Article can be accessed online at http://www.publishingindia.com

Security Vulnerabilities of Registers in LINUX Hosts: Buffer Overflow and Service Disruption Concerns

Dennis Guster*, Erich Rice*, Hazem Farra**

Abstract

Security has become extremely important in the information technology field. Often times the most important resource a company has is the data that it has diligently gathered, the loss or deletion of which could cause the failure of the organization. With the advent of Cloud Computing and the use of shared or Colo (colocation) hardware this has become of even greater concern to organizations. This paper looks at ways in which the LINUX operating system and various software tools can be utilized to shed light on potential vulnerabilities, especially how memory is stored at the base layers of the operating system. The main focus is on the registers, and how certain LINUX based tools such as a debugger can be used to determine where memory resides and how it could potentially be attacked, changed, or deleted. Also, the paper discusses how these various techniques and utilities could be used to provide IT professionals with a better understanding of how these attacks could occur as well as the level of sophistication needed to deal with and prevent them.

Keywords: Cloud Computing, LINUX, Registers, Security, Virtualization

Introduction

Cloud computing and virtualization have changed the focus of computer security techniques from just protecting the data on secondary storage and the network to deep concern about the contents of memory as well. Often data is shared in memory which is needed to provide timely responses to inquiries. Chen et al.(2011) is a great source in understanding the common vulnerabilities of memory access. While object oriented programming (OPP) has numerous advantages it presents dangerous vulnerabilities too. These vulnerabilities are related to verifying that the objects used are legitimate and the mapping of objects is not disrupted. The basic concept of OPP, which creates objects, and facilitates the reuse of code is well documented by Guimaraes (1995). Also important within a cloud architecture is the fact that OPP stresses the importance of interoperability. This leads to cost effective and efficient performance. Further, it is imperative because of the diversity of applications that share information within today's cloud based computing architectures (Aldrich, 2013). In order for OPP to function properly it is important that the location of the objects, the memory segments containing the data as well as the registers be managed effectively. This modular approach requires an "index" of where the objects will reside in memory. In languages such as java and python, memory objects reside in the heap (Callum, Singer & Vengerov, 2015). Abdul, Guster, and Schmidt have demonstrated vulnerabilities within the heap by launching successful denial-of-service attacks against the "heap" (2017). Further, they demonstrated that if the strength of the object pointers within the heap get compromised the whole class would no longer run.

For memory and register related attacks to be successful the hackers need to obtain direct memory access (Buffer Overflow, 2016). By using basic tools on a LINUX host the memory within the segment for the heap can be easily found. To illustrate this, an example is included below

* Department of Information Systems, St. Cloud State University, St. Cloud, Minnesota, United States of America.
E-mail: dguster@stcloudstate.edu
* Department of Information Systems, St. Cloud State University, St. Cloud, Minnesota, United States of America.
E-mail: eprice@stcloudstate.edu
** Department of Information Assurance, St. Cloud State University, St. Cloud, Minnesota, United States of America.
E-mail: faha1301@stcloudstate.edu

for a java class entitled "TempServer" which is running under process ID 2514. A partial memory map for this process is also depicted below:

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

tcp 0 0 0.0.0.0:17002
0.0.0.0:* LISTEN 1018168 2440358
2514/java

CRL\den.gus@os:~/javaclass$ cd /
proc/2514

CRL\den.gus@os:/proc/2514$ cat maps

00400000-00406000 r-xp 00000000
08:01 915810 /usr/lib/jvm/
java-7-openjdk-amd64/jre/bin/java

