APPLICATION LEVEL MEMORY MANAGEMENT STRATGIES VIA THE "GARBAGE COLLECTOR": PERFORMANCE AND SECURITY RAMIFICATIONS

Raqeeb, Abdul
St. Cloud State University
rabdul@stcloudstate.edu

Guster, Dennis C.
St. Cloud State University
dcguster@stcloudstate.edu

Schmidt, Mark B.
St. Cloud State University
mbschmidt@stcloudstate.edu

ABSTRACT

This paper provides an overview of some of the basic vulnerabilities with in the heap. Two examples are provided to illustrate the vulnerabilities on an operational level. The first example depicts how easy it is within the LINUX operating system to find the heap for a given application. Once found a method is presented to overwrite the heap which results in a denial of service attack on the applications level. The second example focuses on memory management as accomplished by the "garbage collector" A simple analysis of the strength of objects within the heap is undertaken. It is found that overtime the strength deteriorates and there is a surprising amount of overhead required to perform the management function. The authors suggest that the examples provide a baseline from which to evaluate the vulnerabilities of the heap in more detail. Further, they suggest more research is needed, particularly in regard to how quickly object strength deteriorates over time.

INTRODUCTION

Object oriented programming (OPP) offers many advantages to developers. Even twenty years ago the basic concepts such as reuse of code, better structured programs and easier transition from analysis to implementation were recognized (Guimaraes, 1995). Another related concept is the importance of interoperability. Given the diversity of applications that share information within today's cloud based computing architectures it is critical that the developers follow a OOP approach (Aldrich, 2013).For this environment to function properly it is critical that the location of the objects as well as the memory segments themselves be managed effectively. Objects that are weak or are no longer used need to be removed and the memory recovered. Conversely, objects that are strong and still used need to be protected and optimized so that the application code runs securely and effectively. Keeping track of where in memory objects reside is typically instantiated in the heap (Callum, Singer, & Vengerov, 2015). It is relatively easy to find where in memory the heap resides. In the example below a java class entitled "TempServer" is instantiated under process ID 11355. A simple memory map for this process is displayed filtered by the string heap. This entry reveals that the heap resides at relative memory address range: 020d4000-020f5000.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 of 1GB. The permissions on this segment are set to read, write and private (hidden from other classes within the package).

```
CRL\den.gus@os:~$ ps -aux | grep java

CRL\de+ 24865  0.1  0.6 2234956 25612 pts/2   Sl+  16:50   0:00 java TempServer

CRL\den.gus@eros:~$ cd /proc/24865

CRL\den.gus@os:/proc/24865$ cat maps | grep heap

020d4000-020f5000 rw-p 00000000 00:00 0                  [heap]

CRL\den.gus@os:~$ java -XX:+PrintFlagsFinal -version | grep -iE
'HeapSize|PermSize|ThreadStackSize'

uintx MaxHeapSize              := 1040187392      {product}
```

This memory segment is important for a number of reasons. First, it is very small segment within all available memory and finding it randomly is hence difficult. Second, it is not loaded into the kernel space but rather into the user space which means that the user that owns the process generating the task can view and modify the heap (certainly the root could do this as well) (Linux Memory Management , 2016). Therefore, a hacker with user level access to an application could simply contaminate the heap which could bring the whole application to a halt. This is certainly more problematic than a flood ping attack against the port to which the application is assigned. The scenario below illustrated how the gnu debugger could be used to extract the contents of the heap for process ID 24865 and how objects are linked to a relative memory address. The examples also show that the jmap command could also be used to obtain memory mapping information and that the “set” command in the debugger could be used to over write a memory address.

```
CRL\den.gus@os:/proc/24865$ gdb --pid 24865

(gdb) dump memory ~/gbpap2 0x02200000 0x02200f00

CRL\den.gus@os:~$ ls -l gb*
-rw-r--r-- 1 CRL\den.gus CRL\domain^users 256 Oct  7  2016 gbpap2

CRL\den.gus@os:~$ xxd gbpap
0000000: 0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
0000010: 5093 520a ae7f 0000 0000 0000 0000 0000  P.R.............
0000020: 2000 0000 0000 0000 4100 0000 0000 0000   .......A.......
0000030: 2f75 7372 2f6c 6962 2f6a 766d 2f6a 6176  /usr/lib/jvm/jav
0000040: 612d 372d 6f70 656e 6a64 6b2d 616d 6436  a-7-openjdk-amd6
0000050: 342f 6a72 652f 6269 6e2f 6a61 7661 0000  4/jre/bin/java..

CRL\den.gus@os:~$ jmap 24865
Attaching to process ID 24865, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 24.111-b01
0x0000000000400000      6K      /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java
0x00007f00c3d92000      437K    /lib/x86_64-linux-gnu/libpcre.so.3.13.1

(gdb) set {char [256]} 0x0216f000 = "Acefffffffffffffff"
```

