SECURITY CONCERNS OF REGISTERS IN LINUX HOSTS: USING DEBUG TO FIND MEMORY ADDRESSES OF SENSITIVE DATA

Hazem Farra
Department of Information Assurance
St. Cloud State University
St. Cloud, MN 56301
faha1301@stcloudstate.edu

Dennis Guster, and Erich Rice
Department of Information Systems
St. Cloud State University
St. Cloud, MN 56301
dguster@stcloudstate.edu & eprice@stcloudstate.edu

Abstract

Security has never been more important in information technology. And often times the most important resource a company has is its data, the loss or deletion of which would be devastating. Especially with the advent of Cloud Computing and the use of shared physical hardware this has become an even greater concern. This paper looks at ways in which the LINUX operating system and various software tools can be used to show potential vulnerabilities in how memory is stored at the base layers of the operating system. It focuses on the registers, and how tools such as a debugger can be used to determine where memory resides and how it could potentially be attacked or disrupted. Finally, the paper discusses how these techniques and utilities could be used to provide students with a better understanding of how attacks could occur and the level of sophistication needed to deal with them.

1 Introduction

Historically, computer security techniques often centered on protecting the data on secondary storage and with the advent of the Internet that concern was expanded to include the network as well. Cloud computing and sharing of data in memory has increased the dependence on memory to provide timely responses to inquiries. Chen et al. (2011) depict in detail how memory itself is vulnerable as well. To some extent object oriented programming (OPP) while improving the development process presents dangerous vulnerabilities too. The basic concept of OPP, which creates objects, and facilitates the reuse of programming code was recognized by Guimaraes (1995). The successful implementation of OPP stresses the importance of interoperability. From a cost perspective this is most effective and is imperative because of the diversity of applications that share information within today's cloud based computing architectures (Aldrich, 2013). For this environment to function properly it is important that the location of the objects, the memory segments containing the data as well as the registers, which control the execution of the instructions themselves be managed effectively. Because of this modular approach an "index" of where the object required for any class will reside in memory is needed. In languages such as java, memory objects reside in the heap (Callum, Singer & Vengerov, 2015). Abdul, Guster, and Schmidt have demonstrated basic denial-of-service attacks against the "heap" (2017). In this study it was demonstrated that if the strength of the object pointers within the heap get compromised the whole class would no longer run.

To some extent the vulnerability is related to the concept of direct memory access (Buffer Overflow, 2016). In a LINUX host it is simple to find the memory within the segment for the heap. An example is provided below for a java class entitled "TempServer" which is running under process ID 26237. The memory map for this process is also displayed partially below:

```
CRL\den.gus@os:~$ netstat -apeen | grep java

tcp        0      0 0.0.0.0:18002           0.0.0.0:*               LISTEN
1018168411 2440358     26237/java

CRL\den.gus@os:~/javaclass$ cd /proc/26237

CRL\den.gus@os:/proc/26237$ cat maps

00400000-00401000 r-xp 00000000 08:01 915810 /usr/lib/jvm/java-7-openjdk-
amd64/jre/bin/java

00828000-00849000 rw-p 00000000 00:00 0                                  [heap]
```

This entry reveals that the heap resides at relative memory address range: 00828000-00849000. The permissions on this segment are set to read, write and private (hidden from other classes within the package). The need to have the write flag in place so that

the object map can be updated creates a degree of vulnerability. If a hacker could overwrite or change any of this then the whole class (or any another class that calls this class) could well be compromised. While the vulnerabilities of the heap are disturbing there are other layers of abstraction below the heap, which could also potentially be compromised. A good example of this would be the registers.

A register can be viewed as a small segment of very fast memory, which is integrated in the central processing unit. This design speeds up execution of instructions by providing quick access to what are often called values. Registers are the root of the memory hierarchy and offer the fastest means to manipulate data. However, like other types of memory, registers are volatile and their contents are lost if the computer is powered down. Often registers store temporary data while a program is executed. So that data and instruction can be introduced into the system some of the registers must be accessible to a user through instructions (Eazynotes, 2016). This ability to control data at the top of the memory provides hackers an opportunity to compromise a computer system despite the best efforts of security personnel and may circumvent many standard security approaches such as downloading software patches.

