# PROTECTION EFFECTIVENESS AND VULNERABILITIES OF THE HEAP WITHIN DOCKER CONTAINER SYSTEMS

Li Dai
Master of Science in
Information Assurance
St. Cloud State University
St. Cloud, MN 56301
dali0802@stcloudstate.edu

Dennis Guster & Erich Rice
Information Systems
St. Cloud State University
St. Cloud, MN 56301
dcguster@stcloudstate.edu
eprice@stcloudstate.edu

## ABSTRACT

Virtualization software is written using object-oriented code and contains a heap, which is an index of the memory locations of objects and instantiated as a process. The PID serves as the logical address of that code in memory. In LINUX a memory map is available for any process including the heap. If the heap is modified the associated running code will fail when it tries to reference an object but no longer has a pointer to its memory address. This paper will focus on the heap in both a virtual machine housed within a Docker container and of the Docker software itself. The goal was to create a test-bed and try to overwrite portions of each heap. To do this the process of finding the process ID, a memory map and the method to overwrite the heap was documented. The ramifications of hierarchical software design on modules was also addressed.

Li Dai, Dennis Guster & Erich Rice
Information Assurance & Information Systems
St. Cloud State University
St. Cloud, MN 56301
dali0802@stcloudstate.edu
dcguster@stcloudstate.edu
eprice@stcloudstate.edu

## INTRODUCTION

The effectiveness of virtualization to logically isolate hosts has been well proven within a cloud environment (Ferreira, K. B., Pedretti, K., Bridges, P. G., Brightwell, R., Fiala, D., & Mueller, F. (2012)). However, most virtualization software is written using an object-oriented form and thus contains a heap. The heap is basically an index of the memory locations of those objects used by the code that is instantiated as a process and the process ID ends up serving as the logical address of that code in memory (Bouffard, G., Lackner, M., Lanet, J.L., & Johannes, L. (2015)). Within the LINUX operating system a memory map can be found for any instantiated process within the /proc directory. Once that memory map is displayed the address of the heap is easily found. Much research has found that if the heap is modified then the associated running code will fail when it tries to reference an object but no longer has a pointer to its memory address (Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., & Kaashoek, M. F. (2011)).

This paper will focus on the heap in both a virtual machine housed within a Docker container and the heap of the Docker software itself. The goal was to create a test-bed and try to overwrite portions of each heap. To do this the process of finding the process ID, a memory map and the method to overwrite the heap will be documented.

This paper will also address the ramifications of using a hierarchical design. Historically, hierarchical design has permeated computer software. This makes sense in that it is an elegant means of organizing processing and data. However, it does create an interesting vulnerability in that in many cases corrupting the module at the top of the hierarchy will have detrimental effects on the modules under the module at the top. In this paper, it was interesting to see the results regarding the protection of the heap on the container manager. Of course, the goal is to provide another layer of abstraction and hence results in another layer that needs to be compromised. One would expect if the heap within a given container were compromised that it would only effect that container. Conversely, if the container manger heap gets compromised one might expect that not only the manager process would fail, but all containers housed under it would fail as well.

Object oriented programming (OOP) has proven to be an effective framework for software development and certainly provides numerous advantages to developers. This concept is not new and goes back at least twenty years. The concepts it provides include: reuse of code, better structured programs and an easier transition from analysis to implementation are well understood (Guimaraes, 1995). Perhaps it is the related concept of interoperability that has driven its acceptance. In a cloud architecture there is a wide diversity of applications which share information and the portability of objects becomes crucial (Aldrich, 2013).

