docker Container: A lightweight execution environment developed by Docker, Inc. which provides shared host operating system kernel resources but isolates running processes.

Object-oriented programming: Program procedure formed by code, written based on the “object” concept, which is constructed by its properties (“Object-oriented programming”, n.d.)

Heap: Index of memory locations of objects for a program written with object-oriented language.

Summary

With the brief background information introduced, it is not difficult to see that a well-built memory management mechanism is fundamental for securing Docker containers, and the host operating system is one that provides the functionalities. Chapter II will provide an in-depth review of current research literature on this topic, seeking possible tool sets to build a test bed, answers or solutions to research questions defined, and discovering areas where more research contribution can be made.

Chapter II: Background and Review of Literature

Introduction

Current research literature is explored to better understand operating system fundamentals that make Docker container technology possible, its history and future, and what they provide for computing security prerequisites. As well as, how this research is designed methodology wise.

A considerable amount of literature review for this research is conducted with operating system documentation, such as referencing “The Linux Kernel v5.6.0-rc6” by “The kernel development community” at www.kernel.org and “Windows Kernel-Mode Driver Architecture” from Microsoft Dev Center at docs.microsoft.com. These kernel module documentations are primary sources to understand kernel functionalities and policies regarding their operating system principles.

Background and Current Research

Physical and Virtual Memory. Physical memory, that is, memory hardware actually installed in a computer, is an essential yet limited resource in traditional computing hardware design. Even though some of today’s technology agrees to hot-pluggable memory, there is always a hard boundary of the maximum size of memory one computer can expand to. (“The Linux Kernel”, n.d.) Random-access memory is a popular form factor of physical memory, it enables reading and writing data by using the same amount of time regardless of where data is physically located on a hardware chip, which shows significant speed advantage compared to limitations of direct-access memory / storage devices (Azimane, 2006). Therefore, physical memory is usually accessed by dedicated yet random assigned address ranges (page frames), frame size also depends on the implementation of hardware architectures. With how physical memory access works as explained, not much effort is necessary to see that direct interaction with physical memory is not an easy task for operating systems and application developers, and to avoid this complication virtual memory was developed. Physical memory is usually divided into pages, which are often to be sized to 4 Kbytes, but is dependent on architecture specifications. With virtual memory in place, every single memory access handle is given a virtual address. Multiple virtual memory pages are possible to be mapped to each single physical page frame and are structured with a hierarchical design. Memory management unit (MMU) is the hardware that passes all memory references through and translates a virtual memory address to physical memory address (Pichai & Hsu & Bhattacharjee, 2015). In modern hardware platforms, MMU is often integrated within the computer's central processing unit (CPU) on its critical processing path. The translation look-aside buffer (TLB) built in MMU caches freshly obtained page table entries (PTE), this reduces address lookup frequency. This way, memory paging is able to deliver a high-performance memory allocation and address translation (Gandhi & Karakostas & Ayar & Cristal & Hill & McKinley & Nemirovsky & Swift & Unsal, 2016). Virtual memory holds abstract data residing in physical memory, which allows only essential portions of application runtime data and shared objects' virtual address among processes in the physical memory space.

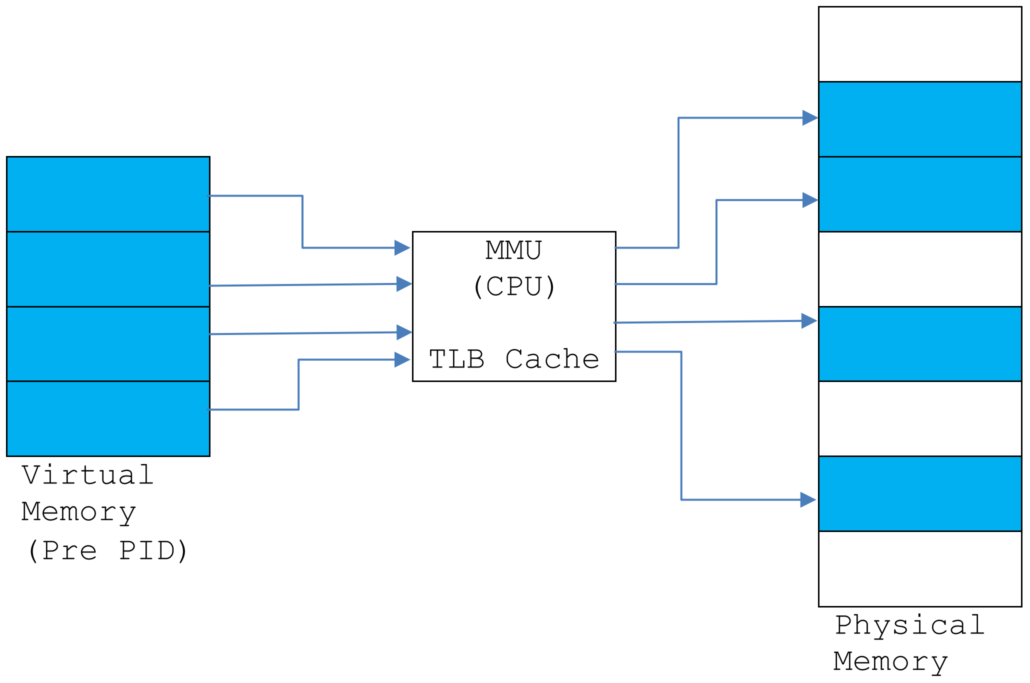


Figure 1. MMU – Memory address translation

Access, Control and Protection to Virtual Memory of Linux Kernel. Memory paging control and protection mechanisms are also implemented within virtual memory by operating system kernels, and are usually performed during kernel build time, by defining relevant kernel configurations. In the hierarchical design of virtual memory paging, higher level ones often contain physical addresses of pages belong to their immediate lower ones, the lowest table thus contains the physical address of actual pages utilized by a given application. A pointer of the top-level address table is entered into a register, when virtual addresses are translated by MMU, such register is then used to access the top-level address table. Since the physical addresses of lower level pages are indexed starting with the top-level downwards, the kernel is then able to access data pages in each layer. As physical memory is volatile memory, a typical way to ship data in and out of them relies on read or write of files on storage hardware, such as a hard drive, solid state drive and possible RAID arrays, which is rather slow in I/O speed compared to memory chips. To minimize this process, page caching is developed to gain adequate data transfer workflow. The size of cache at various level is inversely proportional to its speed. Memory pages are cached both ways regardless of reading or writing and are re-useable if the kernel detects and decides to. A synchronization function is built into the kernel module, which ensures updated data in page cache when it is to be reused.

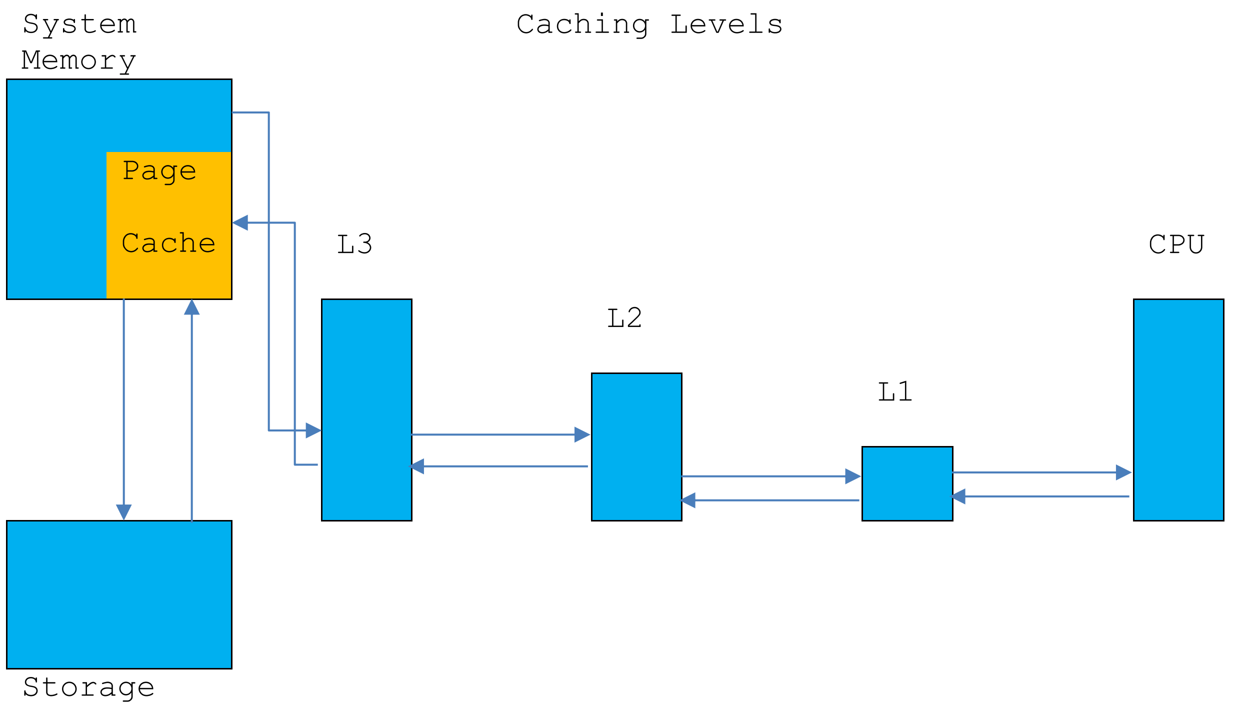


Figure 2. Page cache

Direct memory access (DMA) is a frequently used method to allow different controllers in a computer system to directly read or write to main system memory. DMA not only allows peripherals to communicate between various buses, but also avoid interaction with MMU, which in cases where MMU is integrated with the CPU, it saves CPU cycles (Markatos & Katevenis, 1997). Therefore, in practice, DMA helps lower CPU load and boosts overall system performance.

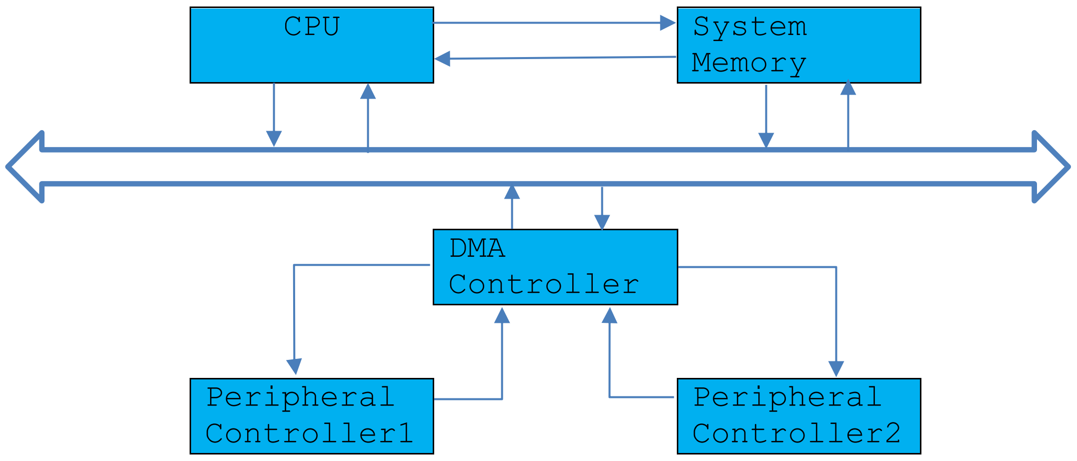


Figure 3. Direct Memory Access

Although memory page reuse among application processes or implementation of DMA provides performance gains, restrictions on the memory page address of a particular process or device can access, has been put in place for multiple security purposes. Thus, devices are not allowed to access all addressable memory pages on the same system. System kernel categorizes memory pages to targets zones, and aims to prevent accidental or unauthorized cross process, device memory access, as well as making sure the kernel itself has enough memory allocation available to perform essential tasks. Non-Uniform Memory Access (NUMA) is developed and has continuously been improved to assist multi-processor systems. It is designed to handle latencies caused by the distance between each processor or processor core and physical memory. Processor cores and banks of memory pages are paired into nodes, that then practices memory management policies and tasks independently. Table 1 below, shows a brief list of memory policies that are relevant to this research.