From the examples illustrated above it is clear that it is quite easy to find the heap as well as the objects that it is mapping for any given process. What is displayed is just a small fraction of what is potentially available. For information on how to obtain additional information please see: *The JMAP Utility* (Utility, 2016). In addition to potential denial of service attacks against the heap there are numerous performance related and logical attack scenarios that can be launched against the heap (Barbu, Thiebeauld, & Guerin, 2010); (Barbu-, Hoogvorst, & Duc, 2012). The depth of this potential vulnerability is depicted in detail in (Drake, 2011). In this work he stated that the “java runtime environment (JRE) was found 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. Based on trending, it is safe to assume that many more vulnerabilities remain to be found.” Given this data additional research related to assuring the wellbeing of the heap is warranted. One of the simplest attacks that could be launched by a hacker would be to use a debugger to modify the contents of the heap with the goal of either creating a denial of service or to exploit vulnerabilities within the object mapping. In the example below the first byte of the heap is changed from its value above of binary 0 (represented by the two nibbles 00) to a binary 7 (07).

```
(gdb) set {int}0x00ef1000 = 7

(gdb) dump memory ~/gcpap3 0x00ef1000 0x00ef1010

CRL\den.gus@os:~$ xxd gcpap3

0000000: 0700 0000 0000 0000 2100 0000 0000 0000  ........!.......
```

REVIEW OF LITERATURE

Overview

The depth of vulnerabilities related to memory management is well documented. A quick look at the CVE vulnerability list reveals hundreds of active vulnerabilities of severity scores greater than seven (on a 10 point scale) (CVE , 2016). (Chen, et al., 2011) discuss the types of vulnerabilities that have been observed and the degree to which they are exploited. In terms of mitigating attacks they found that there are techniques which often protect against specific exploits of a vulnerability, but no one solution that is universally effective. Further, they state that no effective techniques exist to handle semantic vulnerabilities related to violations of high-level security invariants.

It is also worth noting that unexpected tampering with memory segments can be truly problematic to any application and often no protection mechanisms are in place. (Ferreira, et al., 2012) state that often the most critical software running on a node, the operating system (OS), is currently left relatively unprotected in main memory. Because the OS is at the top of the software hierarchy, resiliency is very important. Recent studies have shown that there are still significant memory errors in the region supporting the OS (Levy, Ferreira, Bridges, Thompson, & Trott, 2015). In the case of this paper this is important because the focus is on the heap within an application that resides in the low area of main memory.

Of course memory management is a function of the operating system and the heart of the operating system is the kernel. Just because one gains root access via sudo does not mean they have rights in kernel space. For a detailed explanation see: (Stack Overflow, 2016). This design constraint means that it is much more difficult to compromise data stored in kernel managed memory, but not impossible if the hacker can recompile objects called by the kernel or the kernel itself. This problem is not new and in fact has been an issue for at least the past ten years (Criswell, Geoffray, & Adve, 2009). In fact the problem has not lessened, for example, (Xu & al, 2015) depicts a specific problem in that regard and explain how the kernel and its associated memory segment can be compromised. They further state that the methodology has gone beyond a brute force random design to a "collision" technique that can be used to zero in on a particular memory segment without having access to a system generated memory map. The fact that successful attacks against the kernel and its associated memory segments are taking place strengthens the arguments in this paper. Further, the general review offered herein is just the tip of the iceberg. There are numerous sites such as: (Zero, 2015) that provide information about the various types of attacks and explain their effectiveness. Among these buffer overflows are often the most effective and sensitive, ALL should be monitored accordingly (Avijit, Gupta, & Gupta, 2004). However the primary concerns of this paper are related to denial of service and performance issues related to the heap within the java runtime.