It is fairly easy to gain access to registers on a LINUX host. Typically all that is needed is a shell account, a debugger such as gdb, and the rights to the process in question. If a shell account is compromised and a process is run from that account then typically the rights to the registers related to that process are already in place. Given the process above, PID 26237, the debugger is started below and attached to that PID. Once the debugger prompt is attained then information about the registers can be displayed. Below the address of the program counter contained the instruction to be executed next printed as an integer in hexadecimal format. Note the reference to the register "rdi" which is used to pass the first argument to a function. Also of interest is the "pthread" instruction. This virtual host uses SMP (shared memory multiprocessing) and splits up the java code into a series of lightweight processes or threads, which allows the work to be distributed across multiple processors. So eventually, the contents of each thread needs to be re-assembled via a series of joins.

```
CRL\den.gus@os:/proc/26237$ gdb --pid 26237

(gdb) x/i $pc

=> 0x7f4082ccb4db <pthread_join+171>:   cmpl   $0x0,(%rdi)
```

Overwriting values within registers is also fairly easy to do. In the example below the contents of the register "xmm1" are printed from the debugger. That register is used to pass arguments between and among registers. Although most of its structures are floating point the last structure is a 128-bit integer. That value is modified using the debugger set command to 4095, which equals fff in hexadecimal.

```
(gdb) print $xmm1
$2 = {v4_float = {3.57331108e-43, 0, 0, 0}, v2_double = {
    1.2598673968951787e-321, 0}, v16_int8 = {-1, 0 <repeats 15 times>},
  v8_int16 = {255, 0, 0, 0, 0, 0, 0, 0}, v4_int32 = {255, 0, 0, 0},
```

```
  v2_int64 = {255, 0}, uint128 = 255}r15

(gdb) set $xmm1.uint128 = 0xfff

(gdb) print $xmm1
$3 = {v4_float = {5.73831721e-42, 0, 0, 0}, v2_double = {
    2.0231988197199046e-320, 0}, v16_int8 = {-1, 15, 0 <repeats 14 times>},
  v8_int16 = {4095, 0, 0, 0, 0, 0, 0, 0}, v4_int32 = {4095, 0, 0, 0},
  v2_int64 = {4095, 0}, uint128 = 4095}
```

Changing values in a register can have a detrimental effect on program execution. Besides changing the data itself, which would cause inaccurate results, the order of instruction execution can be changed as well. Modifying the order means that some instruction may never get executed and the program may hit an unexpected end either immediately or after several iterations of its logic set. It is also possible to use registers to find out information about other processes. In particular the step through capability of the debugger can be used to search for a particular piece of information within the registers, which is often applied by hackers to launch a buffer overflow attack. These vulnerabilities plus other security considerations related to registers will be presented in more detail in a subsequent section of this paper.

2 Review of Literature

2.1 Overview

A large number of vulnerabilities related to memory management have been documented. To get a sense of the types of vulnerabilities the common vulnerabilities and exposures (CVE) list reveals hundreds of active vulnerabilities of severity scores greater than seven (on a 10 point scale) (CVE, 2016). For more information, Chen et al. (2011) provides more detail in regard to the type of vulnerabilities that have been observed and how effectively they have been exploited. Part of the problem in mitigating such attacks is that the solutions tend to be attack specific, which means that no one solution is totally effective. Further, they found that no efficient techniques exist to deal with semantic vulnerabilities related to violations of high-level security invariants.

Overwriting memory segments has also been identified as problematic in applications and in some cases no protection mechanisms are in place. In fact, Ferreira et al. (2012) found that the most crucial software running on a host, the operating system (OS) is often left relatively unprotected in main memory. The OS is at the top of the software hierarchy and therefore resiliency is critical. The literature indicates that there are still significant memory errors in the memory areas supporting the OS (Levy, 2015). In the case of this paper, this is important because the focus is on registers used to support an application and some of the registers may well reside in a low area of main memory.

Memory management is carried out in the operating system and the core of the operating system is the kernel. How rights are managed in this regard is pertinent to assessing the