Table 1

Kernel Memory Policies relevant to this research

Name	Coverage	Scope
System Default	All page addresses	A government of all memory pages ensures overall system memory sufficiency and quality.
Task / Process	Optional/ Per-task	Similar usage as system default policy and applied only when individually defined by task.
VMA	Task specific VMA	Governs Virtual Memory Area of a specific task and ensures page allocation is explicit for such task.
Shared	Shared objects	Ensures memory objects shared between tasks are only available to specified ones, regulates above policies among shared memory area.

Many applications are written in a way that allocates all memory space it would possibly need upfront. It provides a good measure for application reliability from minimizing risk of running out of memory, but also creates a waste of memory resources because it only consumes all allocated memory in rare cases. The system kernel usually over-commits virtual memory compared to what it physically has, knowing this application behavior can offer more efficient use of memory. However, if some applications are under heavier load at the same time frame, the system can eventually run out of memory. (Chase, 2013) Kernel tool “Out-Of-Memory” (OOM) killer offers a way to ensure minimum memory is always available for operating system functionality by terminating other applications. Processes can be run with a dynamically assigned and adjustable oom_score, which is a ranking in case something needs to be stopped to release memory. The adjustability of this ranking could also give hackers a possible way to initiate a “Denial of Service” attack by shutting down production applications.

Memory Management – Microsoft Windows. The kernel memory management design of Microsoft Windows operating systems also utilizes virtual memory address spacing. According to kernel documents found at “Microsoft Dev Center”, in a traditional 32-bit architecture, each process within such a system is entitled to a maximum of 4 gigabytes of memory space, multi-threading capable code is allowed to share its memory data within all of associated processor threads, although access to virtual memory addresses of unrelated processes is prohibited to prevent memory corruption. Virtual memory address space is partitioned into higher and lower portions, default policy divides the useable memory space evenly, however, and there are available tuning options that a system administrator can enable for performance optimization. Table 2 illustrates memory space partitioning and tuning with a 4GB memory sample. Examples demonstrate a 32-bit system architecture, with limited memory allowance for process, however, a more modern 64-bit system is able to handle up to 8 terabytes of memory space below Windows version 8 and 128 terabytes currently starting from Windows 8.1.

Table 2

Windows memory space partitioning and tuning with a 4GB memory sample

Location	Address Range	Size	Usage	Tuning
Low	0x00000000 - 0x7FFFFFFF	2GB	Proc	None (Default)
High	0x80000000 - 0xFFFFFFFF	2GB	OS	
Low	0x00000000 - 0xBFFFFFFF	3GB	Proc	4-gigabyte tuning
High	0xC0000000 - 0xFFFFFFFF	1GB	OS	
Low	0x00000000 - Megabytes	2-3GB	Proc	Dynamic (/USERVA)
High	Megabytes+1 -0xFFFFFFFF	1-2GB	OS	

Starting from earlier versions of Windows, such as Windows XP, Microsoft has developed Data Execution Prevention (DEP) as a potential road blocker for memory buffer overflow attacks. DEP supports the system to assign pages of memory as non-executable, stopping any malicious code that resides there from being initiated (Stojanovski & Gusev & Gligoroski & Knapskog, 2007). When an application makes attempts to start code from any of the protected pages, such application will receive "STATUS_ACCESS_VIOLATION" returned and technically bring the application to halt. These pages include but are not limited to heap range, stack range or other designated memory pools. DEP is started during the operating system boot process and applies settings according to policy, system function "GetSystemDEPPolicy" and "SetProcessDEPPolicy" can be called from an application to check for current applied policies and make changes. By default, memory allocations for heap that is assigned via "malloc" or "HeapAlloc" functions are non-executable, therefore running code from the process heap is prevented.

When programming applications intended to run in Windows environments, the Dynamic Link Library (DLL) must be well known. Like most other libraries in programming languages, DLLs offer numerous common functionalities (Kari, 1993), thus it promotes code reuse, modularization and memory usage optimization. According to default Windows memory management policy, virtual memory address space allocated for a DLL is only accessible to the process which called such DLL. At times where multiple application processes are calling the same DLL, the virtual memory pages will be mapped to same physical memory pages, for sharing among all processes to start with.

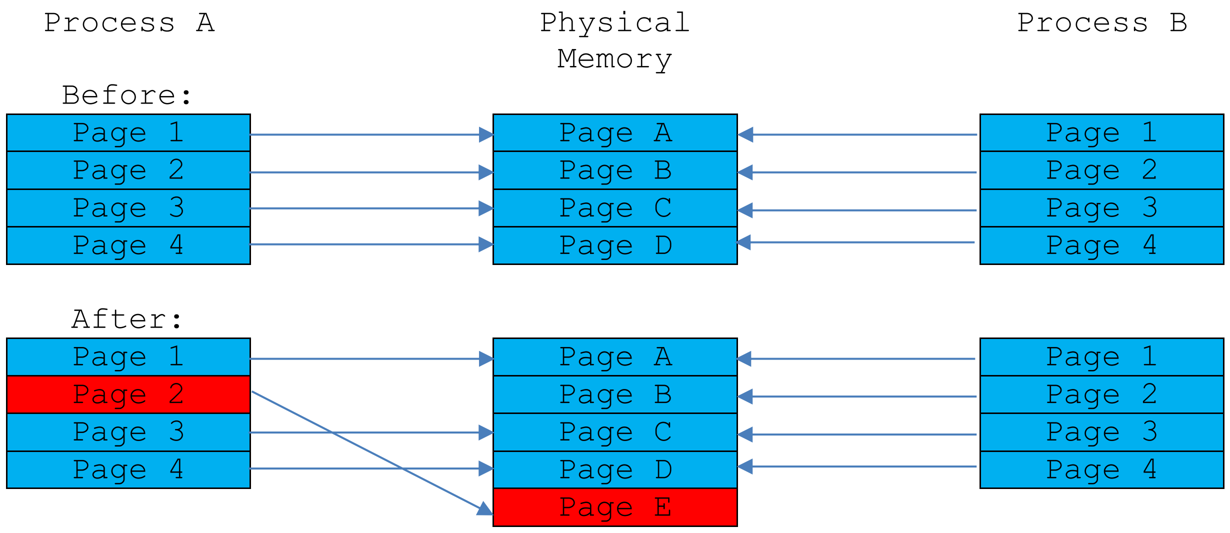


Figure 4. Copy-on-Write Protection

Once any of those processes are started to commit new data to the shared page, such page will then be re-allocated to a dedicated physical memory space. A kernel level protection "Copy-on-Write", updates those processes associated with a virtual memory address accordingly to make compliance of a no cross-process memory access policy.

This lazy evaluation technology makes more efficient physical memory use and saves MMU (CPU cycle) by reducing address translation until necessary. (“Microsoft Docs”, 2017)

Process, Heap and Hierarchical Design. When researching today’s software frameworks and memory related topics, Object-Oriented Programing (OOP) and the heap memory architecture is a hard-to- avoid area. Heap corruptions can cause software issues that easily become difficult to diagnose (Pravat & Hewardt, 2007). In common operating systems, each running piece of program, is assigned with a process ID (PID), which represents as the logical address of such code in memory (Bouffard & Lackner & Lanet & Johannes, 2015). The kernel as the core of the operating system manages these memory segments. Today, enterprise applications are often designed with an object-oriented framework (Mohamed & Douglas, 1997), and the concept goes back at least twenty years and has proven its advantages like reuse of code, well-structured programs and clear transition from design analysis to production implementation for software development (Guimaraes, 1995). A heap is mandatory for this structure, hierarchically at the top of allocated virtual memory space, and acts as an index of all objects which architects the program. Historically, computer software architectures took advantage of hierarchical design, which benefits effectively organizing and efficiently processing data. However, it does create a single point of attack surface at the top, which then potentially leads to catastrophic failure of the hierarchy (Dai & Guster & Rice, 2019). Therefore, to protect the heap from being tampered with, it is an important task included to ensure confidentiality, integrity and availability of such program.

Virtualization, “Containerization” and Cloud Enabling. Advanced computing with virtualization technology started in the era of mainframe machines, and has since been intensely researched and continuously developed by technology giants such as VMware, Oracle, Citrix and Microsoft (“Brief History of Virtualization”, 2012). The success of today’s cloud computing environments proves the benefits from effective virtualization technologies (Ferreira & Pedretti, Bridges & Brightwell & Fiala & Mueller, 2012). Moreover, virtualization platforms enable the possibility to install and run various types of operating systems (virtual machines) independently on top of one physical computer with a hypervisor like VMware vSphere or Microsoft Hyper-V, thus computing resources such as memory can be shared among them (“IBM Cloud Education”, 2019). While this methodology has its strengths, virtual machines also operate with a heavier overhead, especially if only a single dedicated service is intended to reside on the virtual machine which is an unnecessary drawback.