The key to this portability is being able to keep track of where each object resides. This necessitates a table that references the object with its memory location. This process is the responsibility of the heap, which keeps track of where memory objects reside in the memory (Callum, Singer, & Vengerov, 2015). The openness of the LINUX operating system makes it easy to determine the memory of the heap. To gain insight into this process, examples will follow in which the attack is made against a simple VMware virtual machine running the LINUX operating system. In this example a java class entitled "stayrunning" is instantiated under process ID 27092. A memory map for this process is then displayed filtered by the "string heap". The resulting output reveals that the heap resides in the relative memory address range: 00c75000-00c96000. The difference in the address range reveals that the size of the heap is 21000x which translates to 135168 Bytes or 132 kB. This is a small fraction of the maximum heap size on this host with 1GB. The permissions on this segment are set to read, write and private (hidden from other classes within the package). By displaying the contents of the iomem file within the /proc directory, the expected memory segment resides in the System RAM, but not in the kernel area which would be in 0x01000000 range. Therefore, this memory segment is protected on the user level but not the kernel level. Furthermore, because the process is owned by a user entitled "guster" (userid: 16779226) one only needs that user's or higher rights to access the heap's memory segment. Besides gaining access through the operating system the jmap can be used within the java code to obtain access to the memory map provided by the heap. See code below:

```
guster@os:~$ java stayrunning

This sample program should stay running ==> Fri Feb 15 13:03:37 CST 2019

guster@os:~$ ps -al

F  S   UID        PID     PPID   C  PRI  NI ADDR SZ WCHAN  TTY TIME CMD

0  S 16779226  27092  26591  4  80    0 - 556841 -               pts/1  00:00:00 java

guster@os:~$ cd /proc/27092

guster@os:/proc/27092$ cat maps | grep heap

00c75000-00c96000 rw-p 00000000 00:00 0       [heap]

guster@os:/proc/27092$ java -XX:+PrintFlagsFinal -version | grep MaxHeapSize

uintx MaxHeapSize                    := 1040187392      {product}

guster@os:/proc$ cat iomem
```

```
00100000-bfeeffff : System RAM
```

```
dguster@eros:~$ jmap 27092

Attaching to process ID 27092, please wait...

Debugger attached successfully.

Server compiler detected.

JVM version is 24.151-b01

0x0000000000400000      6K      /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java

0x00007f369430a000      437K    /lib/x86_64-linux-gnu/libpcre.so.3.13.1

0x00007f3694578000      34K     /usr/lib/x86_64-linux-gnu/libffi.so.6.0.2
```

The memory segment containing the heap is important for a number of reasons. First, it is a very small segment within all available memory and finding it randomly would be quite difficult. Second, as the iomem file indicates, it is not loaded into the kernel space but rather into the user space. This means that the user that owns the process generating the task can view and modify the heap, though certainly the root could do this as well (Linux Memory Management, 2016). This means that a hacker with user level access to an application could easily contaminate the heap which could halt the process. The example below demonstrates how the gnu debugger could be used to extract the contents of the heap for process ID 27092 and how objects are linked to a relative memory address. Further, once the address and the appropriate user level rights are obtained then the "set" command in the debugger could be used to over write a memory address which will disrupt the running process. See below that the new value in the heap is "The heap is now clobbered!!!!". When the running code next references the heap a fatal error is observed which stops the process.

```
guster@os:/proc/27092$ gdb --pid 27092

(gdb) dump memory ~/gbpheap 0x00c75000 0x00c75f00

guster@os:~$ ls -l
-rw-r--r--  1 dguster domain users  3840 Feb 15 07:54 gbpheap
```

```
guster@os:~$ xxd gbpheap | more
00000000: 0000 0000 0000 0000 5102 0000 0000 0000  ........Q.......
00000280: 2f75 7372 2f6c 6962 2f6a 766d 2f6a 6176  /usr/lib/jvm/jav
00000290: 612d 372d 6f70 656e 6a64 6b2d 616d 6436  a-7-openjdk-amd6
000002a0: 342f 6a72 652f 6269 6e2f 6a61 7661 0000  4/jre/bin/java..
000002b0: 0000 0000 0000 0000 3102 0000 0000 0000

(gdb) set {char [3840]} 0x00c75000 = "The heap is now clobbered!!!!"
(gdb) dump memory ~/gbpheap2 0x00c75000 0x00c75f00

guster@os:~$ xxd gbpheap2 | more
00000000: 5468 6520 6865 6170 2069 7320 6e6f 7720  The heap is now
00000010: 636c 6f62 6265 7265 6421 2121 2100 0000  clobbered!!!!...
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ...............

guster@os:~$ java stayrunning
This sample program should stay running ==> Fri Feb 15 13:03:37 CST 2019
This sample program should stay running ==> Fri Feb 15 13:03:47 CST 2019

This sample program should stay running ==> Fri Feb 15 14:37:33 CST 2019
#
# A fatal error has been detected by the Java Runtime Environment:
#
#
[error occurred during error reporting (printing exception/signal name), id 0xb]

, pid=27092, tid=139872320878336
```