probability of any memory attack being successful. For the purposes of this paper the prime area of interest would be the registers. Often it is believed that root access is the universal key to data throughout a host. Gaining root access via sudo does not mean that rights have been attained in kernel space. For more on this topic see Stack Overflow (2016). This design technique makes it more difficult to compromise data stored in kernel-managed memory. However, it is not impossible particularly if the hacker can recompile objects called by the kernel or the kernel itself. The recognized history of this problem dates back at least ten years (Criswell, Geoffray & Adve, 2009). Subsequent research confirms this is still problematic, for example, Xu et al. (2015) depicts a specific problem in that regard and explains how the kernel and its associated memory segment can be compromised. In fact the sophistication of the attacks have 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. This technique has been used with register attacks to find a particular memory segment (Aleph One, 2016). The fact that successful attacks against the stack (which contains the instruction set) and its associated memory segments are taking place strengthens the need to understand such attacks which is the purpose of this paper. Further, the general review offered herein is just the tip of the iceberg. There are many popular sites such as: Project Zero (2015) that provide detailed information about the various types of attacks, their effectiveness and how to recreate them in a test environment so that they can be studied in detail. In particular buffer overflows are often considered the most effective and should receive careful monitoring (Avijit, Prateek Gupta & Gupta, 2004). However, in this paper concerns related to denial of service carried out via registers within the java runtime will be used to frame the basic problem and then a specific buffer overflow strategy will follow.

2.2 Attacks Against Registers

Attacks using registers are not new, in fact a representative example of a buffer overflow attack is shown by Stack Exchange (2017). In this attack the EBP (base pointer) register is used in an attempt to create an overflow. The attacks can take a wide variety of form even involving perhaps the most dangerous area of memory related to kernel space. Lee, Ham, Kim and Song (2015) depict an attack against kernel space that takes advantage of a 64-bit register. This is disturbing because kernel level rights are the top of the hierarchy and a process owned by the kernel may not even be killable by the root itself. The danger of this topic is further discussed in Xiao, Huang, and Wang (2010) in which they show how easy it is to find memory addresses related to kernel functions and exploit them through byte or register manipulation. Although there have been some measures established to thwart this type of activity it still remains a threat (Rielly et al., 2008). Registers have also been used successfully in compromising cryptographic keys. Genkin, Pachmanov, Pipman, Shamir, and Tromer (2016) describe the use of side channel attacks for reading internal registers which in turn leads to extraction of the keys. This is also disturbing because side–channel attacks are widely used in cloud architectures and have allowed a virtual machine sharing hardware with other virtual machines to snoop on the register, memory and bus levels.

This process of attacking the call stack and hijacking the program control flow, which allows the attacker to execute strategically chosen machine instructions, has been termed "Return-oriented programming" (Buchanan, Roemer, Shacham & Savage, 2008). There have been numerous strategies employed to try to stop such attacks see: Francillon, Perito and Castelluccia, 2009; Li, Wang, Jiang, Grace and Bahram, 2010; and Pappas, 2012 just to name a few. Accordingly methods have appeared that specifically target preventing this problem in clouds as well (Zhou, Reiter & Zhang, 2016).

Since the purpose of this paper is to provide students with a foundation to understand this problem using the tools within the LINUX operating system will be the prime methodology. The focus will be on the intersection of the LINUX commands, finding the memory/register address and using debug to view/modify the appropriate components to illustrate the ease with which a denial of service attack can be launched.

3 Methodology

3.1 Blueprint of a Simple DOS Attack Against a Register

Using the process from the example in the first part of the paper, PID 26237, a blue print can be created in regard to how overwriting a value in a register can create a denial of service attack. As before the debugger is attached to PID 26237 and then the address of the program counter is printed off and by adding the /i the instruction appears as well.

```
(gdb) p/x $pc

$1 = 0x7f4082ccb4db

(gdb) x/i $pc

=> 0x7f4082ccb4db <pthread_join+171>:   cmpl   $0x0,(%rdi)
```

Disassembling the code can be accomplished by simply entering the "disas" command into the debugger. Note that the hexadecimal memory address, the function and the register involved appear, %r15 through %r12 are general-purpose registers and could not be directly guessed as being used.

```
(gdb) disas

Dump of assembler code for function pthread_join:

   0x00007f4082ccb430 <+0>:     push   %r15

   0x00007f4082ccb432 <+2>:     push   %r14

   0x00007f4082ccb434 <+4>:     push   %r13

   0x00007f4082ccb436 <+6>:     push   %r12

   0x00007f4082ccb438 <+8>:     push   %rbp
```

```
   0x00007f4082ccb439 <+9>:     push   %rbx
```

A look at the memory address range can be obtained by dumping the beginning and ending +1 address to a file. In the example below it is being written to a file called "regmem" in the user's home directory.

```
(gdb) dump memory ~/regmem 0x00007f4082ccb430 0x00007f4082ccb43a
```