Attacks Against the Heap

Attacks against the heap are not new, in fact (Govindavajhala & Appel, 2003) reported vulnerabilities in most C based languages and stated that a compromise of just a single byte within the heap could lead to problems. The problem has continues to gain importance over the years and alternate memory managers have been devised to effectively combat the problem (Novark & Berger, 2010).

The most common attack against the heap is a buffer overflow which allows an attacker to get their code into the memory associated with that program and execute it (CVE , 2016). An application written in a programming language is vulnerable to such attacks if the following criteria are met:

- The language used allows buffer overflows to take place and,
- data is copied from buffer to buffer within the on the stack while not verifying the size of the block transferred and,
- proven prevention methods such as canary values are not implemented (Stack Overflow, 2016).

The language used can have a big impact on potential vulnerability. An important factor is whether or not the language used allows direct memory access. The language used in this project java does not, but assembler and C/C++ do (Buffer Overflows , 2016). However, one should note that because java does not allow direct memory access it is not entirely safe. For example, if a hacker could find out the memory address of a java heap then they could use assembler or C/C++ to overwrite and create denial of service attack against the application that heap is supporting. In regard to the language used herein, java, it is considered safe because it protects against buffer overflow vulnerabilities since it is a managed memory model. However, in its associated modules such as JVM and JDK there can be buffer overflow vulnerabilities (Secunia Research Community, 2016).

The importance of a canary value (came from canaries in coal mines that indicated the air was not safe to breath) cannot be overstated when protecting against buffer overflow attacks. It is a fixed or random value embedded in the buffer and if corrupted it means the data stored within that buffer has been corrupted. It has proved an effective memory protection mechanism. For an overview and research related to the current state of canary values see: (Petsios, Kemerlis, Polychronakis, Keromytis, & DynaGuard, 2015).

To an extent the java heap is resilient to attacks involving small amounts of data, to be successful often 64KB blocks need to be overwritten (Bouffard, Lackner, Lanet, & Johannes, 2015).

Further as one would expect there are differences in vulnerabilities based on the operating system (McDonald & Valasek, 2009). Out of the box as one would expect there are advantages in using sophisticated UNIX systems such as BSD (Novark & Berger, 2010). Assuming the appropriate access level is obtained it is still fairly easy to launch a successful denial of service attack against the java run time heap of a class that is providing a service via network socket calls. The blueprint of a simple DOS attack against the heap in the next section will illustrate this vulnerability.

METHODOLOGY

Blueprint of a Simple DOS Attack Against the Heap

In the example below the server side of a temperature conversion java socket call program is run and stays resident in memory while waiting for client connections. The running code is assigned process ID 24325. Then by changing to that processes' directory in /proc and displaying the maps file and searching for the string "heap" the hex relative memory address range for the heap is obtained.

```
CRL\den.gus@os:~$ ps -aux | grep java
CRL\de+ 24325  0.3  0.6 2234956 25364 pts/6   Sl+  15:05   0:00 java TempServer

CRL\den.gus@os:~$ cd /proc/24325
CRL\den.gus@os:/proc/24325$ cat maps | grep heap
017b0000-017d1000 rw-p 00000000 00:00 0       [heap]
```

Once the memory range is obtained a debugger (allows direct memory access) can be attached to that process ID. It is then possible to use the set command to overwrite the current contents of the heap. In the example below we are overwriting the first 20000hex Bytes with a string of characters plus it will null value fill (hex 0s) whatever space is allocated beyond that string.

```
CRL\den.gus@os:/proc/24325$ gdb --pid 24325

(gdb) set {char [20000]} 0x017b0000 = "When the heap is compromised you are in
trouble!"
```

To see if the overwrite attempt was successful we dump the contents of the first 4096 bytes of the memory location of the heap using the dump memory command within the debugger. As expected the string of characters appears followed by null values.

```
(gdb) dump memory ~/gcpapH6 0x017b0000 0x017b1000

CRL\den.gus@os:~$ xxd gcpapH6 | more
0000000: 5768 656e 2074 6865 2068 6561 7020 6973  When the heap is
0000010: 2063 6f6d 7072 6f6d 6973 6564 2079 6f75   compromised you
0000020: 2061 7265 2069 6e20 7472 6f75 626c 6521   are in trouble!
0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Before the heap was over written the client could attach to the service which was running on port 18002. After the heap is overwritten, the debugger is detached and a connection attempt fails because the service crashes.

