

Lab 4

Memory Management

ITSC205: Operating Systems Internals

NAME: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[Lab Outcome(s) 3](#_Toc472777602)

[Reading 3](#_Toc472777603)

[Introduction 3](#_Toc472777604)

[1.0 Linux Memory Information and Settings 4](#_Toc472777605)

[2.0 Memory Segmentation Fault](#_Toc472777606) 6

[3.0 Stack Buffer Overflow](#_Toc472777606) 7

[4.0 Dynamic Memory Allocation- Heap ....](#_Toc472777607) 9

5.0 Conversion Virtual to Physical memory Address……………………………………………10

L*abs must be submitted by the due date for full credit. After due date late submissions will be accepted for a period of one week (seven days) and the grade will be reduced by ten percent (10%) per day after due day.* ***Assignments that are submitted more than seven days late will receive a grade of zero (0).***

I certify that the work submitted in this assignment is my own and that it has not been taken in whole or in part from any other source. I understand that the penalty for plagiarism will include a grade of zero (0) for this assignment plus disciplinary action in accordance with SAIT policies.

|  |  |  |
| --- | --- | --- |
| Linux memory information and settings | 15 |  |
| Memory Segmentation Fault | 10 |  |
| Stack Buffer Overflow | 10 |  |
| Dynamic memory allocation malloc( ) – Heap analysis | 10 |  |
| Virtual and Physical memory addresses and Page table analysis | 10 |  |
| TOTAL MARK | 55 |  |

Lab Outcome(s)

* Examine Linux memory settings.
* Create segmentation fault and stack buffer overflow
* Analyze dynamic memory allocation(heap) results
* Run a program to differentiate between virtual(linear) and physical memory addresses

Reading

* Textbook sections 9.3 (Paging), 9.4.1 (Hierarchical paging), 9.6( Structure of Page Table for Intel 32 and 64 bit Architectures), 20.6.2 (Virtual Memory)

Introduction

Memory is where all the magic happens, all the code and data to be executed resides in physical memory. Evidence of compromise may never be written to disk but memory has a high potential to contain malicious code in whole or in part. A victimized system will contain evidence that system resources were used by an attack.

1. Linux Memory Information and Settings \_\_\_\_/15

There are different tools in Linux that can be used to verify memory settings and usage. Some of the files that contain memory settings are /proc/meminfo, /proc/vmallocinfo, /proc/vmstat . To understand and find a detailed description of the output of memory files, use Linux manual, read handouts provided in class or access the following web site <https://www.kernel.org/doc/Documentation/filesystems/proc.txt>

Alternatively, you can use the commands ***free, top, vmstat*** to verify memory settings.

1. ( 4 marks) Use the Linux manual or access recommended link to find the description of the /proc/meminfo file. Explore the file and explain the purpose of the following:
   1. Dirty
   2. Buffers
   3. Mapped
   4. PageTables
2. (2 marks) Open a terminal and use **free** to view memory utilization. Now start **top** in another terminal and verify the memory usage of top program. Read man **top** and provide the difference between the RES and %MEM parameters?
3. (2 marks) The command ***getconf***  can be used to query system variables. To find out the page size of the system use getconf **PAGESIZE** and write down the page size in bytes?

Page size is defined in page header **page.h**. Access page.h under: /usr/src/linux-headers-3.13.0-164-generic/include/asm-generic (the kernel version of you’re system could be different). If you cannot find it use find command as follows: **find / -name page.h .** Verify the value of **PAGE\_SHIFT.** How many bits are defined for the page size?

1. (1 mark) How many page tables are there in a system and where are page tables located?
2. (1 mark). Use ***ps command*** to display all processes. Identify and explain the purpose of the thread kswapd.
3. ( 3 marks) Use proc directory to analyze memory settings of an existing process as follows:
   1. Create any process (e.g. man, gedit, web browser or run one of the C programs with endless loop).
   2. Access **/proc** for the process created and display the content of maps file for the process. Identify in the maps file
      1. Memory address range for text section
      2. Memory address range for heap section
      3. Memory address range for one of the data section
      4. Memory address range for stack
   3. Create a screen capture that demo the results and attach it to the lab.
   4. Open ***smaps*** file for the process and analyze the results.
      1. What is the purpose of this file?
      2. What is the MMUPagesize?
4. ( 2 marks ) Use **man vmstat** to learn how to use vmstat command and create a screen capture that demo how to display the report of memory statistics every 2 sec (delay) in Megabytes units. Attach the screen capture to the lab
5. Memory –Segmentation Fault \_\_\_\_/10

Segmentation fault is an illegal access to memory regions. A process is only allowed to access memory that belongs to it. Any access outside that area will cause a **segmentation fault. The OS kernel will send the SIGSEGV to the process.** Deference a null pointer, writing to a memory portion that was marked as read only can cause segmentation faults. It is very common in poorly written C or C++ programs

1. Modify pointergdb.c in a way that creates segmentation fault. Modify the pointer in a way that generates segmentation fault
2. Run the program to verify if it generates a segmentation fault
3. Use gdb to debug the program with segmentation error.
4. **gdb run**
5. **gdb print myptr** - Verify if this is the correct address
6. **gdb x address (replace address with myptr address)**
7. **gdb info address myptr**
8. **gdb print myvar**
9. **gdb bt** (backtrace to verify the stack) .What function is in the stack
10. Attach to the lab modified code that generates segmentation fault
11. Create a screen capture that demo the use of debugger gdb with respective commands to analyze the results of segmentation fault

3.0 Stack buffer overflow \_\_\_\_/10

**Stack buffer overflow** occurs when a program writes to a memory address on the program’s stack outside the fixed-length area. Stack buffer overflow bugs are caused when a program writes more data to a buffer located on the stack than what is actually allocated for that buffer. Overfilling the buffer on the stack is easier than the heap because the stack contains return addresses for all active function calls. Stack buffer overflow can be caused deliberately as part of an attack. It is one of the most common memory vulnerabilities

1. Download and analyze the program ***factorial.c***

In ***factorial.c*** program ***n*** was validated for **0**, but what about ***n < 0*** ?