00829000-0084b000 rw-p 00000000 00:00
0 [heap]
```

This entry reveals that the heap resides at relative memory address range: 00829000-0084b000. Just like with files the LINUX operating system assigns access flags to memory. In the example above the permissions are set to read, write and private (hidden from other classes within the package). The write flag must be in place so that the object map can be updated. This creates some degree of vulnerability. One way of looking at this is that if a hacker could overwrite or change the map entries then the whole class (or any another class that calls this class) could be compromised as well. Certainly the vulnerabilities described related to the heap are disturbing. However, there are other layers of abstraction below the heap such as registers that could also be compromised too.

A register can be defined as a small segment of very fast memory. Of course registers are very closely tied to the central processing unit. The goal of this design is to speed up execution of instructions by providing quick access to addresses, instructions and values. Registers can be viewed as the root of the memory hierarchy and typically offer the fastest means to manipulate data. However, like many other types of memory, registers are volatile and their contents are lost if the computer is powered down. Registers can be used to store temporary data while a program executes. In order for both data and instructions to be integrated 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 a shortcut from which to compromise a computer system despite the best efforts of security personnel. Further it may well be a method below the radar and easily circumvent many standard security approaches such as downloading software patches.

As stated earlier gaining access to registers on a LINUX host can be easily accomplished. In many cases, all that is needed is a shell account, a debugger such as gdb, and the rights to the process in question. In regard to obtaining rights, when 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. Tying back to the process above, PID 2514, the debugger attached below to that PID. From the debugger prompt information about the registers can be displayed. In the example below, the address of the program counter contained the instruction to be executed next printed as an integer in hexadecimal format. Also note the reference to the register "rdi" which can be used to pass the first argument to a function. From a cloud perspective the presence of the "pthread" instruction is interesting and expected. 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. It is worth noting the compiler used to generate the code must support this option. Within the LINUX host utilized, SMP was supported by the java complier but not the C compiler.

```
CRL\den.gus@os:/proc/26237$ gdb --pid
2514

(gdb) x/i $pc

=>0x7f4082ccb245 <pthread_join+170>:
cmpl $0x0,(%rdi)
```

Unfortunately it is easy for hacker to overwrite values within registers. Below, the contents of the register "xmm1" are displayed from the debugger. The purpose of that register is to pass arguments between and among registers. While most of its structures are floating point the last structure is a 128-bit integer. That value can be modified using the debugger set command to 256, which equals 100 in hexadecimal.

```
(gdb) print $xmm1
```

```
$2 = {v4_float = {3.57221108e-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 = 0x100