```
tcp        0      0 0.0.0.0:18002           0.0.0.0:*               LISTEN      1018168 450590
24325/java
tcp        0      0 127.0.0.1:52393         127.0.0.1:18002         ESTABLISHED 1018168
450280      24902/java

(gdb) detach

CRL\den.gus@os:~/javaclass$ java TempClient
Connecting to: localhost
Enter a Tempreture in F: 44
Exception in thread "main" java.net.SocketException: Connection reset
        at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:118)
        at java.net.SocketOutputStream.write(SocketOutputStream.java:138)
        at java.io.DataOutputStream.writeBytes(DataOutputStream.java:276)
        at TempClient.main(TempClient.java:86)
```

Blueprint of Assessing the Strength of Objects within the Heap

Setup

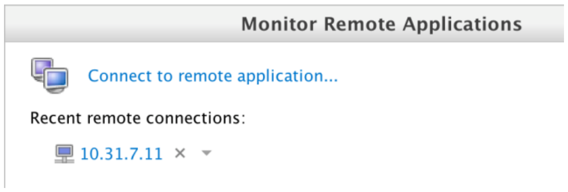
To obtain the needed information about the strength of objects in the heap and the role of the garbage collector in this process a Java Profiler called Yourkit was deployed. Yourkit is available on all common operating system platforms such as Windows/Mac/Linux.

Because the java code used herein was deployed on an Ubuntu Server which doesn’t support GUI, the required GUI can be provided by the client side by connecting to a Mac or Windows Machine. This can be achieved by installing the profiler on both the machines.

The following steps describe this process:

1) Download the Yourkit profiler for LINUX and extract it.
2) Next, the profiler can be attached to java application using the command below.
   java -agentpath:<path to directory>/yjp-2016.02/bin/linux-x86-64/libyjpagent.so <Java class>

3) To connect to this application, start the Yourkit profile on the Mac/Windows machine.
4) Then select the “Connect to remote application” and enter the IP address of the machine that is running the java application as depicted in Figure 1.
5) Now, the client is connected and allows monitoring of a given application.



Monitor Remote Applications

Connect to remote application...

Recent remote connections:

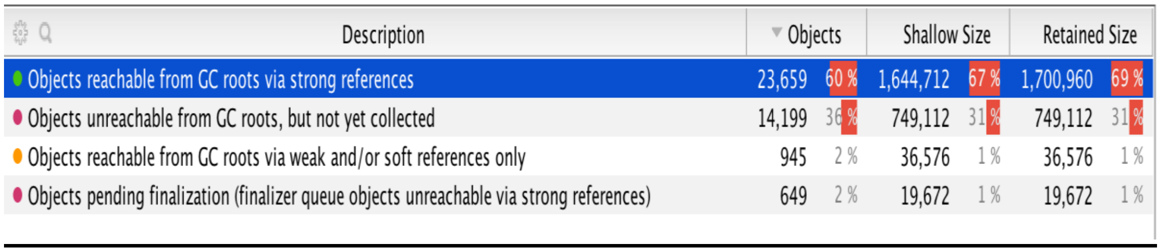
10.31.7.11

Figure 1: Yourkit Client Connection Screen

Observing the strength of the objects:

1st Scenario – immediately after the execution

In this case most of the objects are reachable from the GC (garbage collector) through a strong reference. There are also few unreachable objects which exist before the execution of the java code. The figure below is the screenshot that depicts these relationships.



Description	Objects		Shallow Size		Retained Size	
● Objects reachable from GC roots via strong references	23,659	60 %	1,644,712	67 %	1,700,960	69 %
● Objects unreachable from GC roots, but not yet collected	14,199	36 %	749,112	31 %	749,112	31 %
● Objects reachable from GC roots via weak and/or soft references only	945	2 %	36,576	1 %	36,576	1 %
● Objects pending finalization (finalizer queue objects unreachable via strong references)	649	2 %	19,672	1 %	19,672	1 %

Figure 2: Initial Object Strengths

In this client/server Scenario, a java class entitles DateServer creates the service which listens for connections. This class can be viewed as a socket call program in Java. When a connection to the DateServer is made to the client another java class called DateClient is used. Below in Figure 3 are the frequency observations while the Client makes calls to Server. One can see at startup the calls peak and there is a series of lessor spikes above the base line (approximately 175/s) when the client makes the calls.