The contents of that file can be in turn viewed by using the hexadecimal dump command in LINUX "xxd". In the example below one can see that the hex equivalent of the push function using the %r15 register is 41 57.

```
CRL\den.gus@os:/proc/26237$ xxd ~/regmem | more

0000000: 4157 4156 4155 4154 5553 4889 fb48 83ec  AWAVAUATUSH..H..

0000010: 2848 85ff 0f84 1a01 0000 8b87 d002 0000  (H.............

0000020: 85c0 0f88 0c01 0000 4839 bf28 0600 00b8  ........H9.(....
```

A more effective way of determining these values would be to use the logic contained in the example below. In this example, the "disas" command is again utilized but the /r allows the hex function values to be printed as well. Note the starting address is needed along with how many bytes past the starting address should be displayed. Changing that 41 value to something meaningless, perhaps just null values (hex 00) would result in a denial of service attack because the push would not take place. Also, a push function could easily be changed to a jump function, which could potentially pass control to some other instruction set that might provide a path to compromise sensitive data.

```
(gdb) disas /r  0x00007f4082ccb430, +10

Dump of assembler code from 0x7f4082ccb430 to 0x7f4082ccb43a:

   0x00007f4082ccb430 <pthread_join+0>: 41 57   push   %r15

   0x00007f4082ccb432 <pthread_join+2>: 41 56   push   %r14

   0x00007f4082ccb434 <pthread_join+4>: 41 55   push   %r13

   0x00007f4082ccb436 <pthread_join+6>: 41 54   push   %r12

   0x00007f4082ccb438 <pthread_join+8>: 55      push   %rbp

   0x00007f4082ccb439 <pthread_join+9>: 53      push   %rbx

End of assembler dump.
```

The example below is designed to illustrate how easily the value in a register can be overwritten. To illustrate this a commonly used register for floating point arguments, xmm, was selected as the target. Specifically the structure within the register uint128 was

designated to be over-written. Its initial value as can be seen from the gdb command print $xmm1 is 255.

```
(gdb) print $xmm1

$2 = {v4_float = {3.57331108e-43, 0, 0, 0}, v2_double = {

    1.2598673968951787e-321, 0}, v16_int8 = {-1, 0 <repeats 15 times>},

  v8_int16 = {255, 0, 0, 0, 0, 0, 0, 0}, v4_int32 = {255, 0, 0, 0},

  v2_int64 = {255, 0}, uint128 = 255}r15
```

The set command is used to change this value to 0xfff which would be 4095 in decimal notation.

```
(gdb) set $xmm1.uint128 = 0xfff
```

When running the prior print command we can see that the new value is now 4095 as expected. Once again this was done by attaching the debugger to the process related to the register values (PID 26237) and the debugger run with an account that had rights to that process.

```
(gdb) print $xmm1

$3 = {v4_float = {5.73831721e-42, 0, 0, 0}, v2_double = {

    2.0231988197199046e-320, 0}, v16_int8 = {-1, 15, 0 <repeats 14 times>},

  v8_int16 = {4095, 0, 0, 0, 0, 0, 0, 0}, v4_int32 = {4095, 0, 0, 0},

  v2_int64 = {4095, 0}, uint128 = 4095}
```

To ascertain the affect tampering with registers might have on an application, a java socket call service was used. The characteristics of this service are described in the Introduction of this paper with the "netstat –apeen" command which indicated it was PID 26237, was running on port 18002 and was started by the java runtime command. Its class name is TempServer and it is simply designed to convert a Fahrenheit temperature reading into Centigrade. To illustrate the problem the registers %r12 - %r15 were overwritten with 0xfff. Interestingly, the service continued to run for two iterations (connections), but failed on the third try. The comments provided by the java runtime are useful in understanding the problem that resulted. First, the signal "SIGSEGV" indicates an invalid memory reference. This means that the java application attempted to de-reference a memory address that has not been defined. This makes sense because of the bogus value of 0xfff inserted into the target registers. Second, the address of the program counter (%pc), pc=0x00007f4082cd0350 is near the address range depicted when the code was dis-assembled in the previous section of this paper. It is worth noting that the actual beginning memory address changes each time that the service is instantiated. This can be explained in part because the code is being run in a virtual environment. For

example, in a subsequent server session the code began at: 0x00007ff692817430. Whereas, in the session before the address started with 00007f4. Therefore, without the ability to extract mapping information via the operating system it would be difficult to find the specific relative memory address of a register.