The scenario above illustrates that it is easy to find the heap as well as the objects that it is mapping for any given process. What is depicted is just a small fraction of what is potentially available. For more information about how this is related to java code see: *The JMAP Utility* (Utility, 2016). Beyond potential denial of service attacks there are many performance related and logical attack scenarios that can be launched against the heap (Barbu, Thiebeauld, & Guerin, 2010); (Barbu, Hoogvorst, & Duc, 2012). The extent of this potential vulnerability is described in (Drake, 2011). In this work the author found that the "java runtime environment (JRE) was installed on 89% of end-user computer systems. Unfortunately, JRE is plagued by a long history of security problems, including vulnerabilities in its components built from native code (CVE, 2016). Based on this trend, it is probable that many more vulnerabilities remain to be found." Based on this data additional research related to securing the heap is needed. Therefore the purpose of this paper is to investigate the effectiveness of the Docker container software of protecting the heap of a running java runtime process.

## REVIEW OF LITERATURE

Virtualization technologies have been rapidly developing over the past two decades, although its inception historically began in the era of mainframe computing from the late 1960s into the 1970s (Brief History of Virtualization, 2012). The use of virtualization technologies based on the x86 architecture has greatly accelerated the concept, with companies such as VMware, Oracle, and Microsoft refining and expanding their offerings in this area. The main espoused benefits to utilizing virtualization technologies have included reduced capital and operating costs due to the reduced need to buy and maintain expensive hardware, increased IT productivity and efficiency, faster provisioning of servers and other computing resources, as well as greatly simplified management of computing and data resources (What is Virtualization?, 2019). These benefits have led to the advent and wide adoption of cloud computing platforms such as the Amazon Web Service and Microsoft Azure clouds, where virtualization technologies play a critical role in the managing and provisioning of resources.

While virtualization platforms such as VMware and other hypervisors allow one to logically create multiple virtual machines on one piece of "bare metal" physical hardware, there are some limitations or at the very least considerations that come into play with its use (What is Virtualization?, 2019). The physical server hardware requires an operating system such as LINUX, UNIX, or a Windows Server variant on top of which the virtualization hypervisor software can run, this then allows the creation and deployment of VM's or virtual machines which can be tasked with running multiple services. The one drawback to this setup is that when a VM is created it requires a guest operating system to be placed onto it this is needed to manage the various application software which run the services. This adds an extra layer of system overhead as well as potentially cost (for instance if a Windows Server OS is used) and potentially security risk as well through side channel attacks (Rama Krishna, S. & Padmaja Rani, B., 2017).

This added OS layer is potentially unnecessary, especially if the intent is to only run one dedicated application and segregate it from other applications. For this type of a set up a computing container will often times make more sense. Containerization, much like virtualization, has its roots starting decades ago with the UNIX operating system version 7, which started the concept of process isolation (Osnat, 2018). The first early forms of a containerized application were being developed in the early 2000's, the Solaris Containers were released as a beta version in 2004 and allowed for separation by zones and even allowed for features such as taking snapshots or cloning containers much like what is commonly allowed in various virtualization platforms such as VMware (Osnat, 2018). Various types of containers were continued to be developed through the first decade of the 2000's, although not until the release of the Docker container platform in 2013 did containerization really begin to get widespread market acceptance. The use of containers with application developers grew exponentially, although potential security issues such as the "Dirty COW" vulnerability shed light on the underlying security risks that the new technology presented (Dulgie, 2016). Even so, the use of containers such as Docker and Kubernetes continue to make inroads, and has been wholeheartedly embraced by major cloud computing providers such as Amazon and Microsoft.