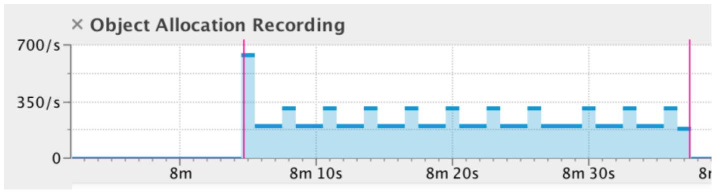


Figure 3: Frequency of client calls to the server

For the sake of consistency the state of other memory segments was observed in Figure 4. This Figure shows that both the size of non-heap memory and the number of loaded classes increased as well.

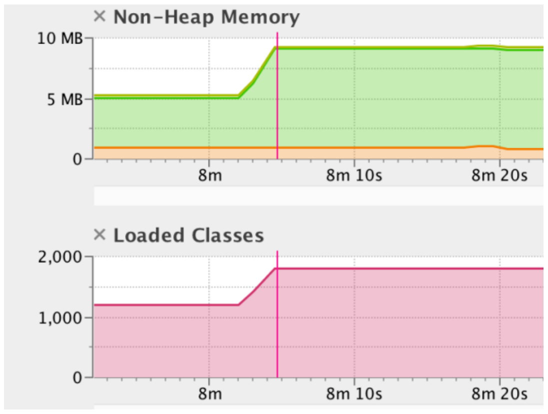
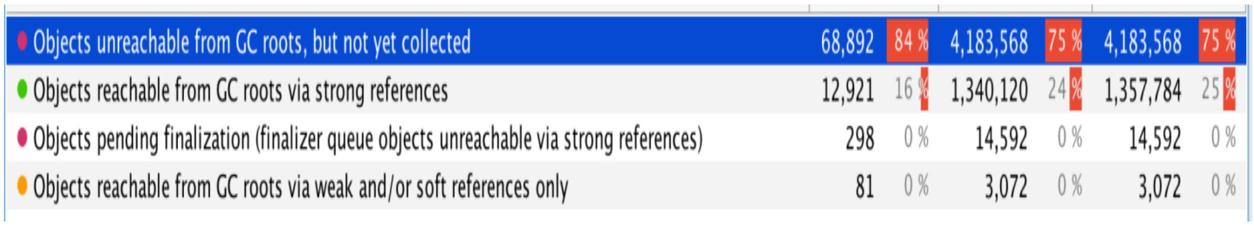


Figure 4: Non-heap memory usage and number of loaded classes

2nd Scenario – 1 day after the execution

In order to assess the decay of object strength over time from the 1st scenario the service was left running for a day. The results are depicted in Figure 5. Interestingly, most of the objects i.e. 84% are unreachable and further, they have not yet been collected by Garbage collector. However, there are still 16% of the objects which are reachable from strong references. There are a negligible number of objects which are available through weak reference. Because the service typically will still work after just one day one would expect that the reachable objects are the important ones. However, any application instantiated memory if given enough time without refreshing may be vulnerable to not working due to loss object references.



● Objects unreachable from GC roots, but not yet collected	68,892	84 %	4,183,568	75 %	4,183,568	75 %
● Objects reachable from GC roots via strong references	12,921	16 %	1,340,120	24 %	1,357,784	25 %
● Objects pending finalization (finalizer queue objects unreachable via strong references)	298	0 %	14,592	0 %	14,592	0 %
● Objects reachable from GC roots via weak and/or soft references only	81	0 %	3,072	0 %	3,072	0 %

Figure 5: Strength of Objects after 1 day

As one would expect the heap memory is relatively constant because the Date service is pretty much idle. As one would expect to some extent the heap memory is freed up by the garbage collector during this period of time (such as survivor space). Figure 6 below shows the memory allocation observed in the heap.

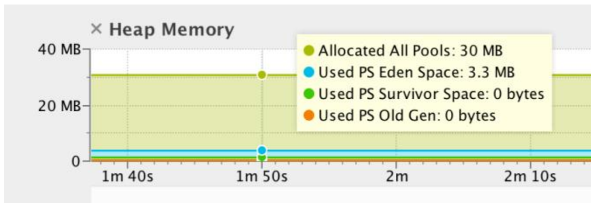


Figure 6: Heap memory allocation during idle time

Correspondingly, as one would expect the number of classes loaded are fairly constant, because the system is pretty much idle Figure 7 depicts this relationship.

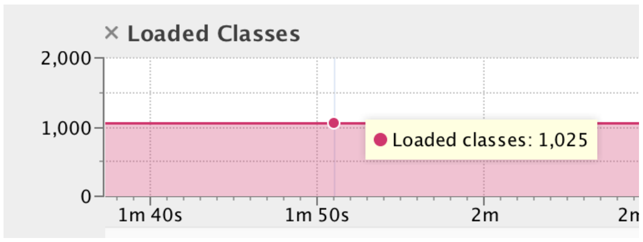
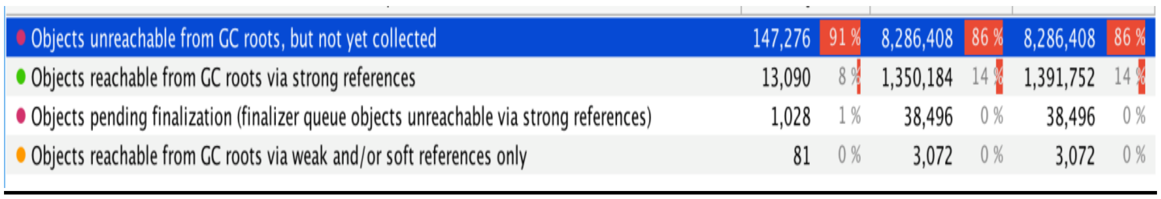


Figure 7: Number of loaded classes during idle time

3rd Scenario – after editing the heap

In this case, a portion of the heap from the 2nd scenario has been editing with random values. There is drastic increase in the number of objects which are not reachable from the Garbage Collector. Figure 8 below provides a description of the relationships observed.



● Objects unreachable from GC roots, but not yet collected	147,276	91 %	8,286,408	86 %	8,286,408	86 %
● Objects reachable from GC roots via strong references	13,090	8 %	1,350,184	14 %	1,391,752	14 %
● Objects pending finalization (finalizer queue objects unreachable via strong references)	1,028	1 %	38,496	0 %	38,496	0 %