Containerization, an old bottled new wine, has its roots from the early days of Linux, which provides the ability to isolate running processes with shared Linux native kernel features (Osnat, 2018). Two main kernel features that are combined to a container are namespaces and cgroups. At the operating system level, namespaces control and separates system resources to process or process groups, basically isolating processes down to their own space (Evans, 2016). There are various available namespaces that come with a Linux installation. For example, PID namespaces, which assigns pid from 1 for the processes running inside; networking namespace, works with iptables and allows processes to have independent IPv4 or IPv6 addresses, ports or firewall rules from their host operating system. And under the same idea, user namespace creates users and groups with dedicated UID and GID, and mount namespace permits processes to mount or unmount its own filesystem. Creating names can be as easy as executing the command “unshare” with intended option flags. Figure 5 demonstrates the creation of a new PID namespace.

```
ldai@thesis:~$ ps -aux | grep bash
ldai       1689  0.0  0.0  21492  5096 pts/0    Ss   21:01
0:00 -bash
ldai       1701  0.0  0.0  21496  5068 pts/0    S    21:01
0:00 -bash
ldai       1808  0.0  0.0  13136  1036 pts/0    S+   21:28
0:00 grep --color=auto bash
ldai@thesis:~$ sudo unshare -f -p --mount-proc bash
[sudo] password for ldai:
root@thesis:~# ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START
TIME COMMAND
root          1  0.0  0.0  21276  4916 pts/0    S    21:22
0:00 bash
root          9  0.0  0.0  38376  3476 pts/0    R+   21:23
0:00 ps -aux
root@thesis:~# exit
exit
```

Figure 5. Creating of a PID namespace

In this case, new process “bash” inside the newly created PID namespace immediately started with PID 1 instead of 1689 from the host, and the user became root. With another terminal shell, the “nsenter” command can be used to access existing namespaces. Cgroups, an abbreviation of control groups, sets limitations to resources like memory and CPU time that one process can use. Control groups can be manually created with a command from the “cgroup-tools” package, called “cgcreate”. Figure 6 shows a sample memory control group setup.

```
Install cgcreate with: sudo apt install cgroup-tools
Create cgroups with 64mb memory limit:
```

```
ldai@thesis:~$ sudo cgcreate -a ldai -g memory:64mb
ldai@thesis:~$ ls -l /sys/fs/cgroup/memory/64mb/
ldai@thesis:/sys/fs/cgroup/memory/64mb$ cat
memory.limit_in_bytes
9223372036854771712
ldai@thesis:/sys/fs/cgroup/memory/64mb$ sudo echo 64000000 >
memory.limit_in_bytes
ldai@thesis:/sys/fs/cgroup/memory/64mb$ cat
memory.limit_in_bytes
64000000
```

```
Executing Java code which take 65Mbytes to run.
```

```
ldai@thesis:~$ sudo cgexec -g memory:64mb java memoryeater
Exception in thread "main" java.lang.OutOfMemoryError: Java
heap space
        at memoryeater.main(memoryeater.java:10)
```

Figure 6. Memory control group configuration

In 2013, Docker as a container platform provider, helped popularized the concept of containerization, and according to Docker over 3.5 million applications have been “containerized” with their Docker engine (“Docker eWeek”, n.d.). Figure 7 below describes a basic idea of how a Docker container differs from a regular virtualization platform. Docker as an open source container solution provider, offers unique answers to application level virtualizations, which makes applications run without dependencies to host operating systems and hardware configurations. In the article “Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems”, that Chen, Mao, Wang, Zhou, Zeldovich and Kaashoek (2011) explained that “Missing pointer checks”, “buffer overflow” and “Memory mismanagement” are typical kernel vulnerabilities, unauthorized or intended heap modifications will lead to failure of associated code runtime. Architectural security is rather challenging today, with the larger amount of resource sharing and scaling playing an important role in cloud computing environments (Manikandasaran & Raja, 2018).

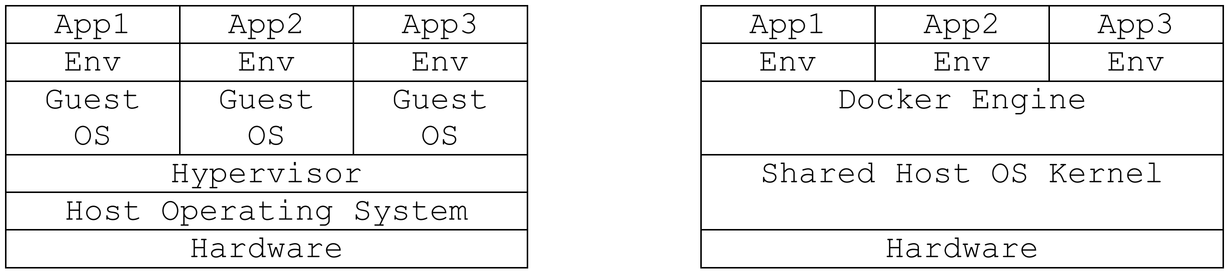


Figure 7. Virtualization & Containerization

Literature Related to the Methodology

Research Foundation. A previous research study by Dai, Guster and Rice (2019), stated that the key for tracking where objects reside is to reference their memory locations, which then leads to the heap. However, acquiring heap memory addresses of a running process in Linux only requires essential tools that comes with most Linux distributions. The research also identified that the heap of a non-root-user initiated process resides in user memory space and has permissions which allows a user level to read and write but is isolated from other processes with a “private” flag. Though a root level process would certainly be able to overwrite it. The article also stated that, after Java code is being packaged inside a Docker container, the memory locations were observed and not accessible with user level permissions anymore.

Though it added another layer of abstraction to protect the heap, the authors were able to clobber the heap process of the docker daemon to demonstrate a denial of service attack with little evidence left in log files. It will be interesting to discover how much damage, a Docker container which has mounted to host /proc directory, can do to host level user processes, root processes (docker daemon) or even filesystem.

Tools in Windows. In the operating system world of Microsoft Windows, there are also a number of native, as well as third-party debugging tools that are capable of searching through process data; such as for list related open files – Windows Process Explorer or to view memory or edit with custom values by – WinDbg (Microsoft Documentation, 2017). However, the way Docker containers operate in Windows is much different than how they do in Linux. For example, the Windows version of Docker engine, exposes system APIs through DLL files instead of Linux syscalls, and containers need at least some Windows kernel level DLLs to support operating system level API calls. And that means, the separation of application containers cannot be completely done away from system services and other DLL files, and it does not matter what language the containerized program is written in (Walker, 2018).

Summary

From system architecture to kernel memory management functionalities and policies, heap protection seems to be well thought out. Development and implementation of Docker containers is still strongly undergoing change, and current research literature relies more on basic functionalities of the operating systems themselves. However, with a similar set of tools offered in Windows as compared to

Linux, tests are ready to be done in a similar way to illustrate the possibilities of a denial of service attack initiated from a clobbered heap and compare both operating systems. Moreover, with these research conclusions and documentation of system level tooling, the purpose of this research and design of the methodology are clearly defined.

Chapter III: Methodology

Introduction

To provide accurate and understandable results of this comparison study, it is necessary to build, and test Docker container environments based on different host operating systems. The goal is to determine based on which host operating system a given Docker container engine is running, which containerized program has minimum to no impact from possible attack on the host. Various attacks will be simulated like a side channel attack, buffer overflow attack and direct content modifications all done on the memory level.

Design of the Study

In the literature review section, the research methodology introduced becomes a foundation of the design to this test bed. To accomplish the study goal, four testing scenarios are created, in consideration of multiple techniques to distribute possible negative impact to running code from a sidecar Docker container. One or more containers are pulled or configured for each stage. In Test 1, the container is built with Ubuntu Linux 18.04 base image and the “nano” program added on top, which offers access to a shell and text editor for potential required changes. Test 2 requires two containers, one built with an official OpenJDK base image, and wraps java code “stayrunning”, which is intended to keep providing a timestamp on stdout (screen) until interruption. The second container is built from Ubuntu Linux with debugging tools installed, in this case, it’s built with GUN Debugger. The container “stayrunning” and “test2” are re-used in Test 3, and a new container “memoryeater” is created based on

OpenJDK packaging a java code that runs and keeps consuming memory and aims to verify OOM practice of the system kernel. Lastly, a container with docker-cli interface will be created to have the basic ability to interact with a mounted host docker daemon socket. Within the host operating system, users ldai (uid 1000) and user (uid1001) are created, both users are added to group docker (gid 999) which has permission to execute commands to interact with docker daemon, however, only user ldai has sudo permission to run root level commands. Docker container instruction scripts and source code of java programs can be seen in Appendices A and B. In these test scenarios, the host operating system will be the changing variable, one being Ubuntu Linux, the other being Microsoft Windows 10, and the Docker engine version, Docker container packaged simple programs, and all other dependencies will remain the same.

Test-bed Scenarios and Purposes

Test 1: User ldai (uid 1000) has a plain text file stored in its home directory, with permission to read and write only by owner. User (uid 1001) tries to initiate unauthorized file system access of ldai's home directory by bind mounting "/home/ldai" into container "test1". This test checks the possibility of one getting elevated access with root permission via Docker container, which not only implies kernel namespace security effectiveness but also becomes an essential requirement proceeding to following tests.

Test 2: With container "stayrunning" activated by ldai (uid 1000), user (uid 1001) is normally not permitted to access memory mappings and heap segments of such code without being the owner of it or having root permissions. This test utilizes "test2" container, with flags turned on to bind mount "/proc" directory, PID namespace and privileges from the host system. This test checks if the system kernel prevents a user with elevated root access beyond PID namespace and overwrite memory segments of other processes.

Test 3: With container packaged java code “stayrunning” initiated in the background, user (uid 1001) executes containerized “memoryeater” code, which exhausts memory space by continuously consuming it, and tests will be conducted with smaller chunks and relatively larger pages for a comparison of kernel reaction. Container “test2” will also be used to gain access to “/proc” directory of “stayrunning” container process, tests are to be continued by overwrites to “oom_score_adj” with a larger number, and manually trick kernel to initiate OOM kill by passing “f” flag to file “/proc/sysrq_trigger”. This test verifies effectiveness of OOM kernel memory policy implementation and potential security threats.

Test 4: Again, with “stayrunning” container functioning, user (uid 1001) tries with container “test4” which has host docker daemon socket mounted and interacts with host docker daemon to stop other running containers. Proposal of container with mounted docker daemon socket might seem to be an unusual way to use container technology overall, but it is a proper method to test if host docker daemon can be controlled with a sidecar container, and how would kernel namespace prevent these activities? How are log files going to keep track of them?

The above tests are to be conducted on both operating system variations, targeting for a thorough comparison of reactions from kernel policies implementations and effectiveness of functionalities supporting them.

Data Collection

Data collection is crucial to this study, there is one data collection table designed for each test described above. Tables are aimed to accurately record test results and represent them in an easy-to-read fashion. The step-by-step test processes are to be recorded by shell command tables or screenshots, whichever applies better.

Hardware and Software Environment

Dell XPS 9360 – Specification:

CPU: Intel i7 8550U (1.80Ghz)

RAM: 16.0 GB (15.7 usable)

SSD: 512 GB M.2 NVMe

OS: Windows 10 (Version 1903)

Virtual Machine:

i. Ubuntu server: 18.04
ii. Microsoft Windows 10

Tools and Techniques

Ubuntu Linux. GDB: GNU Debugger, debugging tools which allows examination and modification to memory content of given running program in Linux.

Windows. Process Explorer: Displays basic information of a running process, such as PID and associated open DLL files.

RAMMap: Displays memory usage, priority information and physical range of given process ID.

WinDbg: Debugging tool for use in Windows, similar to GDB in Linux.

Summary

The test stages of this study are designed to step one on top of each other. The lower lever tests are essential for the next process. These tests should go through smoothly in a Linux environment. However, they are not guaranteed to work in all Windows environments, because some of the testing tools developed for Windows are developed with a Graphical User Interface, where Docker container for Windows runs in a command line interface or tends to be ran on a service like “head-less” mode.

Chapter IV: Data Presentation and Analysis

Introduction

In the “Data Presentation” section, a step-by-step testing process will be listed as in either shell command line records in tables, or screenshots of user interfaces. The following tables will summarize results of each test in a “Data Analysis” section.