(gdb) print $xmm1

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

    2.023198819046e-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 = 256}
```

Besides changing a value itself, modifying values in a register can have a detrimental effect on program execution. Often the the order of instruction execution ends up being modified too. Changing the order means that some instructions may never get executed and the program may hit an unexpected end. This can happen immediately or after several iterations of its logic set. Hackers could also 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. This is described in the first example in the methodology section. This vulnerability plus an attack on a service created by a java socket call program will be presented in more detail in a subsequent section of this paper.

Review of Literature

Overview

Many vulnerabilities related to memory management have been documented previously. To get a better sense of the types of vulnerabilities that are prevalent one can look to the common vulnerabilities and exposures (CVE) list. The list reveals hundreds of active vulnerabilities of severity scores greater than seven (on a 10 point scale) (CVE, 2016). To provide more detail, Chen et al. (2011) gives a detailed description of the type of vulnerabilities that have been observed and how effectively they have been exploited. The problem in mitigating such attacks is that the solutions tend to be attack specific (i.e. only relevant to one attack scenario), which often means that no one solution is 100% effective. Further, it has been found that no efficient technique exists to deal with semantic vulnerabilities related to violations of high-level security invariants.

The overwriting of memory segments has previously been identified as a problem; especially in applications and in some cases no protection mechanisms are found to be in place. In fact, Ferreira et al. (2012) found that the most critical software running on infrastructure, the operating system (OS) is often relatively unprotected within the main memory. The OS is at the top of the software hierarchy and therefore its resiliency is of primary concern. The reviewed literature shows that there is still the potential for significant memory errors supporting the OS (Levy, 2015). In the case of this paper, it is very important because the primary focus is on the registers used to support an application and some of the registers often reside in a lower area of the main memory.

The main memory management is carried out within the operating system and the core of the OS is the kernel. How the various rights are managed in this regard is very pertinent to assessing the potential probability of a memory attack being successful in nature. For this paper the prime area of concern would be the registers. It is often times believed that root access is the primary key to accessing data within a given host. The gaining of root access via sudo (super user do) does not however mean that rights have been attained within the kernel space. For more on this topic seethe Stack Overflow article (2016). This design technique makes it far more difficult to compromise data stored within the kernel-managed memory. However, it is not impossible especially if the potential hacker can recompile related objects that are called by the kernel. The recognized history of this potential problem dates back at least ten years (Criswell, Geoffray & Adve, 2009). Subsequent research confirms that this is still a problem, for example by the research

in Xu et al. (2015) which depicts a specific problem in that regard and explains how the kernel and its associated memory segment can potentially be compromised. The sophistication of the attacks have gone far beyond a brute force random number design to a more sophisticated "collision" technique that can be used to hone in on a particular memory segment without requiring access to a system generated memory map. This technique has been used with known register attacks to ascertain a certain memory segment (Aleph One, 2016). The possibility that successful attacks against the stack (which contains the instruction set) and its associated memory segments are taking place makes the need to understand such attacks even more important, and is the purpose of this paper. Furthermore, the general view offered herein is just the tip of the vulnerability iceberg. There are many popular sites such as: Project Zero (2015) that provide detailed information about the various types of attacks currently in the wild, and their effectiveness and how to recreate them in a test environment so that they can be studied in more detail. In many ways buffer overflow attacks are often considered the most effective form of attack and should therefore receive even more careful monitoring (Avijit, Prateek Gupta & Gupta, 2004). However, in this paper related concerns to potential denial of service (DoS) carried out via the registers within the java runtime will be used to frame the basic problem and then a more specific buffer overflow strategy will follow.

Attacks Against Registers

Attacks using the registers are not new, and in fact a representative example of a buffer overflow attack is shown in the Stack Exchange article (2017). In this attack the EBP (base pointer) register is used in an attempt to create a overflow attack. The attacks can take a wide variety of forms and even involve perhaps the most dangerous area of memory related to the kernel space. Lee, Ham, Kim and Song (2015) depict an attack against kernel space that takes advantage of a 64-bit register. This is most disturbing considering that the kernel level rights are the top of the hierarchy and a process owned by the kernel may not even be able to be killed by the root itself. The danger of this potential issue is further explained within Xiao, Huang, and Wang (2010) in which they show the ease with which memory addresses are found related to kernel functions and can be exploited by them through byte or register manipulation. Although there have been measures established to thwart this type of potential activity it remains a grave threat (Rielly et al., 2008). The registers have also been used in successful attacks to compromise cryptographic or encryption keys. Genkin, Pachmanov, Pipman, Shamir, and Tromer (2016) have described the use of so called side-channel attacks for reading internal registers which in turn can lead to the extraction of the encryption keys. This is disturbing because these side-channel attacks are often widely used in cloud computing architectures and have allowed the compromise of a virtual machine which is sharing hardware with other virtual machines to snoop on the register, memory and bus levels.

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

Since the purpose of this paper is to provide IT professionals with a foundation in understanding this problem, as well as instruction in using the tools within the LINUX operating system, the interaction between registers/memory and the LINUX operating system will be the thrust of this paper. The focus will be on the intersection of the LINUX commands, finding the memory/register address and using the debug utility to view/modify the appropriate components to illustrate the ease with which a denial of service or compromising sensitive data attack could be launched successfully.

Methodology

Blueprint of a Simple Buffer Overflow Attack Facilitated by a Register Address

To analyze the stack and how the program is structured within the memory, the authors created the simple C code below:

```
#include <stdio.h>
```

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
voidGetUserInput()
{
      // define 8 byte-long char variable
      char buffer[8];
      // read the user input from stdin
            gets(buffer);
      // write user input to stdout
            puts(buffer);
}
int main()
{
      // Get input from user
      GetUserInput();
      return 0;
}
```

The code was compiled using the gcc complier in 32 bit mode and switches to facilitate debug mode were added. The stack boundaries are aligned in 8 byte intervals and the executable created is named bof32.

```
$ gcc -ggdb -m32 -mpreferred-stack-boundary=2 -o bof32 bof32.c
```

To gain an understanding of the register/memory assignments the debugger (gdb) is attached to the executable which is stored in the current directory.