● Objects reachable from GC roots via weak and/or soft references only	81	0 %	3,072	0 %	3,072	0 %

Figure 8: Strength of objects after tampering with the heap

Putting random values in the heap increased object allocation. Of course one would have to expect that the objects would be bogus and perhaps meaningless. Figure 9 below depicts the activity over about a 35 second time frame.

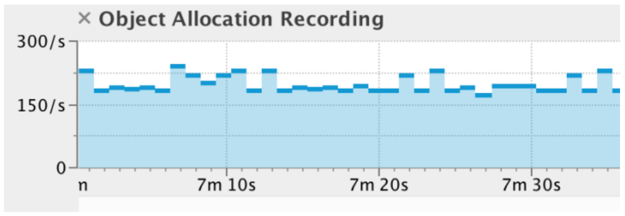


Figure 9: Object allocation after the heap had been tampered with

Correspondingly, as one would expect the heap memory is increased a little bit by about 1 MB. Below is the graphic entitled Figure 10.

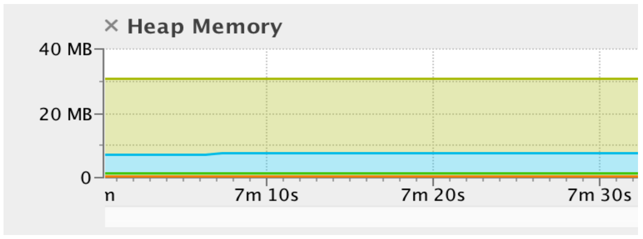


Figure 10: Heap memory size after the heap had been tampered with

Interestingly, a fairly significant amount of CPU usage is required to manage the objects and to edit the Heap Memory. At one point it peaks at almost 25% of the assigned CPU and this relationship is shown in Figure 11 below.

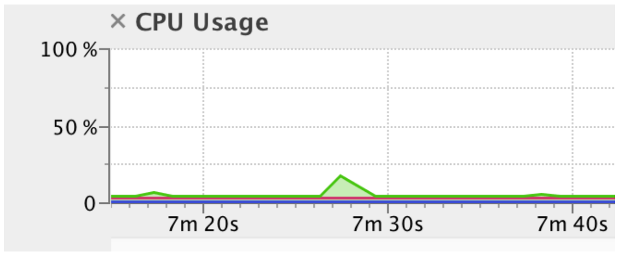


Figure 11: CPU usage to manage bogus objects

DISCUSSION/CONCLUSIONS

It is most disturbing to see the extent a denial of service attack can be launched if access is gained to the memory space of any given application. In this case the vulnerability of the heap is illustrated and in fact its memory address range is available with a simple map call by the owner of the application. Object oriented programming is the foundation of much of the software development over the past decade. The benefits of object oriented programming are well documented and being able to reuse code has a very positive economic impact. However, mapping the objects is a necessary function and as the experiments showed if it's compromised the entire application may well fail.

The complexity of the mapping is amazing in terms of the number of classes used as well as the volume of space used for indexing. It is also clear that managing that space is challenging as well which means tuning the garbage collector can have positive effects. This is a dynamic process in that objects that are no longer used need to be removed and the space reallocated. Based on the number of references observed within the data herein it is a challenge to keep the size to a minimum thereby helping the lookup performance metrics. It is also important to realize that the strength of the object references deteriorate over time. A search of the literature provides limited information about the degree that this is a problem. However anyone that has worked as a system administrator knows that over time services get flakey or stop working altogether. Often the solution is just to reboot that service and performance is then restored at least for the short term. This behavior would be indicative of corrupted memory and would be consistent with the object map or heap losing some of its pointers. This is certainly an area that could benefit from further research. Research is needed with a variety of applications involving numerous objects to get a true understanding of the effect of the object references deterioration problem.