Data Presentation

Ubuntu Linux Test-bed Configuration

```
ldai@thesis:~$ uname -a
Linux thesis 4.15.0-91-generic #92-Ubuntu SMP Fri Feb 28
11:09:48 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
```

Operating System Details

```
ldai@thesis:~$ id
uid=1000(ldai)gid=1000(ldai)groups=1000(ldai),4(adm),24(cdrom
),27(sudo),30(dip),46(plugdev),108(lxd),999(docker)
ldai@thesis:~$ id user
uid=1001(user)gid=1001(user)groups=1001(user),999(docker)
user@thesis:~/test2$ sudo cat
user is not in the sudoers file.  This incident will be
reported.
```

User Details

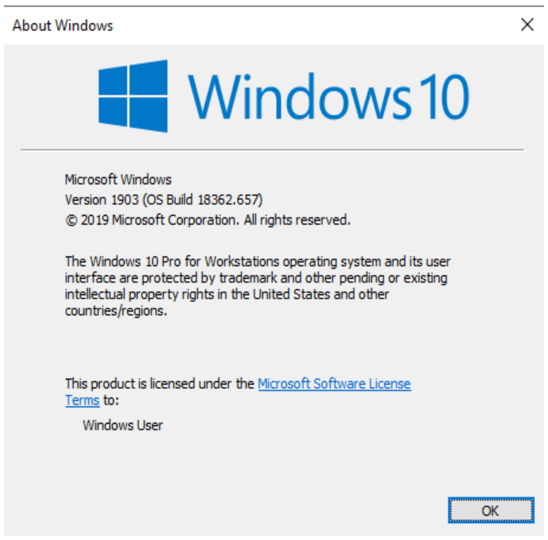
```
ldai@thesis:~$ ls -la aaa
-rw------- 1 ldai ldai 29 Mar 26 20:34 aaa
ldai@thesis:~$ cat aaa
There is only one line here.
```

Plain Text File owned by ldai with 600 permission

```
ldai@thesis:~$ docker -v
Docker version 19.03.5, build 633a0ea838
ldai@thesis:~$ docker-compose -v
docker-compose version 1.25.3, build d4d1b42b
```

Docker Engine Version

Microsoft Windows 10 Test-bed Configuration


About Windows

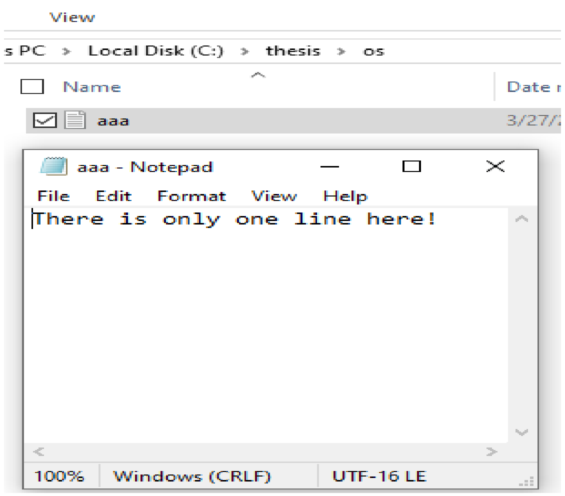
Microsoft Windows
Version 1903 (OS Build 18362.657)
© 2019 Microsoft Corporation. All rights reserved.

The Windows 10 Pro for Workstations operating system and its user interface are protected by trademark and other pending or existing intellectual property rights in the United States and other countries/regions.

This product is licensed under the Microsoft Software License Terms to:

Windows User

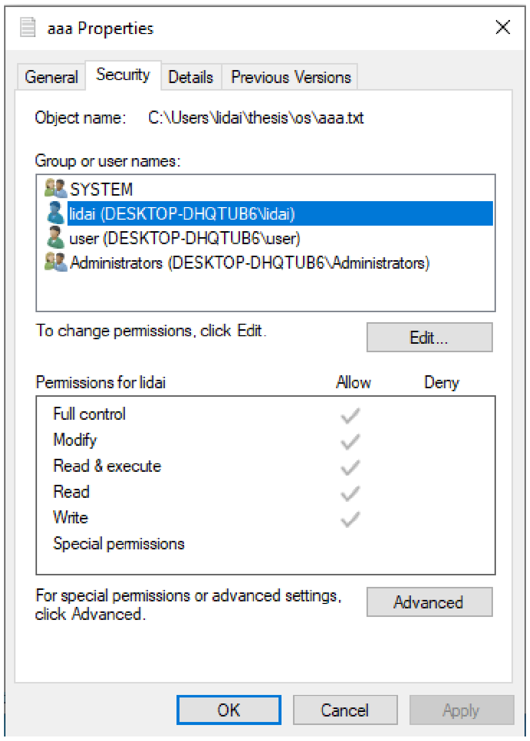
Windows Version Details


View
s PC > Local Disk (C:) > thesis > os
Name
Date
aaa
3/27/
aaa - Notepad
File Edit Format View Help

There is only one line here!

100%
Windows (CRLF)
UTF-16 LE

Content of “aaa.txt”


aaa Properties

Object name: C:\Users\lidai\thesis\os\aaa.txt

Group or user names:

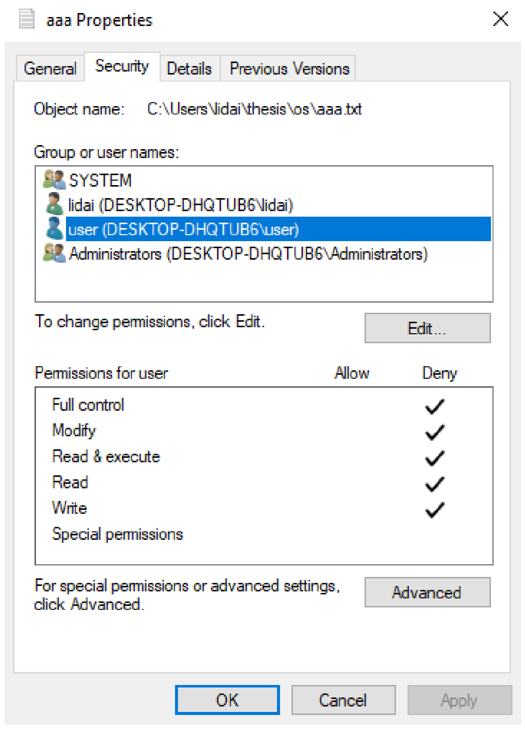
SYSTEM
lidai (DESKTOP-DHQTUB6\lidai)
user (DESKTOP-DHQTUB6\user)
Administrators (DESKTOP-DHQTUB6\Administrators)

To change permissions, click Edit.

Permissions for lidai	Allow	Deny
Full control	✓	
Modify	✓	
Read & execute	✓	
Read	✓	
Write	✓	
Special permissions		

For special permissions or advanced settings, click Advanced.

User lidai has full permission


aaa Properties

Object name: C:\Users\lidai\thesis\os\aaa.txt

Group or user names:

SYSTEM
lidai (DESKTOP-DHQTUB6\lidai)
user (DESKTOP-DHQTUB6\user)
Administrators (DESKTOP-DHQTUB6\Administrators)

To change permissions, click Edit.

Permissions for user	Allow	Deny
Full control		✓
Modify		✓
Read & execute		✓
Read		✓
Write		✓
Special permissions		

For special permissions or advanced settings, click Advanced.

User is no permission

```
PS C:\Users\lidai\thesis\os> docker -v
Docker version 19.03.5, build 633a0ea
```

Docker Engine Version

Test 1 – Ubuntu Linux

```
ldai@thesis:/$ su user
Password:
user@thesis:/$ ls -la /home
total 16
drwxr-xr-x  4 root root 4096 Feb  3 21:40 .
drwxr-xr-x 25 root root 4096 Mar 25 20:29 ..
drwxr-xr-x  6 ldai ldai 4096 Mar 26 20:34 ldai
drwxr-xr-x  5 user user 4096 Feb  9 16:33 user
user@thesis:/$ ls -la /home/ldai/aaa
-rw------- 1 ldai ldai 29 Mar 26 20:34 /home/ldai/aaa
user@thesis:/$ cat /home/ldai/aaa
cat: /home/ldai/aaa: Permission denied
```

Verify if other user (1001) can access ldai (1000)'s file

```
user@thesis:~$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
user@thesis:~$ docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
user@thesis:~$ docker build -t test1:latest .
Sending build context to Docker daemon  18.94kB
Step 1/2 : FROM ubuntu:18.04
18.04: Pulling from library/ubuntu
5bed26d33875: Pull complete
f11b29a9c730: Pull complete
930bda195c84: Pull complete
78bf9a5ad49e: Pull complete
Digest:
sha256:bec5a2727be7fff3d308193cfde3491f8fba1a2ba392b7546b43a0
51853a341d
Status: Downloaded newer image for ubuntu:18.04
 ---> 4e5021d210f6
Step 2/2 : RUN apt update && apt -y install nano
 ---> Running in 4892ac1b7b56
Removing intermediate container 4892ac1b7b56
 ---> f63075002cd4
Successfully built f63075002cd4
Successfully tagged test1:latest
user@thesis:~$ docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
test1               latest              f63075002cd4
56 seconds ago      93.5MB
ubuntu              18.04               4e5021d210f6        6
days ago           64.2MB
user@thesis:~$ docker run -it --mount
type=bind,source=/home/ldai,target=/test/ldai test1
root@1cd82de03972:/# cd /test/ldai
root@1cd82de03972:/test/ldai# ls -la aaa
-rw------- 1 1000 1000 29 Mar 27 01:34 aaa
root@1cd82de03972:/test/ldai# cat aaa
There is only one line here.
root@1cd82de03972:/test/ldai# echo 'NOW! There are TWO
lines!' >> aaa
root@1cd82de03972:/test/ldai# cat aaa
There is only one line here.
NOW! There are TWO lines!
root@1cd82de03972:/test/ldai# exit
exit
user@thesis:~$
```

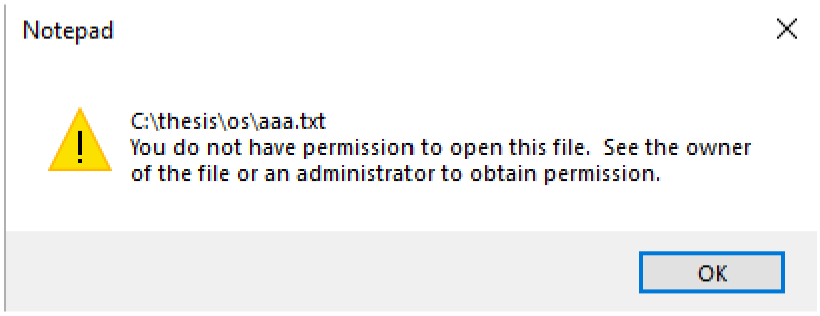
User (1001) successfully elevated permission then accessed and modified ldai (1000)’s file with a container.

```
user@thesis:~$ exit
exit
ldai@thesis:/$ cd
ldai@thesis:~$ cat aaa
There is only one line here.
NOW! There are TWO lines!
ldai@thesis:~$ stat aaa
  File: aaa
  Size: 55              Blocks: 8          IO Block: 4096
regular file
Device: 802h/2050d      Inode: 1572867     Links: 1
Access: (0600/-rw-------)  Uid: ( 1000/    ldai)   Gid: (
1000/    ldai)
Access: 2020-03-26 21:28:42.859029240 -0500
Modify: 2020-03-26 21:28:39.947011864 -0500
Change: 2020-03-26 21:28:39.947011864 -0500
```

```
 Birth: -
```

File aaa still has same metadata associated.