## METHODOLOGY

In the introductory section a scenario was presented in which a successful attack was launched against the heap of a java class hosted by a standard virtual machine. Much of the ease of launching this attack can be traced to the design of the LINUX operating system which provides a developer excellent functionality. While a LINUX operating system can be made very secure, to do so requires overcoming this functionality by hardening it. Specifically, in this case the fact that excellent memory maps for a given process can be obtained with just user level privileges means that hacker can zero in on the vulnerable memory segment. Further, with user level rights and the known memory address it is a simple process to overwrite the vulnerable segment which creates a fatal error and the code aborts. This would be especially problematic in a user devised service which would be constantly running and linked to some transport layer port. So ultimately, a strategy is needed that uses memory segments that are not mapped and have protection above simply the user level. The purpose of this paper is to examine the Docker container software and assess how well it protects the heap in a java socket call class. To do so it is important to evaluate the processes created when a java socket call class is instantiated into the system. In the example below a java class entitled "tempserver" is used to illustrate the scenario. This java class simply performs temperature conversions. To start this class a "docker run" command is entered below and the "Waiting for connection…" message indicates that it is running normally.

```
[dguster@localhost ~]$ sudo docker run tempserver
[sudo] password for dguster:
Waiting for connection...
```

However, in the example provided in the introductory section the java runtime and class are called from a directory, but in this example a preexisting image is called. The "docker images" command shows that two images have been created and are available. These images are the tempserver class and its corresponding client class called tempclient.

```
[dguster@localhost ~]$ sudo docker images
[sudo] password for dguster:

REPOSITORY    TAG      IMAGE ID        CREATED        SIZE
tempclient    latest   996eddd76286    41 hours ago   821MB
```

```
tempserver          latest        73bf19a061d2         41 hours ago      821MB
```

The “ps” command that follows shows that the image appears to be implemented as a shell and is running on port 12400. This shell then serves to isolate the code and can be considered as an independent container because it has its own process ID.

```
[dguster@localhost ~]$ sudo docker ps -a

CONTAINER ID   IMAGE     COMMAND     CREATED    STATUS   PORTS
30eb22372fe0     tempserver  "/bin/sh 'java Te"   27 hrs ago   Up 27hr   12400/tc
```

Looking deeper into the process structure with the “ps” command provides more insight. The Docker system itself is running under process ID 10942, whereas the tempserver image is running under process ID 10982. The java code itself is running under process ID 11013. This situation creates two levels of abstraction: the image to the docker process and the java runtime to the image itself. From a security point of view this results in two levels of isolation which improves the overall encapsulation strategy. Further, it is worth noting that both the image and the run time contained therein are owned by the root. This basically eliminates the possibility of user owned services. The next question is, does this organizational structure protect the memory segments containing the code.

```
[dguster@localhost ~]$ ps -al
F  S  UID   PID     PPID   C PRI  NI  ADDR SZ  WCHAN  TTY   TIME     CMD
4  S    0    10938   10919   0  80     0 - 60284        poll_s     pts/1  00:00:00  sudo
4  S    0    10942   10938   0  80     0 -  64588        futex_      pts/1  00:00:00 docker

[dguster@localhost ~]$ ps -aux | grep java
root     10982  0.0  0.0   4280   380 ?        Ss   09:40   0:00 /bin/sh -c java
TempServer
root     11013  0.2  1.3 2537756 25888 ?       Sl   09:40   0:00 java TempServer
```

In the example below process ID for the tempserver container is 15989. An attempt is made to find the memory location of the heap with only user level rights and permission is denied. Then root level access is attempted via the sudo command and a memory address of 0x5577e0a81000 is obtained for the heap. Note the memory protection flags are rwp.

```
[dguster@localhost ~]$ cd /proc/15989
[dguster@localhost 15989]$ cat maps | grep heap
cat: maps: Permission denied
[dguster@localhost 15989]$ sudo cat maps | grep heap
```

```
[sudo] password for dguster:
5577e0a81000-5577e0aa2000 rw-p 00000000 00:00 0  [heap]
```

Next an attempt is made from a debugger to dump the first 8192 Bytes to a file called dddd.

```
(gdb) dump memory /home/dguster/dddd 0x5633e4d85000 0x5633e4d87000
```