Third, the PID 26237 is printed which verifies this is in fact the code that was started at the beginning of the paper. Fourth, the tid (thread ID) indicates that shared memory multiprocessing was being used. The results of the "ps –ALF" command (displayed below) shows that when the class TempServer is run it is split into 14 lightweight processes (threads). Below, only the first two are displayed and this output was generated in a subsequent experiment and so the root PID is 15188 and not 26237. So therefore, one can view the TID as a specific process in a group of related processes tied to the original process ID.

Fifth, the problematic frame appears to deal with a thread library module (libpthread.so.0+0xe350). Specifically, the instruction that leads to the failure is _pthread_cleanup_pop+0x0. This makes sense given that the java code is broken up into 14 lightweight processes or threads. Last, the potentially valuable diagnostic information is being suppressed because the core dump option is disabled. For complex analysis related to detecting a security threat the core dump option could be quite valuable. Also, there is a LINUX "strace" command that could also be useful in this regard as well. In either case the starting point could be gleaned by searching for the SIGEGV occurrence.

```
CRL\den.gus@os:~/javaclass$ java TempServer
Waiting for connection....

Connection 1 recived from: localhost
Temp in F: 44
Waiting for connection....

Connection 2 recived from: localhost
Temp in F: 33
Waiting for connection....

Connection 3 recived from: localhost
Temp in F: 55
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00007f4082cd0350, pid=26237, tid=139915053430528
#
# JRE version: OpenJDK Runtime Environment (7.0_111-b01) (build 1.7.0_111-b01)
# Java VM: OpenJDK 64-Bit Server VM (24.111-b01 mixed mode linux-amd64
compressed oops)
# Derivative: IcedTea 2.6.7
# Distribution: Debian GNU/Linux 8.6 (jessie), package 7u111-2.6.7-2~deb8u1
# Problematic frame:
# C  [libpthread.so.0+0xe350]  _pthread_cleanup_pop+0x0
```

```
#
# Failed to write core dump. Core dumps have been disabled. To enable core
dumping, try "ulimit -c unlimited" before starting Java again
#
# An error report file with more information is saved as:
# /home/den.gus/javaclass/hs_err_pid26237.log
#
# If you would like to submit a bug report, please include
# instructions on how to reproduce the bug and visit:
#   http://icedtea.classpath.org/bugzilla
#
Aborted


CRL\den.gus@os:~$ ps -ALF | grep java
BCRL\de+ 15188 15182 15188  0   14 558740 25228  1 10:11 pts/1    00:00:00 java
TempServer
BCRL\de+ 15188 15182 15189  0   14 558740 25228  1 10:11 pts/1    00:00:00 java
TempServer
```

3.2 Blueprint of an Attack to Find a Memory Address from a Register

To illustrate how a register containing a memory address can be used to find data in memory, the following C program is used which accepts input data via the keyboard (default input device).

```
CRL\den.gus@os:~$ ./add
Enter the number of integers you want to add
4
Enter 4 integers
444444
444444
444444
```

Based on the authors' previous experience with this type of code the source index register (rsi) is potentially a prime source for obtaining the relative memory address of the input data. If one has user level rights for the program a debugger (gdb) can be used to view the address contained in the register. However, the PID of the program (15285) must be obtained first so the debugger can be attached to that process. After that the "info registers" command displays the address currently in rsi.

```
CRL\den.gus@os:/proc$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 1018168411 14755 14646  0 80 0 - 12974 -    pts/1    00:00:19 gdb
0 S 1018168411 15285 14512  0 80 0 - 1021 -     pts/0    00:00:00 add


(gdb) info registers
rax            0xfffffffffffffe00       -512
rbx            0x7f7de5c384e0   140178702435552
rcx            0xffffffffffffffff       -1
```

```
rdx            0x400    1024
rsi            0x7f7de5e59000   140178704666624
```

So if one uses debug to copy a memory range starting with the address provided by rsi and then uses the hexadecimal dump command (xxd) to view the contents of that memory range which was saved to disk then the last value entered from the keyboard is displayed. Note that it pulls numbers in ASCII form so 34 hex is equal to a 4 in ASCII. So the results are a string of six 4s plus the 0a hex indicates end of line in ASCII.

```
(gdb) dump memory ~/regmemadd 0x7f7de5e59000 0x7f7de5e5d000

CRL\den.gus@os:/proc/15285$ xxd ~/regmemadd | grep 44

0000000: 3434 3434 3434 0a00 0000 0000 0000 0000  444444..........
```