Test 1 – Microsoft Windows 10



Notepad

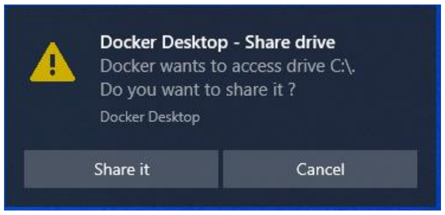
C:\thesis\os\aaa.txt
You do not have permission to open this file. See the owner of the file or an administrator to obtain permission.

Verify user cannot access file aaa.txt

```
PS C:\thesis\test1> docker build -t test1:latest .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM ubuntu:18.04
18.04: Pulling from library/ubuntu
5bed26d33875: Pull complete
f11b29a9c730: Pull complete
930bda195c84: Pull complete
78bf9a5ad49e: Pull complete
Digest:
sha256:bec5a2727be7fff3d308193cfde3491f8fba1a2ba392b7546b43a0
51853a341d
Status: Downloaded newer image for ubuntu:18.04
 ---> 4e5021d210f6
Step 2/2 : RUN apt-get update && apt-get -y install nano
 ---> Running in e559d10ea86d
Removing intermediate container e559d10ea86d
 ---> dc6be42f2de9
Successfully built dc6be42f2de9
Successfully tagged test1:latest
SECURITY WARNING: You are building a Docker image from
Windows against a non-Windows Docker host. All files and
directories added to build context will have '-rwxr-xr-x'
permissions. It is recommended to double check and reset
permissions for sensitive files and directories.
```

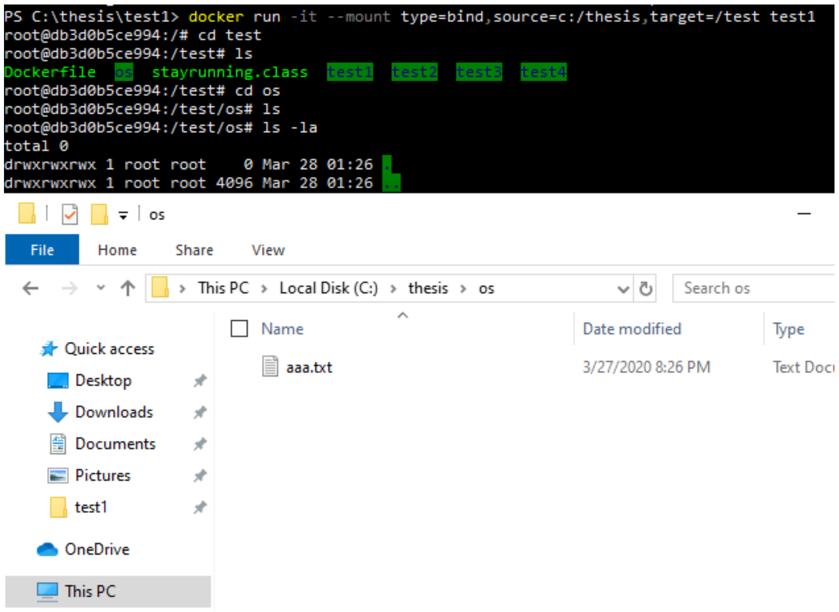
Building container “test1” (Notice above warning)

```
PS C:\thesis\test1> docker run -it --mount
type=bind,source=c:/thesis,target=/test test1
root@db3d0b5ce994:/# cd test
root@db3d0b5ce994:/test# ls
Dockerfile  os  stayrunning.class  test1  test2  test3  test4
root@db3d0b5ce994:/test# cd os
root@db3d0b5ce994:/test/os# ls
root@db3d0b5ce994:/test/os# ls -la
total 0
drwxrwxrwx 1 root root    0 Mar 28 01:26 .
drwxrwxrwx 1 root root 4096 Mar 28 01:26 ..
```



Docker Desktop - Share drive
Docker wants to access drive C:\.
Do you want to share it ?
Docker Desktop

“test1” container started after allowing mount by clicking “Share it”



```
PS C:\thesis\test1> docker run -it --mount type=bind,source=c:/thesis,target=/test test1
root@db3d0b5ce994:/# cd test
root@db3d0b5ce994:/test# ls
Dockerfile  os  stayrunning.class  test1  test2  test3  test4
root@db3d0b5ce994:/test# cd os
root@db3d0b5ce994:/test/os# ls
root@db3d0b5ce994:/test/os# ls -la
total 0
drwxrwxrwx 1 root root    0 Mar 28 01:26 .
drwxrwxrwx 1 root root 4096 Mar 28 01:26 ..
```

Name	Date modified	Type
aaa.txt	3/27/2020 8:26 PM	Text Doc

Root user inside container was not able to see “aaa.txt” but was able to see other files with access permission.

```
PS C:\thesis\test1> docker run mcr.microsoft.com/windows:1903
Unable to find image 'mcr.microsoft.com/windows:1903' locally
1903: Pulling from windows
```

```
af1a530dff54: Downloading [=====================>
]  1.517GB/3.657GB
c0f80931c4bb: Downloading
[==================================>                ]
1.623GB/2.367GB
Digest:
sha256:bbb680fb17fa5c93a95fcf97f6ea81bee4494ff405cf31f859686f
6ca9e761be
Status: Downloaded newer image for
mcr.microsoft.com/windows:1903
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\>
PS C:\thesis\test1>
```

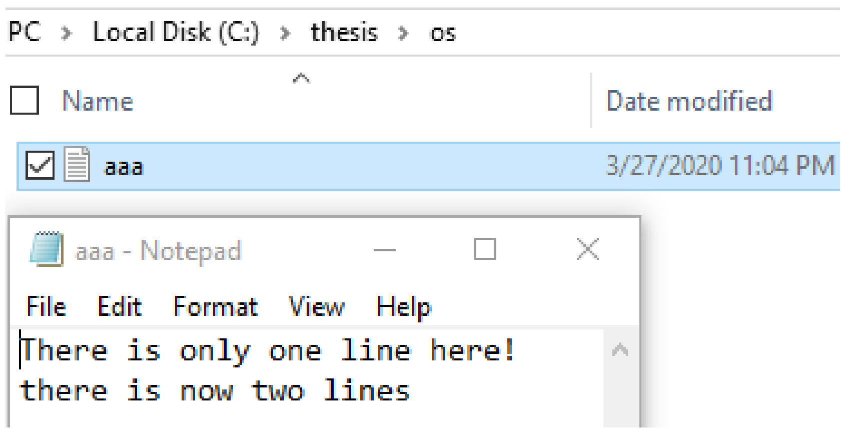
Getting "windows" container, because license issues, this study cannot run microsoft/nanoserver or Microsoft/servercore images.

```
PS C:\thesis\test1> docker run -it --mount
type=bind,source=c:\thesis,target=c:/test
mcr.microsoft.com/windows:1903 powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\> echo "there is now two lines" >> c:\test\os\aaa.txt
PS C:\>
```

Using powershell command to edit text file.



```
There is only one line here!
there is now two lines
```

File edited successfully.

Test 2 – Ubuntu Linux

```
ldai@thesis:~$ java stayrunning
This sample program should stay running ==> Thu Mar 26
21:48:35 CDT 2020
This sample program should stay running ==> Thu Mar 26
21:48:55 CDT 2020
This sample program should stay running ==> Thu Mar 26
21:49:15 CDT 2020
This sample program should stay running ==> Thu Mar 26
21:49:35 CDT 2020
This sample program should stay running ==> Thu Mar 26
21:49:55 CDT 2020
This sample program should stay running ==> Thu Mar 26
21:50:15 CDT 2020
```

Running Java code “stayrunning” on raw host by ldai (1000)

```
user@thesis:~/test2$ ps -aux | grep jaca
user      12051  0.0  0.0  13136  1108 pts/1    S+   21:57
0:00 grep --color=auto jaca
user@thesis:~/test2$ ps -aux | grep java
ldai      12002  0.3  0.5 4748916 41820 pts/0   Sl+  21:55
0:00 java stayrunning
user      12053  0.0  0.0  13136  1032 pts/1    S+   21:57
0:00 grep --color=auto java
user@thesis:~/test2$ cd /proc/12002
user@thesis:/proc/12002$ cat maps | grep heap
cat: maps: Permission denied
```

User (1001) does not have permission to view memory mapping of such running java code.

```
user@thesis:~/test2$ docker build -t test2:latest .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM ubuntu:18.04
 ---> 4e5021d210f6
Step 2/2 : RUN apt update && apt -y install libc6-dbg gdb
valgrind
 ---> Running in 98729dfbc0d2
Removing intermediate container 98729dfbc0d2
 ---> 1d2dbc59adc4
```

```
Successfully built 1d2dbc59adc4
Successfully tagged test2:latest
```

Building test2 container image with debugging tools installed.

```
user@thesis:~/test2$ docker run -it --mount
type=bind,source=/proc,target=/test/proc  test2
root@9bbd6b210161:/# cd /test/proc/12002
root@9bbd6b210161:/test/proc/12002# cat maps
cat: maps: Permission denied
root@9bbd6b210161:/test/proc/12002# cat pagemap
cat: pagemap: Permission denied
root@9bbd6b210161:/test/proc/12002# cat stack
cat: stack: Permission denied
root@9bbd6b210161:/test/proc/12002# exit
exit
```

Still not able to access virtual memory mappings, with elevated root permission.

```
ldai@thesis:~$ docker build -t stayrunning:latest .
Sending build context to Docker daemon    105kB
Step 1/3 : FROM openjdk:latest
latest: Pulling from library/openjdk
cd17e56c322c: Pull complete
ecdd73bb9922: Pull complete
e742458088f5: Pull complete
Digest:
sha256:85e34d6934d5b00048e31e93ee7abef73307cf0524d4205dcce4c9
b2a5870128
Status: Downloaded newer image for openjdk:latest
 ---> e2b050e4e3da
Step 2/3 : COPY ./stayrunning.class .
 ---> 5d0c5d201dea
Step 3/3 : CMD java stayrunning
 ---> Running in 6649376d29c7
Removing intermediate container 6649376d29c7
 ---> d8b148a44480
Successfully built d8b148a44480
Successfully tagged stayrunning:latest
ldai@thesis:~$ docker run -it stayrunning
This sample program should stay running ==> Fri Mar 27
03:26:40 GMT 2020
This sample program should stay running ==> Fri Mar 27
03:27:00 GMT 2020
```

```
This sample program should stay running ==> Fri Mar 27
03:27:20 GMT 2020
This sample program should stay running ==> Fri Mar 27
03:27:40 GMT 2020
```