$ gdb ./bof32

The first step is to list the code which will allow the instructions to be viewed and break points to be set up.

```
(gdb) list 1
1     #include <stdio.h>
2     #include <unistd.h>
3     #include <stdlib.h>
4     #include <string.h>
5
6
7     void GetUserInput()
8     {
9     // define 8 byte-long char variable
10    char buffer[8];
11    // read the user input from stdin
12    gets(buffer);
13    // write user input to stdout
14    puts(buffer);
15    }
16    intmain()
17    {
18          // Get input from user
19          GetUserInput();
20          return 0;
21    }
```

Sequentially, the program will start at the main() (line 16), proceed to GetUserInput() (line 19) and transfer the operation to that routine (line 7). At line 10, an 8-byte variable named "buffer" is created. This variable will hold the input from the user after the gets (buffer) command executes. Then, the content of buffer is printed to the screen (line 14) and the operation is returned back to the main() at line 20. To effectively illustrate our problem the break points will be set at line 19 and line 12.

```
(gdb) b 19

Breakpoint 1 at 0x80484cb: file bof32.c, line 19.

(gdb) b 12

Breakpoint 2 at 0x804849c: file bof32.c, line 12.
```

The goal of this paper is to illustrate how registers can be used to find memory locations of sensitive data. This process begins in the next step which illustrates where the instructions are located within the stack. The disas (disassemble) command allows the instructions locations within the memory to be obtained for the main and then the GetUserInput function.

```
(gdb) disas main

Dump of assembler code for function main:

0x080484c8 <+0>:      push   %ebp

0x080484c9 <+1>:      mov    %esp,%ebp

0x080484cb <+3>:      call   0x804848b <GetUserInput>

0x080484d0<+8>:       mov    $0x0,%eax

0x080484d5 <+13>:     pop    %ebp

0x080484d6 <+14>:     ret
```

```
End of assembler dump.
```

(gdb) disas GetUserInput

Dump of assembler code for function GetUserInput:

```
0x0804848b <+0>:     push   %ebp
0x0804848c <+1>:     mov    %esp,%ebp
0x0804848e <+3>:     sub    $0xc,%esp
0x08048491 <+6>:     mov    %gs:0x14,%eax
0x08048497 <+12>:    mov    %eax,-0x4(%ebp)
0x0804849a <+15>:    xor    %eax,%eax
0x0804849c <+17>:    lea    -0xc(%ebp),%eax
0x0804849f <+20>:    push   %eax
0x080484a0 <+21>:    call   0x8048340 <gets@plt>
0x080484a5 <+26>:    add    $0x4,%esp
0x080484a8 <+29>:    lea    -0xc(%ebp),%eax
0x080484ab <+32>:    push   %eax
0x080484ac <+33>:    call   0x8048360 <puts@plt>
0x080484b1 <+38>:    add    $0x4,%esp
0x080484b4 <+41>:    nop
0x080484b5 <+42>:    mov    -0x4(%ebp),%eax
0x080484b8 <+45>:    xor    %gs:0x14,%eax
0x080484bf <+52>:    je     0x80484c6
<GetUserInput+59>
0x080484c1 <+54>:    call   0x8048350 <__stack_chk_
fail@plt>
0x080484c6 <+59>:    leave
0x080484c7 <+60>:    ret
```

End of assembler dump.

An important address for the purpose of this experiment is at line number 4 of the disassembled main () code above (0x080484d0). This address will be used by the program to transfer control once the call to GetUserInput (line number 3) returns a value. This return value function (ret) is assigned to address 0x080484c7 above.

In normal operations, the RET address at line 4, 0x080484d0, will stay intact. However, in the study of buffer overflow problems, the address will be overwritten when the user input exceeds the variable size. In the example above that buffer boundary is set at of 8 bytes. To gain a better understanding of this process the program will be "stepped" through while observing the stack at register %esp (32 bit stack pointer).

Breakpoint 1, main () at bof32.c:19

19 GetUserInput();

To truly understand this process the stack needs to be examined before and after each step.

(gdb) x/8xw $esp

```
0xffffd0c8:  0x00000000  0 x f 7 e 1 1 6 3 7
0x00000001   0xffffd164

0xffffd0d8:  0xffffd16c  0 x 0 0 0 0 0 0 0 0
0x00000000   0x00000000
```

The RET at this stage has not been assigned and in its location we see a prior address: 0xffffd16c Once we hit the second break point, the RET will be set to the address mentioned above. It is highlighted in yellow below.

Breakpoint 2, GetUserInput () at bof32.c:12

12 gets(buffer);

(gdb) x/8xw $esp

```
0xffffd0b4:  0x0804821c  0 x 0 8 0 4 8 4 e 9
0x37d0fe00   0xffffd0c8

0xffffd0c4:  0x080484d0  0 x 0 0 0 0 0 0 0 0
0xf7e11637   0x00000001
```

The highlighted address above shows that the RET has been set on the stack for the program to use once the program finishes the execution of the GetUserInput() method.