Suppose that the potential attacker did not have experience with how different languages used memory/registers. A little knowledge of the UNIX (in our case LINUX) operating system could be used to reverse engineer the addressing process. In other words use the memory address to find which register is using it. To evaluate where data is stored from keyboard input the “strace” (system trace) command in LINUX can be used. In the example below the memory map command “mmap” indicates that a read is to follow and the results will be stored in relative address 0x7fc544869000.

```
strace ./add

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fc544869000
read(0, 4
"4\n", 1024)                    = 2
write(1, "Enter 4 integers\n", 17Enter 4 integers
)      = 17
read(0, 5
"5\n", 1024)                    = 2
read(0, 55
"55\n", 1024)                   = 3
read(0, 5
"5\n", 1024)                    = 2
read(0, 5
"5\n", 1024)                    = 2
write(1, "Sum of entered integers = 70\n", 29Sum of entered integers = 70
) = 29
exit_group(0)                           = ?
+++ exited with 0 +++
```

Unfortunately once “strace” is attached to the process started by the C program the debugger cannot also be attached to that process. However, the maps file in /proc/”processID” can provide a list of unnamed (anonymous) memory segments.

```
warning: process 1690 is already traced by process 1687
```

In the example below we find the range 7f7de5e59000-7f7de5e5d000 that we found above to be tied to rsi. It is worth noting that one is working within the bounds of a

dynamic system. This means that the process IDs will change each time the program is run as well as the relative memory addresses. To an extent this helps from a security perspective, but the trends of using similar memory addresses and the placement of the anonymous memory segments within the memory maps still makes it relatively simple to find where target data may reside in memory. Of course to some degree the fact that cloud computing coupled with virtualization was the environment utilized explains the variation of memory addresses.

```
CRL\den.gus@os:/proc$ cd 15285
CRL\den.gus@os:/proc/15285$ cat maps
00400000-00401000 r-xp 00000000 00:21 2228254
/home/den.gus/add
00600000-00601000 rw-p 00000000 00:21 2228254
/home/den.gus/add
7f7de5e59000-7f7de5e5d000 rw-p 00000000 00:00 0
```

Conclusions

Too often students have unrealistic expectations in regard to how attacks take place and how to prevent them. Unfortunately, there is often a mindset that downloading a software patch can provide universal protection for an application (or at least until the next patch is released). Also particularly in a cloud environment symbolic links and other types of canonical names often shield students from the actual data. When remediating complex security problems there is often a need to evaluate primary data. That data, of course, may well be contained in memory or registers. The trick may be to navigate the pointers to get to that data. The examples used herein have shown that the LINUX operating system often makes mapping available to do so. Because the limitation of viewing this mapping is related to user access rights, compromising a user that owns an application allows access to the mapping. Granted the dynamic nature of LINUX complicates guessing where sensitive data may reside in memory, but if the rights are obtained the mapping provided by LINUX makes it fairly easy to access the memory segment.

There were two primary examples depicted herein in the first case it was shown that overwriting the values in a register could very easily disrupt a service. This disruption could occur either immediately or in a delayed fashion depending on when that register is actually called. In the second case the register was used to find the memory address of potentially sensitive data. Unfortunately, these examples just represent the tip of the iceberg. The literature review presented several more dangerous scenarios. However, the examples presented provide a foundation for understanding how the more complex scenarios could take place. It is also clear that the LINUX operating system is ripe with tools that while intended to make system administration more efficient could be used to gain information, which could be used to launch memory/register attacks. This high level of functionality within the operating system even though it can be misused provides an excellent teaching tool in understanding how memory/register attacks can take place. The scenarios offered reinforce concepts such as memory mapping, relative memory addressing, registers and process management. Often, students grasp these concepts if

framed in the context of a larger problem. Of course in this paper the larger context was an attack against memory/registers.

In terms of protection against such attacks it is critical not to provide access to the cloud, zone and host in which the memory/register target is housed. This can begin with sound firewalling techniques and security layering within the cloud. Writing applications using sound software security techniques is also critical. For example, object orient programming languages provide a wealth of classes that catch exceptions that can prevent buffer overflow attacks, but alas they are not always implemented. The prime purpose of this paper was to use the LINUX operating system to illustrate the basic problems related to memory/register attacks. It was felt that the first step for students was to gain a basic operational understanding of the problem. It is expected that further investigation would be needed to be truly effective in preventing the problem.