Build docker container image and run java code “stayrunning”

```
user@thesis:~/test2$ ps -aux | grep java
root      19665  0.1  0.4 4783940 39024 pts/0   Ssl+ 22:44
0:04 java stayrunning
root      20721  0.2  0.4 4783940 38200 pts/0   Ssl+ 23:19
0:01 java stayrunning
user      21681  0.0  0.0  13136  1004 pts/1    S+   23:28
0:00 grep --color=auto java
user@thesis:~/test2$ docker run -it --mount
type=bind,source=/proc,target=/test/proc --pid=host --
privileged  test2
root@b0d26312a293:/# cd /test/proc/20721
root@b0d26312a293:/test/proc/20721# cat maps | grep heap
023a8000-023c9000 rw-p 00000000 00:00 0
[heap]
root@b0d26312a293:/test/proc/20721# gdb --pid 20721
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online
at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 20721
[New LWP 20767]
[New LWP 20776]
[New LWP 20777]
```

```
[New LWP 20778]
[New LWP 20779]
[New LWP 20780]
[New LWP 20781]
[New LWP 20782]
[New LWP 20783]
[New LWP 20784]
[New LWP 20785]
[New LWP 20786]
[New LWP 20787]
[New LWP 20788]
[New LWP 20795]
[New LWP 20796]
[New LWP 20797]

warning: Expected absolute pathname for libpthread in the
inferior, but got target:/lib64/libpthread.so.0.

warning: Unable to find libthread_db matching inferior's
thread library, thread debugging will not be available.

warning: Target and debugger are in different PID namespaces;
thread lists and other data are likely unreliable.  Connec

warning: Expected absolute pathname for libpthread in the
inferior, but got target:/lib64/libpthread.so.0.

warning: Unable to find libthread_db matching inferior's
thread library, thread debugging will not be available.
0x00007f8996170017 in pthread_join () from
target:/lib64/libpthread.so.0
(gdb) dump memory ~/gdbheap 0x023a8000 0x023a8f00
(gdb) set {char [3840]} 0x023a8000 = "The heap is now
clobbered!!!!"
(gdb) dump memory ~/gdbheap_2 0x023a8000 0x023a8f00
(gdb) detach
Detaching from program: target:/usr/java/openjdk-14/bin/java,
process 20721
(gdb) q
root@b0d26312a293:~# ls
gdbheap  gdbheap_2
root@b0d26312a293:~# cat gdbheap
```

```
!0▒:!P▒:p▒:!java!stayrunning1/usr/java/openjdk-
14/bin/java!P▒:p▒:1A/usr/java/openjdk-
14/lib/server/libjvm.soA/usr/java/openjdk-
14/lib/serverlibjvm.so▒▒▒!/lib64/libm.so.6!/lib64libm.so.6AP
▒:0z▒▒{▒P▒:0z▒▒{▒q▒:▒Y{▒▒T{▒P▒:0z▒▒{▒▒:▒Y{▒▒T{▒P▒:0z▒▒{
▒!▒▒!▒▒!▒▒!▒▒Q▒▒▒P
`▒:!@▒:▒▒▒▒:▒▒▒▒P▒:▒▒{▒▒▒:▒:▒▒▒▒▒▒▒▒▒▒▒▒h▒▒▒8▒▒▒▒▒▒ؼ▒▒躭
▒H▒▒▒(▒▒▒ڌ▒▒軭
▒₵▒▒Ⱥ▒▒▒▒▒▒▒▒▒(▒▒▒▒▒x▒▒▒h▒▒▒X▒▒▒H▒▒▒▒▒8▒▒▒X▒▒▒@▒▒▒▒▒%H▒:▒:▒:▒

P_▒P□▒D_▒A▒▒▒▒▒▒▒▒,□P▒:▒▒▒`E▒▒p▒▒{▒H▒l@▒H▒:eO)&▒▒▒▒▒▒▒▒▒â▒▒▒T
{▒Ӥ▒▒▒▒▒▒▒▒▒▒▒▒▒@=
▒:@n▒/usr/java/openjdk-14/lib/server/libjvm.so▒`r▒▒▒:▒m▒▒▒
▒▒▒:P▒:▒:▒m▒▒▒n▒▒▒n▒▒
n▒▒@n▒▒Pn▒▒▒n▒▒▒n▒▒▒n▒▒`n▒▒pn▒▒▒m▒▒▒m▒▒▒m▒▒▒n▒▒▒n▒▒▒m▒▒n▒▒▒m▒
▒n▒▒o▒▒0o▒▒ o▒▒o▒▒▒n▒▒Po▒▒@o▒▒0n▒▒@`r▒P▒r▒"▒▒:▒▒:P?:
0▒r▒ dr▒،:libm.so.6aK▒▒▒q▒N▒▒{▒H▒l`@▒:e`&Ë:{r▒P▒:{r▒p
▒▒▒▒                            ▒▒▒▒ui        ▒▒▒▒i▒▒▒ii
▒▒▒▒▒▒▒▒ui       ڗ▒▒w▒▒▒rii       ڗ▒▒
▒▒▒root@b0d26312a293:~# xxd gdbheap
bash: xxd: command not found
root@b0d26312a293:~# ls
gdbheap  gdbheap_2
root@b0d26312a293:~# cat gdbheap_2
The heap is now clobbered!!!!root@b0d26312a293:~#
```

User 1000 was able to attach host PID namespace and by using -–privileged flag, to access and overwrite given heap address.

```
This sample program should stay running ==> Fri Mar 27
04:37:46 GMT 2020
This sample program should stay running ==> Fri Mar 27
04:38:06 GMT 2020
This sample program should stay running ==> Fri Mar 27
04:38:26 GMT 2020
This sample program should stay running ==> Fri Mar 27
04:38:46 GMT 2020
This sample program should stay running ==> Fri Mar 27
04:39:06 GMT 2020
#
# A fatal error has been detected by the Java Runtime
Environment:
```

```
#
#  SIGSEGV (0xb) at pc=0x00007f89956e8cdf, pid=1, tid=23
#
# JRE version: OpenJDK Runtime Environment (14.0+36) (build
14+36-1461)
# Java VM: OpenJDK 64-Bit Server VM (14+36-1461, mixed mode,
sharing, tiered, compressed oops, g1 gc, linux-amd64)
# Problematic frame:
# C
[error occurred during error reporting (printing problematic
frame), id 0xb, SIGSEGV (0xb) at pc=0x00007f8995cd21c5]

# Core dump will be written. Default location: Core dumps may
be processed with "/usr/share/apport/apport %p %s %c %d %P
%E" (or dumping to //core.1)
#
# An error report file with more information is saved as:
# //hs_err_pid1.log
#
# If you would like to submit a bug report, please visit:
#   https://bugreport.java.com/bugreport/crash.jsp
#
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[error occurred during error reporting (), id 0xb, SIGSEGV
(0xb) at pc=0x00007f8995bccbb7]
[Too many errors, abort]
[Too many errors, abort]
[Too many errors, abort]
[Too many errors, abort]
[Too many errors, abort]
```

```
[Too many errors, abort]
[Too many errors, abort]
[Too many errors, abort]
 [Too many errors, abort]
```

Container packaged java code is no longer running.

Test 2: Microsoft Windows 10

```
PS C:\thesis> docker build -t stayrunning:latest .
Sending build context to Docker daemon  10.24kB
Step 1/3 : FROM openjdk:latest
 ---> 6adc576f6a58
Step 2/3 : COPY ./stayrunning.class .
 ---> Using cache
 ---> c3727279bb9f
Step 3/3 : CMD java stayrunning
 ---> Using cache
 ---> 6e23da6cbf17
Successfully built 6e23da6cbf17
Successfully tagged stayrunning:latest
SECURITY WARNING: You are building a Docker image from
Windows against a non-Windows Docker host. All files and
directories added to build context will have '-rwxr-xr-x'
permissions. It is recommended to double check and reset
permissions for sensitive files and directories.
PS C:\thesis> docker run -it stayrunning
This sample program should stay running ==> Sat Mar 28
15:51:05 GMT 2020
This sample program should stay running ==> Sat Mar 28
15:51:07 GMT 2020
This sample program should stay running ==> Sat Mar 28
15:51:09 GMT 2020
This sample program should stay running ==> Sat Mar 28
15:51:11 GMT 2020
This sample program should stay running ==> Sat Mar 28
15:51:13 GMT 2020
```

Built and started “stayrunning” container by lidai

```
PS C:\Users\user> docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED              STATUS              PORTS
NAMES
```

```
bed1fa8adbcd        stayrunning         "/bin/sh -c 'java
st…"   About a minute ago   Up About a minute
zealous_curie
PS C:\Users\user> Get-Process -Name vmwp
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI
ProcessName
-------  ------    -----      -----     ------     --  -- ---
--------
    370      18     6584      20412              8712   0
vmwp

PS C:\Users\user> Stop-Process -ID 8712
Stop-Process : Cannot stop process "vmwp (8712)" because of
the following error: Access is denied
At line:1 char:1
+ Stop-Process -ID 8712
+ ~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : CloseError:
(System.Diagnostics.Process (vmwp):Process) [Stop-Process],
ProcessCommandEx
   ception
    + FullyQualifiedErrorId :
CouldNotStopProcess,Microsoft.PowerShell.Commands.StopProcess
Command
```

User can list running docker containers process ID of “Virtual Machine Worker

Process” but cannot stop such container due to lack of permission.

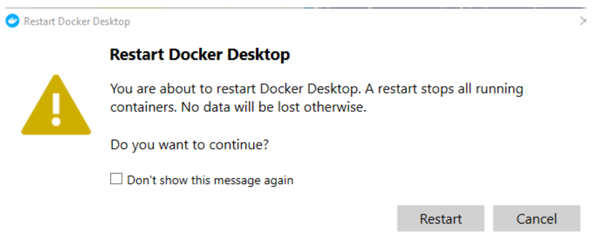
```
PS C:\Program Files\Docker\Docker> .\DockerCli.exe -
SwitchDaemon
 error during connect: Get
http://%2F%2F.%2Fpipe%2Fdocker_engine/v1.40/containers/json:
open //./pipe/docker_engine: The system cannot find the file
specified. In the default daemon configuration on Windows,
the docker client must be run elevated to connect. This error
may also indicate that the docker daemon is not running.
PS C:\Program Files\Docker\Docker> docker ps
error during connect: Get
http://%2F%2F.%2Fpipe%2Fdocker_engine/v1.40/containers/json:
open //./pipe/docker_engine: The system cannot find the file
specified. In the default daemon configuration on Windows,
the docker client must be run elevated to connect. This error
may also indicate that the docker daemon is not running.
```

```
PS C:\Program Files\Docker\Docker> PS C:\Users\user> docker
ps
Get-Process : A positional parameter cannot be found that
accepts argument 'docker'.
At line:1 char:1
+ PS C:\Users\user> docker ps
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-
Process], ParameterBindingException
    + FullyQualifiedErrorId :
PositionalParameterNotFound,Microsoft.PowerShell.Commands.Get
ProcessCommand
```

User tris to switch from Docker Linux container mode to Windows container mode via PowerShell command line interface but interrupted (crashed) docker daemon.

```
This sample program should stay running ==> Sat Mar 28
16:35:04 GMT 2020
This sample program should stay running ==> Sat Mar 28
16:35:06 GMT 2020
This sample program should stay running ==> Sat Mar 28
16:35:08 GMT 2020
This sample program should stay running ==> Sat Mar 28
16:35:10 GMT 2020
time="2020-03-28T11:35:12-05:00" level=error msg="error
waiting for container: unexpected EOF"
PS C:\thesis> docker run -it  stayrunning
C:\Program Files\Docker\Docker\resources\bin\docker.exe:
error during connect: Post
http://%2F%2F.%2Fpipe%2Fdocker_engine/v1.40/containers/create
: open //./pipe/docker_engine: The system cannot find the
file specified. In the default daemon configuration on
Windows, the docker client must be run elevated to connect.
This error may also indicate that the docker daemon is not
running.
See 'C:\Program Files\Docker\Docker\resources\bin\docker.exe
run --help'.
```

Running “stayrunning container” crashes and refuses to restart after daemon interruption.