At this point, the program will pause and wait for input. Once the character "A" is entered 18 times from the keyboard (the simulated sensitive data) the memory content of the stack (below) can be reexamined. The results indicate that the RET address has been overwritten (line 2 below).

```
(gdb) s
AAAAAAAAAAAAAAAAAA
(gdb) x/8xw $esp
0xffffd0b4:      0x41414141 0x41414141
0x41414141 0x41414141
0xffffd0c4:      0x08004141 0x00000000
0xf7e11637 0x00000001
Short term the program will still
continue executing by printing the input
out to the screen as in the following:
(gdb) x/8xw $esp
0xffffd0b4:      0x41414141 0x41414141
0x41414141 0x41414141
0xffffd0c4:      0x08004141 0x00000000
0xf7e11637 0x00000001
(gdb) s
AAAAAAAAAAAAAAAAAA
```

Stepping forward, however, causes the program to crash giving the message below. So in sum, the register could be used to provide the memory address of sensitive data as well as creating a denial of service at some future point.

*** stack smashing detected ***: /bof32 terminated

Program received signal SIGABRT, Aborted.

0xf7fd8be9 in __kernel_vsyscall ()

Blueprint of an Attack to Find a Memory Address from a Register in a Service

A socket call program can be used to create the server side of a client server model. If that service is compromised then information received or sent to the client can be read by a hacker. Part of this is due to the fact that the data sits in a buffer before being sent as well in a receive buffer upon receipt. To depict this problem in detail a C socket service side program will be used that looks up and returns the current date. While most data transmission between a client and server would be more complex this scenario illustrates the basic process. Once again the goal is to find a short cut to the memory address by using a register containing the memory address which can then be used to find data in memory.

The C program (sock) that returns the date is run and its process ID appears below along with the PID of the debugger (gdb) and a client session (nc) used to instantiate the inquiry. Once the inquiry takes place the returned date value is loaded into a buffer and sent. Typically, it remains in that buffer until the space is overwritten or relinquished by the process and reused. It is the goal of this example to view it before it is over written.

```
CRL\den.gus@os:~$ ./sock
CRL\den.gus@os:~$  cat  numbers.of  |
nclocalhost 6000
Sat Feb 11 15:42:58 2017
F S   UID   PID  PPID  C PRI  NI ADDR
SZ WCHAN  TTY          TIME CMD
0 t 1018168 6320 4664  0 80 0 -  1019
-      pts/1    00:00:00 sock
0 S 1018168 6335 4539  0 80 0 - 15058
-      pts/0    00:00:00 gdb
0 S 1018168 6407 4680  0 80 0 -  1164
-      pts/2    00:00:00 nc
```

Based on the authors' previous experience with this type of code the date will sit in a general purpose register. In the example below r8 through r15 are included in the register map. Of those registers r8, r9 and r13 offer the greatest potential because the memory is in the range one would expect for general purpose storage for a VM host. Also the content looks like data rather than a memory address. Therefore the three potential registers will be queried in sequence until the target data is found beginning with r8. 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 (6320) must be supplied to the debugger so it can attach to that process. After that the "info registers" command was used and the result depicted below.

The debug program provides register information we focus on r8:

```
(gdb) info registers
rax                0xfffffffffffffe00
-512
```

```
rbx             0x0      0
rcx          0xffffffffffffffff      -1
rdx             0x0      0
rsi             0x0      0
rdi             0x3      3
rbp      0x7ffdf034cbc0  0x7ffdf034cbc0
rsp      0x7ffdf034c768  0x7ffdf034c768
r8                       0x7ffdf034c6a0
140728633444000
r9                       0x7ffdf034c4e0
140728633443552
r10             0x8      8
r11             0x246    582
r12             0x4007b0 4196272
r13                      0x7ffdf034cca0
140728633445536
r14             0x0      0
r15             0x0      0
rip                      0x7fa1e8a2a2c0
0x7fa1e8a2a2c0<__accept_nocancel+7>
eflags          0x246    [ PF ZF IF ]
cs              0x33     51
ss              0x2b     43
ds              0x0      0
es              0x0      0
fs              0x0      0
gs              0x0      0
```

So now it is possible to use debug to gain an un-interpreted copy of the memory range starting with the address provided by r8. Note that it provided a string of 16 hex characters which would represent 64 bits which is to be expected because a 64 bit word length is used on this 64 bit processor based host. It is a little tricky interpreting this output because the words are reversed. So2062654620746153 would translate in ASCII to S (53), a (61), t (74) and so forth with the last byte of the word being a <space> (20). There is a better way to view this output by using the hexadecimal dump command (xxd) which will reorder and interpret that output. The result of using this command follows in the next section.