**NOTE: Do not correct the code just run and debug it for negative values**

* 1. Compile the program (factorial.c)
  2. Used **gdb** to run and debug the program
  3. Provide positive values. Press ^C to end the loop
  4. Use ***gdb bt*** to verify stack results
  5. Use ***gdb run*** to run the program again and provide negative values
  6. Use ***gdb bt*** to verify and analyze stack results
  7. Attach respective screen captures that demo stack overflow.

1. There are some extensions for gdb used in exploit development. These extensions are: Pwngdb, PEDA or GEF. All of them has the pros and cons. We will use PEDA for the following stack overflow exercise. If you prefer GEF you can use it too. To install debugger gdb-peda do the following:
   1. **git clone https://github.com/longld/peda.git ~/peda**
   2. **echo "source ~/peda/peda.py" >> ~/.gdbinit**

**NOTE:** gdb-peda modifies gdb configuration file located at **~/.gdbinit**. Use cat **~/.gdbinit** to verify the settings. To remove peda and go back to gdb you can remove .gdbinit as follows: rm -rf ~/.gdbinit

1. Download and analyze **stackover.c** program
   1. Compile it
   2. Use the gdb-peda debugger to debug the code
   3. Use **break** command to create **break points** at main and at copybuffer function.
   4. Use **list** command to display the code
   5. To fill the buffer input 14 strings as follows: **run aaaaaaaaaaaaaa** if you have python you can also run it as follows: **run $(python3 –c “print( ‘a \* 14’)”)**
   6. Use **next** or **step** to move to the next line
   7. After each **next** or **step** command analyze registers (rsp,rbp and rip) and stack sections results.
   8. Use **bt** (backtrace) command to analyze more details on the stack
   9. Is there buffer overflow?
   10. How many strings do you need to input (you can input any character) to create stack buffer overflow?
   11. Analyze results of registers and stack sections before and after buffer overflow. What is the difference?
   12. Attach to the lab respective screen captures that demo the use of gdb-peda to analyze the results before buffer overflow and after stack buffer overflow

4.0 Dynamic Memory Allocation-Heap \_\_\_\_/10

1. Read ***malloc*** and ***free*** from Linux manual ***man malloc*** to learn basics on how to allocate and free dynamic memory.
2. The following code ***memory.c*** allocates dynamically blocks of memory (known as **heap**) and fills up the **stack**. Download from D2L the program called ***memory.c*** and analyze the code. Identify the lines that contain the malloc( ) function. How big are the memory blocks assign by this function?
3. Compile and execute the program with the following options and describe the results for:

./a.out - -malloc

./a.out --stack

1. What is the purpose of ***free( )*** function?
2. Modify memory.c program to free memory. Add the following line in the respective place to prevent memory leak.

***free(pAllocated) ;***

1. Compile the code and run it again with the --malloc option. Analyze the results. What is the effect of free(pointer) function ?
2. Attach respective screen captures to demo results.

**5.0 Conversion Virtual to Physical Memory Addresses \_\_\_\_/10**

Operating systems and hardware architectures use a technique called MMU (Memory Management Unit) that converts the virtual address into the physical address. The purpose of this activity is to identify virtual address and the respective physical address + offset of a process.

The mapping of virtual memory to physical memory is also accessible in user space in /proc/PID/pagemap. To understand the format of the pagemap file read the following web site:

<https://www.kernel.org/doc/Documentation/vm/pagemap.txt>

The purpose of the following programs (showadr.c and pagetables.c) is to find the **virtual address** of the main( ) function and the stack for the showadr program (process) and use the pagetable program to find the **respective physical address and offset** of these virtual addresses

1. Download the following C code from D2L:
   1. [showadr.c](https://cmps.biomea.com/cmps254/code/pagetables.c) and
   2. [pagetables.c](https://cmps.biomea.com/cmps254/code/pagetables.c)
2. Compile showadr.c as follows: ***gcc –w –o show showadr.c***
3. Run the program **./show** and record the PID of this process: \_\_\_\_\_\_\_
4. Record the base **virtual addresses** of:
   1. main( ) function:
   2. stack:

NOTE: The last three digits of the virtual address is the offset. Identify and underline the

offset of each address

1. Do NOT exit show program. Keep it running

NOTE: In order to access physical address and run page tables you need high privileges. You have to sudo to run pagateables.c

1. Open a second terminal and compile pagetables.c program as follows:

***gcc -w -g `getconf LFS\_CFLAGS` -o pagetables pagetables.c***

Now run pagetable program to find the respective physical memory address of the current running ***show*** program.

1. Run pagetables program as follows:

***./pagetables PID*** (replace PID with the PID of the running show program. PID recorded above)

1. Analyze the results. Notice that the results of this program are extracted from the /proc/pid/maps file

Virt - Represents Virtual page address

Present – Represents the Base Physical page address

Phys – Represents the Base Physical address +offset (last three values)

1. Find and write down the respective physical address of the virtual address of main( ) function (address recorded above).
2. Find and write down the respective physical address of the virtual address of the stack (address recorded above).
3. Attach respective screen captures that demo only the address conversion of the main( ) function and the stack