On the surface it appears that that heap data was copied into the file. However, a dump of the file reveals that the file "dddd" contains nothing but null values. So the container appear to protect the contents of the heap from unauthorized reads.

```
[dguster@localhost ~]$ ls -l
-rw-r--r--. 1 root    root    8192 Feb 26 10:08 dddd

[dguster@localhost ~]$ od -w1 -c -Ad dddd
0000000  \0
*
0008192
```

When trying to overwrite the first 3840 Bytes of the heap using a debugger a "cannot access memory" message was returned which again appears to confirm the effectiveness of the container software. A look at the memory addresses cited is interesting. If one looks at the "cannot access memory" message the 8 hex character address is the last 8 hex characters of the full 16 hex character address contained in the set command (note 4 leading 0's suppressed). As one would expect the memory used within the container is masked so as to protect its contents. So the sub-address has a different meaning within the container software than the host itself. A quick look at the host level memory map reveals that the sub-address is not actually mapped for specific purpose. Further, it was observed that each time the code was run different memory address ranges were used.

```
(gdb) set {char [3840]} 0x5577e0a81000 = "gggggggggggggggggggggggggggg"
Cannot access memory at address 0xe0a81000

[dguster@localhost proc]$ cat iomem
80208000-8020bfff : 0000:00:10.0
e5c00000-e5cfffff : PCI Bus 0000:22
```

However, a couple of abnormalities were observed. Occasionally, the attempt to write would result in a "broken pipe" error. A pipe is a data transfer mechanism often used to support inter-process communication. While the data itself was not compromised, the result was an effective denial of service.

```
(gdb) set {char [3840]} 0x5633e4d85000 =
"gggggggggggggggggggggggggggggggggggg"Write failed: Broken pipe

[dguster@localhost ~]$ sudo docker run tempserver
[sudo] password for dguster:
Waiting for connection...

Write failed: Broken pipe
```

An interesting error was also observed on occasion when reading from the dump and writing the dump to a user directory. In the example below, two 4kB pages of memory were requested to be written to the file heapdumpx. When an attempt was made to read that file with the “od” command an input/output error was returned. Further on the server code itself (tempserver) returned a fatal error. That error appears to be related to memory getting contaminated so that the values (which may be pointers) are corrupted. A look at the error report for the stack (contains the low level instructions) supports this assessment.

```
(gdb) dump memory /home/dguster/heapdumpx 0x55f2fbea5000 0x55f2fbea7000
(gdb) quit

[dguster@localhost ~]$ od -w1 -c -Ad heapdumpx
-bash: /usr/bin/od: Input/output error

[dguster@localhost ~]$ sudo docker run tempserver
Waiting for connection...

#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGBUS (0x7) at pc=0x00007f27ec2baa2e, pid=6, tid=16

Stack slot to memory mapping:
stack at sp + 0 slots: 0x00007f27e863ce50 points into unknown readable
memory: a0 ce 63 e8 27 7f 00 00
stack at sp + 1 slots: 0x0000000000000001 is an unknown value
stack at sp + 2 slots: 0x00007f27e863ce50 points into unknown readable
memory: a0 ce 63 e8 27 7f 00 00
```

Further, if the address range of the heap is extended past its size as seen in the example below, a broken pipe error may result. Note the error appears on both the debug and the tempserver session. Thus it results in a denial of service because the container that holds the tempserver code crashes.

```
set {char [80000000]} 0x55c8fcbc3000 = "gggggggggggggggggggggggggggg"
Write failed: Broken pipe

[dguster@localhost ~]$ sudo docker run tempserver
Waiting for connection...

Write failed: Broken pipe
```

## DISCUSSION/CONCLUSIONS

Isolation of services is an important tool in providing sound security. The Docker container software certainly added to this goal. An examination of the Docker related processes revealed three major processes resulted: the Docker software itself, the shell for executing the tempserver class and the protected image of the tempserver class. The goal of this paper was to examine how well the tempserver image itself was protected. First, based on the observations herein it was well protected from user level attempts to read or overwrite the data. It was also interesting to note that the memory addresses were obscured to some extent by truncating the leading part of the full address. Further, the address ranges used by the tempserver image were not in the "iomem" map which is the LINUX tool which provides the memory ranges assigned to the VM host. This would make it somewhat more difficult for a hacker to find the memory range and if the ranges were guessed that would then be better protected. So overall the Docker software was effective at mitigating memory level attacks from the user level.