Debug is used to print out the first 64 16 bytes segments:

```
(gdb) x/64ca 0x7ffdf034c6a0
0x7ffdf034c6a0: 0x0      0x0
0x7ffdf034c6b0:          0x7fa1e8944d28
0x7fa1e894f8a0
0x7ffdf034c6c0:          0x7ffdf034c640
0x4173d8
0x7ffdf034c6d0:          0x7ffdf034c630
0x7fa1e89b3409<_IO_vsnprintf+121>
0x7ffdf034c6e0: 0x0      0x7fa1e8f0ba10
0x7ffdf034c6f0:          0x7fa1e8f0b4c0
0x400463
0x7ffdf034c700:          0x7fa1e8951d78
0x4003e8
0x7ffdf034c710:             0x100000000
0x100000725
0x7ffdf034c720:          0x7ffdf034cb90
0xffffffff00000000
0x7ffdf034c730: 0x0      0x7ffdf034c790
0x7ffdf034c740: 0x1      0x0
0x7ffdf034c750:          0x7ffdf034cbc0
0x4007b0<_start>
0x7ffdf034c760:          0x7ffdf034cca0
0x40096c<main+198>
0x7ffdf034c770:          0x7ffdf034cca8
0x17c96f087
0x7ffdf034c780: 0x1      0x589f8562
0x7ffdf034c790:      0x2062654620746153
0x32343a3531203131
0x7ffdf034c7a0:      0x373130322038353a
0x3030303030000a0d
0x7ffdf034c7b0:      0x3030303030303030
0x3030303030303030
0x7ffdf034c7c0:      0x3030303030303030
0x3030303030303030
```

To view the contents of the r8 memory range the dump memory command is used in debug. In the example below it is being saved to the home directory (~) of the users under the file name regmemsock. Note we are not starting at the beginning of the register, but rather at the point where the data appears to begin: 0x7ffdf034c780. By subtracting this address from the ending address in the gdb command we get 60 hex which means weare only dumping the first 1536 (256x6) bytes which is more than enough to store the date message. The result is hex dumps with the ASCII values interpreted which appear on the right side. Note that it pulls numbers in ASCII form so 34 hex is equal to a 4 in ASCII. One can read the message in clear and it appears starting on the 2nd row and end with

the soft characters 0a, 0d which in effect indicate end of message. Note that the value observed exactly matches the value returned to the client in the previous section even though many minutes elapsed before the values was extracted from memory. If another client had made an inquiry afterwards then that value would have been extracted.

Then dump the contents of the buffer starting at 0x7ffdf034c780

```
(gdb) dump memory ~/regmemsock 0x7ffdf034c780 0x7ffdf034c7e0
CRL\den.gus@os:~$ xxd ~/regmemsock | more
0000000: 0100 0000 0000 0000 6285 9f58 0000 0000  ........b..X....
0000010: 5361 7420 4665 6220 3131 2031 353a 3432  Sat Feb 11 15:42
0000020: 3a35 3820 3230 3137 0d0a 0030 3030 3030  :58 2017...00000
0000030: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
0000040: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
0000050: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
```

Conclusions

Many information technology professionals have unrealistic expectations in regard to how attacks can take place and how they can be prevented. There is often, unfortunately, a mindset that downloading new software patches can provide universal protection for an application (or at least until the next patch is released). Also, especially in a cloud environment symbolic links and other types of canonical names often shield users from the actual underlying data, which can be problematic. Often times when remediating complex security problems there is a need to evaluate this primary data. That data of course, may well be contained within memory or the registers. The trick then becomes how to best navigate the pointers to get to that data. The examples used herein have shown that the LINUX operating system often provides the memory mapping tools needed to do this. Because the potential limitation of viewing the mapping is related to user access rights, the potential compromise of a user that owns an application would then allow access to the memory mapping. Granted that the dynamic nature of the LINUX OS complicates the potential guessing of where sensitive data may reside in memory, however if the rights are obtained then the mapping provided by LINUX makes it fairly easy to access a user owned memory segment.