Restart Docker Desktop

You are about to restart Docker Desktop. A restart stops all running containers. No data will be lost otherwise.

Do you want to continue?

☐ Don't show this message again

However, docker desktop application is still running

```
PS C:\thesis> docker run -it  stayrunning
This sample program should stay running ==> Sat Mar 28
17:06:53 GMT 2020
This sample program should stay running ==> Sat Mar 28
17:06:55 GMT 2020
This sample program should stay running ==> Sat Mar 28
17:06:57 GMT 2020
```

Restarted docker daemon and “stayrunning” container

```
PS C:\Users\user> docker ps
CONTAINER ID          IMAGE                 COMMAND
CREATED               STATUS                PORTS
NAMES
d624e2b49317          stayrunning           "/bin/sh -c 'java
st…"   3 minutes ago        Up 3 minutes
unruffled_blackburn
PS C:\Users\user> docker exec -it d624 /bin/sh
sh-4.2# ps
sh: ps: command not found
sh-4.2# cd /proc/22
sh-4.2# cat maps | grep heap
01d46000-01d88000 rw-p 00000000 00:00 0
[heap]
```

User is still able to find heap memory allocation information. But would not have enough tools around unless a Windows container switch can be done while the Linux container is running.

Test 3: Ubuntu Linux

```
ldai@thesis:~$ docker run -it stayrunning
This sample program should stay running ==> Fri Mar 27
04:53:16 GMT 2020
```

```
This sample program should stay running ==> Fri Mar 27
04:53:36 GMT 2020
This sample program should stay running ==> Fri Mar 27
04:53:56 GMT 2020
```

Start “stayrunning” container

```
user@thesis:~/test3$ docker build -t memoryeater:latest .
Sending build context to Docker daemon  4.096kB
Step 1/3 : FROM openjdk:latest
 ---> e2b050e4e3da
Step 2/3 : COPY ./memoryeater.class .
 ---> 8d27609c1c06
Step 3/3 : CMD java memoryeater
 ---> Running in ab989c51b079
Removing intermediate container ab989c51b079
 ---> 255067894aaf
Successfully built 255067894aaf
Successfully tagged memoryeater:latest
```

Build “memoryeater” container

```
user@thesis:~/test3$ docker run -it memoryeater
free memory: 67834240
free memory: 91081408
free memory: 90557120
free memory: 90557120
free memory: 221104896
free memory: 155568896
free memory: 90557184
free memory: 351141072
free memory: 285605072
free memory: 220593360
free memory: 480634248
free memory: 415622536
free memory: 350610824
free memory: 611175872
free memory: 545639872
free memory: 480628160
free memory: 741196656
free memory: 675660656
free memory: 610648944
free memory: 785241144
free memory: 719705144
free memory: 654693432
free memory: 590207696
free memory: 524671696
free memory: 459659984
free memory: 395167280
free memory: 329631280
free memory: 264619568
free memory: 200127696
free memory: 134591696
Exception in thread "main" java.lang.OutOfMemoryError: Java
heap space
        at memoryeater.main(memoryeater.java:10)
```

Above container is designed to take up 650 megabytes until memory exhausts.

```
user@thesis:~/test3$ docker build -t memoryeater:latest .
Sending build context to Docker daemon  4.096kB
Step 1/3 : FROM openjdk:latest
 ---> e2b050e4e3da
Step 2/3 : COPY ./memoryeater.class .
 ---> b23a94ff4fc1
Step 3/3 : CMD java memoryeater
 ---> Running in f105de607315
Removing intermediate container f105de607315
 ---> 84a7b0d75fa9
Successfully built 84a7b0d75fa9
Successfully tagged memoryeater:latest
user@thesis:~/test3$ docker run -it memoryeater
free memory: 721088880
Exception in thread "main" java.lang.OutOfMemoryError: Java
heap space
        at memoryeater.main(memoryeater.java:10)
```

Above container is modified to take up 1 gigabyte until memory exhausts.

```
This sample program should stay running ==> Fri Mar 27
05:07:36 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:07:56 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:08:16 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:08:36 GMT 2020
```

“Stayrunning” container is still running as intended.

```
user@thesis:~$ docker run -it --mount
type=bind,source=/proc,target=/test/proc --pid=host test2
root@b61bcf4e752d:/# ps -aux | grep java
root      19665  0.1  0.4 4783940 39356 pts/0   Ssl+ 03:44
0:08 java stayrunning
root      21920  0.1  0.4 4783940 38656 pts/0   Ssl+ 04:53
0:01 java stayrunning
root      22714  0.0  0.0  11464  1008 pts/0    S+   05:08
0:00 grep --color=auto java
root@b61bcf4e752d:/# cd /test/proc/21920
root@b61bcf4e752d:/test/proc/21920# cd ..
root@b61bcf4e752d:/test/proc# echo f > sysrq-trigger
```

“oom_score_adj” is overwritten to high number, and “f” flag overwritten to “sysrq-trigger” to manually trigger OOM Kill.

```
This sample program should stay running ==> Fri Mar 27
05:10:16 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:10:37 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:10:57 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:11:17 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:11:37 GMT 2020
This sample program should stay running ==> Fri Mar 27
05:11:57 GMT 2020
Killed
ldai@thesis:~$
```

Java code in “stayrunning” container is killed

Test 3: Microsoft Windows 10

```
This sample program should stay running ==> Sat Mar 28
17:33:13 GMT 2020
This sample program should stay running ==> Sat Mar 28
17:33:15 GMT 2020
This sample program should stay running ==> Sat Mar 28
17:33:17 GMT 2020
This sample program should stay running ==> Sat Mar 28
17:33:19 GMT 2020
```

Container “stayrunning” is up

```
PS C:\thesis\test3> docker build -t memoryeater:latest .
Sending build context to Docker daemon   5.12kB
Step 1/3 : FROM openjdk:latest
 ---> 6adc576f6a58
Step 2/3 : COPY ./memoryeater.class .
 ---> 8728ff2dbf13
Step 3/3 : CMD java memoryeater
 ---> Running in 039283c91ada
Removing intermediate container 039283c91ada
 ---> d68398122600
Successfully built d68398122600
Successfully tagged memoryeater:latest
SECURITY WARNING: You are building a Docker image from
Windows against a non-Windows Docker host. All files and
directories added to build context will have '-rwxr-xr-x'
permissions. It is recommended to double check and reset
permissions for sensitive files and directories.
```

Built “memoryeater” container

```
PS C:\thesis\test3> docker run -it memoryeater
Exception in thread "main" java.lang.OutOfMemoryError: Java
heap space
        at memoryeater.main(memoryeater.java:10)
```

Container “memoryeater” cannot be started due to lack of memory space for Java heap.

OOM kill test cannot be accomplished due to a lack of tooling without “Windows” container mode.

Test 4: Ubuntu Linux

```
ldai@thesis:~$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
ldai@thesis:~$ docker run -it stayrunning
This sample program should stay running ==> Fri Mar 27
21:08:55 GMT 2020
```

```
This sample program should stay running ==> Fri Mar 27
21:09:15 GMT 2020
This sample program should stay running ==> Fri Mar 27
21:09:35 GMT 2020
```

Container “stayrunning” started with no others running.

```
user@thesis:~/test4$ docker build -t test4:latest .
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu:18.04
 ---> 4e5021d210f6
Step 2/5 : RUN apt-get update     && apt-get -y install nano
apt-transport-https     ca-certificates     curl     gnupg-
agent     software-properties-common
 ---> Running in 3633cbc67518
Removing intermediate container 3633cbc67518
 ---> 1fd07e1a075d
Step 3/5 : RUN curl -fsSL
https://download.docker.com/linux/ubuntu/gpg | apt-key add -
 ---> Running in 44f16f014df4
Warning: apt-key output should not be parsed (stdout is not a
terminal)
OK
Removing intermediate container 44f16f014df4
 ---> 55ec00a945aa
Step 4/5 : RUN add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs)
stable"
 ---> Running in 824b3008755b
Removing intermediate container 824b3008755b
 ---> 6f47cae8997b
Step 5/5 : RUN apt-get update     && apt-get -y install
docker-ce docker-ce-cli containerd.io
 ---> Running in 437c2bb47874
Removing intermediate container 437c2bb47874
 ---> 932153a3db81
Successfully built 932153a3db81
Successfully tagged test4:latest
```

Building container “test4” with docker-cli installed

```
user@thesis:~/test4$ docker run -it --mount
type=bind,source=/var/run/,target=/var/run/ test4
root@f1372aa74231:/# docker ps
```

```
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
f1372aa74231        test4               "/bin/bash"
4 seconds ago       Up 2 seconds
dazzling_sutherland
8612a3174684        stayrunning         "/bin/sh -c 'java
st…"   8 minutes ago       Up 8 minutes
xenodochial_gould
root@f1372aa74231:/# docker kill 8612a
8612a
root@f1372aa74231:/#
```

Stopping “stayrunning” container with no error.

```
This sample program should stay running ==> Fri Mar 27
21:18:55 GMT 2020
This sample program should stay running ==> Fri Mar 27
21:19:15 GMT 2020
This sample program should stay running ==> Fri Mar 27
21:19:35 GMT 2020
ldai@thesis:~$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
f1372aa74231        test4               "/bin/bash"         3
minutes ago       Up 3 minutes
dazzling_sutherland
```

“Stayrunning” stopped without user ldai (1000)’s acknowledgement.

```
Mar 27 16:19:53 thesis dockerd[1186]: time="2020-03-
27T16:19:53.613022943-05:00" level=warning
msg="8612a3174684532f60991620bb84aa41f8523584c0de3f0faed6919a
11e4fd5c cleanup: failed to unmount IPC: umount
/var/lib/docker/containers/8612a3174684532f60991620bb84aa41
Mar 27 16:28:26 thesis dockerd[1186]: time="2020-03-
27T16:28:26.831996811-05:00" level=info msg="ignoring event"
module=libcontainerd namespace=moby topic=/tasks/delete
type="*events.TaskDelete"
```

One record found in docker daemon log about stopped container, nothing indicating which user has done so.

Test 4: Microsoft Windows 10

Test cannot be conducted due to lack of stability issue of Windows version docker daemon and “Windows” container mode.