If the hacker obtains root level access then a couple of issues were identified. In cases where a debugger was used to read content from the heap and then try to write them to a file in the default directory there were issues where the write would fail and the server process would abort. If the write was successful the actual contents of the memory segment would not be written, rather the entire file would contain null values (hex 0's). So data could not be compromised based on the observations herein. In other cases, the attempt to write would result in a "broken pipe" and the server software would abort causing a denial of service. A look at the dump provided by java indicated that an instruction in the stack could not be resolved. While the container software did protect the integrity of the data it is still vulnerable to memory level attacks, if the hacker can obtain root access.

Overall using the Docker container software does add to the integrity of a server image. However, like any security software there are areas that need to be protected. Of note is memory protection. Prevention of execution of direct memory software such as C or debug at least needs to be monitored. Also, stateful inspection to try to identify and remove such direct memory access software is warranted. On a very basic level once again the results herein confirm the importance of limited root access and monitoring when it is used. As is typically the case there is no one security solution, but solutions that will reduce the probability of success and the size of the hacker pool that could achieve success.

## References

Aldrich, J. (2013). Why Objects Are Inevitable. *2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, (pp. 101-116).

Barbu, G., Hoogvorst, P., & Duc, G. (2012). Application-Replay Attack on Java Cards: When the Garbage Collector Gets Confused. In G. Barthe, B. Livshits, & R. S. (eds.), *ESSoS 2012. LNCS, vol. 7159* (pp. 1-13). Springer, Heidelberg.

Barbu, G., Thiebeauld, H., & Guerin, V. (2010). Attacks on java card 3.0 Combining Fault and Logical Attacks. In D. Gollmann, J. Lanet, & J. I.-C. (eds), *CARDIS LNCS, vol 6035* (pp. 148-163). Springer, Heidelberg .

Bouffard, G., Lackner, M., Lanet, J.L., & Johannes, L. (2015). Heap . . . Hop! Heap Is Also Vulnerable . In M. J. (Eds.), *CARDIS LNCS 8968* (pp. 18-31).

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

Callum, C., Singer, J., & Vengerov, D. (2015). The Judgement of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes. *International Symposium on Memory Management*, (pp. 143-156).

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.

*CVE*. (2016). Retrieved from https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html

Drake, J. (2011). *Exploiting Memory Corruption Vulnerabilities in the Java Runtime*. Retrieved from https://media.blackhat.com/bh-ad-11/Drake/bh-ad-11-DrakeExploiting_Java_Memory_Corruption-WP.pdf

Dulgie, S. (2016). Dirty COW Vulnerability: Impact on Containers. Retrieved from https://blog.aquasec.com/dirty-cow-vulnerability-impact-on-containers

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* .

Govindavajhala, S., & Appel, A. (2003). Using Memeory Errorrs to Attack Virtual Machines. *IEEE Symposium on Security and Privacy*, (pp. 11-14).

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

*Linux Memory Management*. (2016). Retrieved from http://tldp.org/HOWTO/KernelAnalysisHOWTO-7.html

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

Rama Krishna, S. & Padmaja Rani, B. (2017). Virtualization Security Issues and Mitigations in Cloud Computing. Retrieved from https://www.researchgate.net/publication/311215879_Virtualization_Security_Issues_and_Mitigations_in_Cloud_Computing

Utility, T. J. (2016). Retrieved from https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr014.html

*What is Virtualization?*. (2019). Retrieved from https://www.vmware.com/solutions/virtualization.html