In terms of educating students in regard to the vulnerabilities within memory, the goal was to devise some simple examples. This paper presented some examples that illustrated the problem. Further, these examples built on the capabilities of the LINUX operating system. Therefore, it gave students operationalized examples and a framework from which to pursue other memory related investigations.

In terms, of protecting against such attacks it is clear that denying rights and preventing software with direct memory access from running on a given host is critical. Cloud computing associated with virtualization certainly offer some degree of protection. First the attacker must pierce the cloud and then perhaps a zone in the cloud (defined by sub-netting). Next the attacker must compromise the host in question and obtain the rights and gain access to the appropriate software. The fact that the host may be virtual often alters the memory locations from that of a dedicated host. Hopefully, this layered approach offer a sound first line of defense. However, the literature indicates that there are numerous successes by hackers in attacking memory. Hopefully the information offered herein will be useful to students in gaining a foundation of how the process actually works.

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).

Avijit, K., Gupta, P., & Gupta, D. (2004). TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. *13th conference on USENIX Security Symposium.* San Diego.

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).

Buffer Overflows . (2016). Retrieved from https://www.owasp.org/index.php/Buffer_Overflows

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.

Criswell, J., Geoffray, N., & Adve, V. (2009). Memory Safety for Low-Level Software/Hardware Interactions. *18th conference on USENIX security symposium*, (pp. 83-100). Montreal, Canada .

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-Drake-Exploiting_Java_Memory_Corruption-WP.pdf

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.

Levy, S., Ferreira, K. B., Bridges, P. G., Thompson, A. P., & Trott, C. (2015). A Study of the Viability of Exploiting Memory Content Similarity to Improve Resilience to Memory Errors International . *Journal of High Performance Computing Applications* .

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

McDonald, J., & Valasek, C. (2009). Practical Windows XP/2003 Heap Exploitation. *Black Hat USA.*

Novark, G., & Berger, E. (2010). DieHarder: Securing the Heap. *ACM Conference on Computer and Communications Security.*

Petsios, T., Kemerlis, V. P., Polychronakis, M., Keromytis, A. D., & DynaGuard. (2015). Armoring Canary-Based Protections Against Brute-Force Attacks. *Annual Computer Security Applications Conference.* Los Angeles, CA.

Secunia Research Community. (2016). Retrieved from http://secunia.com/advisories/25295

Stack Overflow. (2016). Retrieved from http://stackoverflow.com/questions/21761185/is-there-a-difference-between-sudo-mode-and-kernel-mode

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

Xu, W., & al, e. (2015). From Collision to Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. *ACM SIGSAC Conference on Computer and Communications Security*, (pp. 414-425).

Zero, P. (2015). Retrieved from https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html