References

Abdul, R., Guster, D., & Schmidt, M. (2017). Application Level Memory Management Strategies via the "Garbage Collector": Performance and Security Ramifications. This paper is to be presented at the 2017 Midwest Instructional Computing Symposium.

Aldrich, J. (2013). Why Objects Are Inevitable, Onward!. Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, 101-116.

Aleph One (2016). Smashing the Stack for Fun and Profit. http://insecure.org/stf/smashstack.html.

Avijit, K., Gupta, P., Gupta, D. (2004). TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. Proceedings of the 13th Conference on USENIX Security Symposium, p.4-4, August 09-13, 2004, San Diego, CA.

Buchanan, E., Roemer, R., Shacham, H., & Savage, S. (October 2008). When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. Proceedings of the 15th ACM conference on Computer and communications security - CCS '08 *(PDF*). pp. 27–38. doi:10.1145/1455770.1455776. ISBN 978-1-59593-810-7.

Callum, C., Singer, J., & Vengerov, D. (2015). The Judgement of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes. ISMM: Proceedings of the 2015 International Symposium on Memory Management, 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. Proceedings

of the Second Asia-Pacific Workshop on Systems, July 11-12, 2011, Shanghai, China [doi>10.1145/2103799.2103805].

Criswell, J., Geoffray, N. & Vikram, A. (2009) Memory Safety for low-level Software/Hardware Interactions. Proceedings of the 18th conference on USENIX security symposium, p.83-100, Montreal, Canada.

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

Eazynotes (2016). http://www.eazynotes.com/pages/computer-system-architecture/computer-registers.html.

Ferreira, K. B., Pedretti, K., Bridges, P. G., Brightwell, R., Fiala, D., & Mueller, F. (2012). Evaluating Operating System Vulnerability to Memory Errors. ROSS 2012: Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers [Workshop Papers].

Francillon, A., Perito, D., & Castelluccia, C. (2009). Defending Embedded Systems Against Control Flow Attacks. In Proceedings of SecuCode 2009, S. Lachmund and C. Schaefer, Eds. ACM Press, 19–26.

Genkin, D., Pachmanov, L., Pipman, I., Shamir, A., & Tromer, E. (2016). Physical Key Extraction Attacks on PCs. Communications of the ACM, Vol. 59 No. 6, Pages 70-79.

Guimaraes, J. (1995). The Object Oriented Model and its Advantages. ACM SIGPLAN OOPS Messenger, Volume 6, Number 1, 40-49.

Lee, J., Ham, H., Kim, I. & Song, J. (2015). Poster: Page Table Manipulation Attack. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Pages 1644-1646.

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 (IJHPCA).

Li, J., Wang, Z., Jiang, X., Grace, M., & Bahram, S. (2010). Defeating Return-Oriented Rootkits with "Return-Less" Kernels. In Proceedings of EuroSys 2010, G. Muller, Ed. ACM Press, 195–208

Pappas, V. (2012). kBouncer: Efficient and Transparent ROP Mitigation. April 2012. http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf.

Petsios, T., Kemerlis, V. P., Polychronakis, M., & Keromytis, A. D. (2015). DynaGuard: Armoring Canary-based Protections against Brute-force Attacks. Proceedings of the 31st Annual Computer Security Applications Conference, December 07-11, 2015, Los Angeles, CA, USA. [doi>10.1145/2818000.2818031].

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

Riley, R., Jiang, X., & Xu, D. (2008). Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing, Proceedings of the 11th international Symposium on Recent Advances in Intrusion Detection, September 15-17, 2008, Cambridge, MA, USA. [doi>10.1007/978-3-540-87403-4_1].

Stack Exchange (2017). Buffer Overflow and Register Contents. http://security.stackexchange.com/questions/89139/buffer-overflow-and-register-contents.

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

Xiao, J., Huang, H., & Wang, H. (2010). Kernel Data Attack is a Realistic Security Threat. Security and Privacy in Communication Networks Volume 164 of the series Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering pp 135-154.

Xu, Wen et al. (2015). From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 414-425.

Zhou, Z., Reiter, M. K., & Zhang, Y. (2016). A Software Approach to Defeating Side Channels in Last-level Caches. arXiv preprint arXiv:1603.05615, 2016.