## Appendix

```
[dguster@localhost ~]$ sudo docker run tempserver
Waiting for connection...

#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGBUS (0x7) at pc=0x00007f27ec2baa2e, pid=6, tid=16
#
# JRE version: OpenJDK Runtime Environment (11.0.2+9) (build 11.0.2+9-
Debian-3bpo91)
# Java VM: OpenJDK 64-Bit Server VM (11.0.2+9-Debian-3bpo91, mixed mode,
sharing, tiered, compressed oops, serial gc, linux-amd64)
# Problematic frame:
# V  [libjvm.so+0xbdba2e]
```

```
#
# Core dump will be written. Default location: //core.6
#
# Can not save log file, dump to screen..
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGBUS (0x7) at pc=0x00007f27ec2baa2e, pid=6, tid=16
#
# JRE version: OpenJDK Runtime Environment (11.0.2+9) (build 11.0.2+9-
Debian-3bpo91)
# Java VM: OpenJDK 64-Bit Server VM (11.0.2+9-Debian-3bpo91, mixed mode,
sharing, tiered, compressed oops, serial gc, linux-amd64)
# Problematic frame:
# V  [libjvm.so+0xbdba2e]
#
# Core dump will be written. Default location: //core.6
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.java.com/bugreport/crash.jsp
#

---------------  S U M M A R Y ------------

Command Line: TempServer

Host: Quad-Core AMD Opteron(tm) Processor 2384, 1 cores, 1G, Linux
Time: Wed Feb 27 13:17:14 2019 UTC elapsed time: 529 seconds (0d 0h 8m 49s)

---------------  T H R E A D  ---------------

Current thread (0x00007f27e4093800):  WatcherThread [stack:
0x00007f27e853e000,0x00007f27e863e000] [id=16]

Stack: [0x00007f27e853e000,0x00007f27e863e000],  sp=0x00007f27e863ce10,
free space=1019k
Native frames: (J=compiled Java code, A=aot compiled Java code, j=interpreted,
Vv=VM code, C=native code)
V  [libjvm.so+0xbdba2e]
V  [libjvm.so+0xc9524d]
V  [libjvm.so+0xd0011e]
V  [libjvm.so+0xd2c699]
V  [libjvm.so+0xb935c0]
```

```
siginfo: si_signo: 7 (SIGBUS), si_code: 2 (BUS_ADRERR), si_addr:
0x00007f27ed7a1800

Register to memory mapping:

RAX=0x0000000000120cf8 is an unknown value
RBX=0x00007f27ed7a1800 points into unknown readable memory: f8 0c 12 00
00 00 00 00
RCX=0x0000000000000002 is an unknown value
RDX=0x00007f27ebbe8800: <offset 0x0000000000509800> in /usr/lib/jvm/java-
11-openjdk-amd64/lib/server/libjvm.so at 0x00007f27eb6df000
RSP=0x00007f27e863ce10 points into unknown readable memory: 50 ce 63 e8
27 7f 00 00
RBP=0x00007f27e863ce20 points into unknown readable memory: 50 ce 63 e8
27 7f 00 00
RSI=0x0000000000000032 is an unknown value
RDI=0x00007f27e4022d30 points into unknown readable memory: 50 d3 96 ec
27 7f 00 00
R8 =0x0000000000000001 is an unknown value
R9 =0x00000000000126c7 is an unknown value
R10=0x00007f27e863cd70 points into unknown readable memory: 00 00 00 00
00 00 00 00
R11=0x00225953864993a8 is an unknown value
R12=0x0000000000000008 is an unknown value
R13=0x00007f27e4093120 points into unknown readable memory: 40 31 09 e4
27 7f 00 00
R14=0x0000000000000002 is an unknown value
R15=0x00007f27eca86020: <offset 0x00000000013a7020> in /usr/lib/jvm/java-
11-openjdk-amd64/lib/server/libjvm.so at 0x00007f27eb6df000


Registers:
RAX=0x0000000000120cf8, RBX=0x00007f27ed7a1800,
RCX=0x0000000000000002, RDX=0x00007f27ebbe8800
RSP=0x00007f27e863ce10, RBP=0x00007f27e863ce20,