Data Analysis

Table 3

Test 1 Data Summary

	Mounting Filesystem	Root Access	File Readable	File Writeable
Linux	Y	Y	Y	Y
Windows	Y	Y	Y	Y

Table 4

Test 2 Data Summary

	Mounting Filesystem	/proc/maps Access	Heap Readable	Heap Writeable	Negative Impact
Linux	Y	Y	Y	Y	Y
Windows	Y	Y	N/A	N/A	Y

Table 5

Test 3 Data Summary

	“memoryeater” starting	Kernel OOM Kill	Negative impact	Manual OOM Kill	Negative Impact
Linux	Y	Y	N	Y	Y
Windows	N	Y	N	N/A	N/A

Table 6

Test 4 Data Summary

	Mounting Host Docker Daemon	Host Docker Daemon Interaction	Negative impact	Log File Record
Linux	Y	Y	Y	N
Windows	N/A	N/A	N/A	N/A

Summary

This chapter covers a majority of the testing process which happened in both operating system environments. Attempted trials and detailed steps have been recorded, as accurately as possible, in the data collection section and a summary of test results are presented after the data analysis process.

Chapter V: Results, Conclusion, and Recommendations

Introduction

This chapter serves as an overview of the thesis research which has been done. A representation of research outcomes, an explanation of how such data reflects to the purpose of this thesis, and any feasible future works are described in the sections of Results, Conclusion and Future Work.

Results

From a high-level point of view, the overall methodology of this research started with learning from kernel documentation (i.e. manuals), with the knowledge of how kernel manages memory for applications. The tests were designed with a hierarchical mind set, each stage's steps move forward and practices kernel functions. Results obtained from the study are clear and answered the following research questions.

Q: How does kernel memory management work on Microsoft Windows or Ubuntu Linux?

A: Both kernel software researched by this thesis, showed which centralized memory management is key to provide policy enforcement consistency, where the kernel defines memory allocation policies based on system reliability and security, and kernel modules like malloc or mbrlenare are there to offer assistance for code developing, and Out-Of-Memory like functions are in place to enforce policy.

Q: Can a misused Docker container become a tool for unauthorized permission escalation? How do reactions differ between operating systems?

A: According to Test 1 in this study, user accounts with group access to docker daemon are capable of unauthorized permission escalation to root, in both operating system kernels, by simply starting a new Docker container. Such that a user essentially becomes root within their own namespace, but this permission is carried over to a host.

Q: Are memory mappings of running Docker containers readable or writeable by sidecar containers? How do reactions differ between operating systems?

A: As Test 2 has proven, that in a Linux kernel, mapping information of code in a current running container, is not accessible by a non-root user other than the processes they own unless such user has docker group permission to start a container within the same daemon. The heap memory allocation is readable and writable by unauthorized users in this way and can bring down running code. This activity technically grants that user root permission to at least the same namespace, which the docker daemon is hosting all its containers on by default. However, this test cannot be completely done with Microsoft Windows, because of instability of docker daemon when switching operation mode and lack of essential command line debugging tools for Windows PowerShell.

Q: Can kernel memory management tools be misused by non-root users within a Docker container for hacking? How are kernel reactions different?

A: Kernel functions such as OOM kill is in place to prevent malicious or poorly written code to overly allocate system memory, so foundational system can function stably.

Test 3 identifies that, OOM kill helped providing reliable memory resources to running code while new malicious program takes an unreasonable amount of memory. However, OOM kill function can be manually initiated to targeted processes if a regular user has elevated to root access, which is made available with Docker containers. Unfortunately, this test also cannot be conducted with Windows equipped virtual machines, again because of instability of its docker daemon.

Conclusion

In conclusion, this study learned that, even though, kernel software offers impressive central resource management and policies, and preventative modules are in place to ensure reliable operation, there are always vulnerabilities or security threats if tools are used for an inappropriate purpose. While virtualization and containerization accelerate effective and efficient computing resource sharing, information security and protection is still a valid concern for computer users, especially enterprise users who provide services or hold data for the general public.

In other words, it is never wrong for software or DevOps engineers to wait on product, infrastructure development, or adopting new technologies until they fully understand how it fundamentally operates. Always following good security practices while developing is another key to lowering risk and avoiding threats.

Future Work

Some of the designed tests could not be conducted during this research, primarily because of the lack of available tools and stability concerns. Some are well worth studying in the future if new tools can be developed to allow researchers to ascertain the desired results and these findings would promote good security practices and patching vulnerabilities.

References

Bouffard, G., Lackner, M., Lanet, J., & Loinig, J. (n.d.). *Heap . . . Hop! Heap Is Also Vulnerable*. *8968 2015*. https://doi.org/10.1007/978-3-319-16763-3_2

Brief History of Virtualization. (2012). In *Oracle.com*. Retrieved from https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html

Chase, R. (2013). IT Infrastructure, Technical Article, *Oracle*. *How to Configure the Linux Out-of-Memory Killer*. Retrieved from https://www.oracle.com/technical-resources/articles/it-infrastructure/dev-oom-killer.html

Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., & Kaashoek, M. F. (2011). Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems. *Asia-Pacific Workshop on Systems*. Shanghai, China.

Dai, L., Guster, D., & Rice, E. (2019). Protection Effectiveness and Vulnerabilities of the heap within Docker container systems. Poster presented at the *2019 Midwest Instruction and Computing Symposium*, Fargo, ND.

Douglis, F., & Krieger, O. (2013). Virtualization. *IEEE Internet Computing*, *17*(2), 6–9. https://doi.org/10.1109/MIC.2013.42

Evans, J. (2016). What even is a container: namespaces and cgroups. In *jvns.ca* Retrieved from: https://jvns.ca/blog/2016/10/10/what-even-is-a-container/

Fayad, M., & Schmidt, D. (1997). *Object-oriented application frameworks*. *40*(10), 32–38. https://doi.org/10.1145/262793.262798

Ferreira, K. B., Pedretti, K., Bridges, P. G., Brightwell, R., Fiala, D., & Mueller, F. (2012). Evaluating Operating System Vulnerability to Memory. *International Workshop on Runtime and Operating Systems for Supercomputers*.

Guimaraes, J. (1995). The Object Oriented Model and its Advantages. *ACM SIGPLAN OOPS Messenger, 40-49.*

IBM Cloud Education. *IBM Cloud Learn Hub. Virtualization*. (2019). Retrieved from: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide.

Israeli, A., & Feitelson, D. (2010). The Linux kernel as a case study in software evolution. *The Journal of Systems & Software, 83(3), 485–501.* https://doi.org/10.1016/j.jss.2009.09.042

Kari, R. (1993). Dynamic link libraries in Windows 3.x. *Journal of Chemical Education, 70(3), 248–.* https://doi.org/10.1021/ed070p248

Kerner, S. (2018). eWeek, Docker, Inc. *Docker Turns 5: A look at How the Technology Popularized Containers.* Retrieved from: https://www.docker.com/node/17907

M. Azimane, "High-Quality Memory Test," 2006 *IEEE International Workshop on Memory Technology, Design, and Testing* (MTDT'06), Taipei, 2006, *pp. xviii-xviii.*

Manikandasaran, S.S. & Raja, s. (2018). Secure Architecture for Virtual Machine to Container Migration in Cloud Computing. *Journal of Physics: Conference Series. 1142. 012017. 10.1088/1742-6596/1142/1/012017.*

Markatos, E. P. & Katevenis, M. G. H. (1997). User-level DMA without operating system kernel modification. *Proceedings Third International Symposium on High Performance Computer Architecture*, pp. 322- 331. Retrieved from: https://doi.org/10.1109/HPCA.1997.569696

Memory Management. (n.d.) *The Linux Kernel (5.6.0-rc7). Kernel.org.* Retrieved From: https://www.kernel.org/doc/html/latest/admin-guide/mm/index.html

Microsoft Docs, Hardware Dev Center. (2017). *Overview of Windows Memory Space.* Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-memory-space

Object-oriented programming. (n.d.), In *Wikipedia*. Retrieved from: https://en.wikipedia.org/wiki/Object-oriented_programming

Osnat, R. (2018). *A Brief History of Containers: From the 1970s to 2017.* Retrieved from https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

Pichai, B., Hsu, L., & Bhattacharjee, A. (2015). Address Translation for Throughput-Oriented Accelerators. *IEEE Micro, 35(3), 102–113.* https://doi.org/10.1109/MM.2015.44

Pravat, D.,Hewardt, M. (2007). *Advanced Windows Debugging: Memory Corruption Part II—Heaps. Inform IT.* Retrieved from: https://www.informit.com/articles/article.aspx?p=1081496&seqNum=2

Stojanovski, N., Gusev, M., Gligoroski, D., & Knapskog, S. (2007, April). *Bypassing Data Execution Prevention on MicrosoftWindows XP SP2. 1222–1226.* https://doi.org/10.1109/ARES.2007.54

Virtualization. (n.d.), In *VMware.com.* Retrieved from: https://www.vmware.com/solutions/virtualization.html

VMware, Inc. (2010). In *vSphere 5 Documentation*, Retrieved from: https://pubs.vmware.com/vsphere-50/topic/com.vmware.vcli.getstart.doc_50/cli_about.html

Windows Debugging tools. (n.d.) *Microsoft Documentation*s. Retrieved from: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/memory-window

Walker, J. (2018). *A short introduction to Windows Containers*. Retrieved from: https://medium.com/jettech/a-short-introduction-to-windows-containers-db5adc0db536

Zahedi, S. (2014). Virtualization Security Threat Forensic and Environment Safeguarding. At *2014-01-16, B3033, Universitetsplatsen 1, 352 52, Växjö,* Retrieved from: https://core.ac.uk/display/151389563

Appendix A: Dockerfile Source Code

```
FROM ubuntu:18.04
RUN apt-get update \
    && apt-get -y install nano
```

Container “Test1”

```
FROM openjdk:latest
COPY ./stayrunning.class .
CMD java stayrunning
```

Container “Stayrunning”

```
FROM ubuntu:18.04
RUN apt update \
    && apt -y install libc6-dbg gdb valgrind
```

Container “Test2”

```
FROM openjdk:latest
COPY ./memoryeater.class .
CMD java memoryeater
```

Container “memoryeater”

```
FROM ubuntu:18.04
RUN apt-get update \
    && apt-get -y install nano \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
RUN curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
apt-key add -
RUN add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubun$
RUN apt-get update \
    && apt-get -y install docker-ce docker-ce-cli
containerd.io
```

Container “Test4”

Appendix B: Java Program Source Code

```
import java.util.Calendar;
public class stayrunning {
        public static void main(String args[]) {
                stayrunning object = new stayrunning();
                object.waitMethod();
        }
        private synchronized void waitMethod() {
                while (true) {
                        System.out.println("This sample
program should stay running ==> " +
Calendar.getInstance().getTime());
                        try {
                                this.wait(20000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

stayrunning.java

```
import java.util.Vector;

public class memoryeater
{
  public static void main(String[] args)
  {
    Vector v = new Vector();
    while (true)
    {
      byte b[] = new byte[1073741824];
      v.add(b);
      Runtime rt = Runtime.getRuntime();
      System.out.println( "free memory: " + rt.freeMemory()
);
    }
  }
}
```

Memoryeater.java