There are two primary examples depicted within this paper, in the first case it was shown that a given program could be run and the address of the sensitive data could be obtained via a register. Furthermore, that example also showed that a buffer overflow attack could result in the potential crashing that program. This disruption could occur either immediately or in a delayed fashion depending on various factors such as the buffer size. In the second example the register was used to find the memory address of potentially sensitive data on a socket call generated by a service. Unfortunately, these examples just represent the tip of the proverbial iceberg. The literature review presented several more dangerous scenarios and one can imagine many more potential dangers that could come to light with further research. However, the examples presented provide a foundation for understanding how the potentially 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 also be used to gain access to important information, and could also be used to launch memory/register attacks. This high level of functionality within the OS, even though it can be misused, provides an excellent diagnostic tool that can be used to gain a better understanding of how memory/register attacks take place. The scenarios offered reinforce concepts such as memory mapping, relative memory addressing, registers and process management. Often, information technology professionals grasp these concepts if framed in the context of a larger problem and of course in this paper the larger context was an attack against memory/registers.

In terms of the protection against these types of attacks it is often critical not to provide access to the cloud, zone and host in which the memory/register target is housed. This can begin with sound firewall techniques and the use of security layering (or segmenting) within the cloud. Programming applications while using sound software security techniques are also critical to the overall security posture. For example, object orient programming languages provide a wealth of classes that catch various exceptions that can then prevent potential buffer overflow

attacks. However, they are often times not implemented or if they are it is not done properly. In preparing examples for this paper it was clear that there are advantages to be gained from using the programming language java. Its memory map was far more complex than that of a similar program written in the C language. Its virtual machine context makes memory assignment far more dynamic (hence the need for a hacker to find the address in registers) and it does not allow direct memory addressing. The java code also took advantage of SMP, which further complicated the memory mapping and resulted in a further layer of encapsulation. For example there were 14 stacks associated with the socket call program because it was divided into 14 lightweight processes or threads, making the memory mapping far more difficult. 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 to provide information technology professionals or college students a quick review of the problem and then why it is so important within a cloud-computing environment. This topic will require further investigation to gain an understanding of the true scope of the potential problem described herein.

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*, (pp.101-116).

Aleph One. (2016). *Smashing the Stack for Fun and Profit*. Retrieved from 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*, (pp.4-4), August 09-13, 2004, San Diego, CA.

Buchanan, E., Roemer, R., Shacham, H., & Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to RISC. Proceedings of the 15th ACM Conference on Computer and Communications Security (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,* (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. *Proceedings of the 2nd Asia-Pacific Workshop on Systems,* July, 11-12, Shanghai, China.

Criswell, J., Geoffray, N., & Vikram, A. (2009). Memory Safety for low-level Software/Hardware Interactions. *Proceedings of the 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.

Eazynotes. (2016). Retrieved from 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, pp. 19-26.

Genkin, D., Pachmanov, L., Pipman, I., Shamir, A., & Tromer, E. (2016). Physical Key Extraction Attacks on PCs. *Communications of the ACM,* 59(6), 70-79.

Guimaraes, J. (1995). The object oriented model and its advantages. *ACM SIGPLAN OOPS Messenger*, 6(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*, (pp. 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*, 29(1), 5-20.

Li, J., Wang, Z., Jiang, X., Grace, M., & Bahram, S. (2010). Defeating return- Oriented rootkits with "return-less" kernels. *In Proceedings of EuroSys*, G. Muller, Ed. ACM Press, (pp. 195-208).

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

Petsios, T., Kemerlis, V. P., Polychronakis, M., & Keromytis, A. D. (2015). Dyna guard: 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). Retrieved from 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, (pp.15-17), Cambridge, MA, USA. [doi>10.1007/978-3-540-87403-4_1].

Stack Exchange. (2017). *Buffer overflow and register contents*. Retrieved from http://security.stackexchange.com/questions/89139/buffer-overflow-and-register-contents.

Stack Overflow. (2016). Retrieved from 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, W. (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*, (pp. 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.