RSI=0x0000000000000032, RDI=0x00007f27e4022d30
R8 =0x0000000000000001, R9 =0x00000000000126c7,
R10=0x00007f27e863cd70, R11=0x00225953864993a8
R12=0x0000000000000008, R13=0x00007f27e4093120,
R14=0x0000000000000002, R15=0x00007f27eca86020
RIP=0x00007f27ec2baa2e, EFLAGS=0x0000000000010202,
CSGSFS=0x0000000000000033, ERR=0x0000000000000007
  TRAPNO=0x000000000000000e
```

```
Top of Stack: (sp=0x00007f27e863ce10)
0x00007f27e863ce10:   00007f27e863ce50 0000000000000001
0x00007f27e863ce20:   00007f27e863ce50 00007f27ec37424d
0x00007f27e863ce30:   00000000000126c7 0000000000000001
0x00007f27e863ce40:   0000000000000032 0000000000000032

Instructions: (pc=0x00007f27ec2baa2e)
0x00007f27ec2baa0e:   00 00 48 89 f8 48 8b 7f 38 48 85 ff 74 1b 55 48
0x00007f27ec2baa1e:   89 e5 53 48 83 ec 08 48 8b 58 28 48 8b 07 ff 10
0x00007f27ec2baa2e:   48 89 03 48 83 c4 08 5b 5d f3 c3 90 66 0f 1f 44
0x00007f27ec2baa3e:   00 00 55 48 8b 4f 28 48 89 f0 48 63 f2 48 8d 15

Stack slot to memory mapping:
stack at sp + 0 slots: 0x00007f27e863ce50 points into unknown readable
memory: a0 ce 63 e8 27 7f 00 00
stack at sp + 1 slots: 0x0000000000000001 is an unknown value
stack at sp + 2 slots: 0x00007f27e863ce50 points into unknown readable
memory: a0 ce 63 e8 27 7f 00 00
stack at sp + 3 slots: 0x00007f27ec37424d: <offset 0x0000000000c9524d> in
/usr/lib/jvm/java-11-openjdk-amd64/lib/server/libjvm.so at 0x00007f27eb6df000
stack at sp + 4 slots: 0x00000000000126c7 is an unknown value
stack at sp + 5 slots: 0x0000000000000001 is an unknown value
stack at sp + 6 slots: 0x0000000000000032 is an unknown value
stack at sp + 7 slots: 0x0000000000000032 is an unknown value


---------------  P R O C E S S  ---------------

Threads class SMR info:
_java_thread_list=0x00007f27e409b9d0, length=9, elements={
0x00007f27e4010800, 0x00007f27e4048000, 0x00007f27e404a000,
0x00007f27e4051800,
0x00007f27e4053800, 0x00007f27e4055800, 0x00007f27e4057800,
0x00007f27e4091000,
0x00007f27e409a000
}

Java Threads: ( => current thread )
  0x00007f27e4010800 JavaThread "main" [_thread_in_native, id=7,
stack(0x00007f27ed7a8000,0x00007f27ed8a9000)]
  0x00007f27e4048000 JavaThread "Reference Handler" daemon
[_thread_blocked, id=9, stack(0x00007f27e8df1000,0x00007f27e8ef2000)]
```

```
  0x00007f27e404a000 JavaThread "Finalizer" daemon [_thread_blocked, id=10,
stack(0x00007f27e8cf0000,0x00007f27e8df1000)]
  0x00007f27e4051800 JavaThread "Signal Dispatcher" daemon
[_thread_blocked, id=11, stack(0x00007f27e8a42000,0x00007f27e8b43000)]
  0x00007f27e4053800 JavaThread "C2 CompilerThread0" daemon
[_thread_blocked, id=12, stack(0x00007f27e8941000,0x00007f27e8a42000)]
  0x00007f27e4055800 JavaThread "C1 CompilerThread0" daemon
[_thread_blocked, id=13, stack(0x00007f27e8840000,0x00007f27e8941000)]
  0x00007f27e4057800 JavaThread "Sweeper thread" daemon [_thread_blocked,
id=14, stack(0x00007f27e873f000,0x00007f27e8840000)]
  0x00007f27e4091000 JavaThread "Service Thread" daemon [_thread_blocked,
id=15, stack(0x00007f27e863e000,0x00007f27e873f000)]
  0x00007f27e409a000 JavaThread "Common-Cleaner" daemon
[_thread_blocked, id=17, stack(0x00007f27e843c000,0x00007f27e853d000)]
```