

Lab 6

Linux 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 and Satck Buffer Overflow](#_Toc472777606) 7

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

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

|  |  |  |
| --- | --- | --- |
| Linux memory information and settings | 15 |  |
| Memory Segmentation Fault and Stack Buffer overflow | 10 |  |
| Stack Buffer Overflow | 5 |  |
| Dynamic memory allocation malloc( ) – Heap analysis | 10 |  |
| Virtual and Physical memory addresses and Page table analysis | 10 |  |
| TOTAL MARK | 50 |  |

Lab Outcome(s)

* Examine Windows memory settings.
* Examine Windows memory management techniques.
* Analyze memory behavior of normal system operations.

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: Memory which is waiting to get written back to the disk



* 1. Buffers: Relatively temporary storage for raw disk blocks

shouldn't get tremendously large (20MB or so)



* 1. Mapped:files which have been mmaped, such as libraries



* 1. PageTables:amount of memory dedicated to the lowest level of page

tables.



1. (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?

* MEM:A task's currently used share of available physical memory.
* RES: The non-swapped physical memory a task is using.

1. (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?



1. (1 mark). Us ps command to display all processes. Identify and explain the purpose of the thread kswapd.

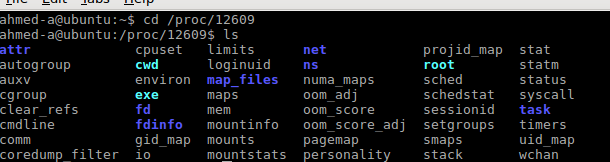
Kswapd: The Kernel Swap Daemon

-is a bit of a misnomer as the daemon does more than just swap modified pages out to the swap file. Its task is to keep the memory management system operating efficiently. The Kernel swap daemon (kswapd kernel init process at startup time and sits waiting for the kernel swap timer to periodically expire. ) is started by the Every time the timer expires, the swap daemon looks to see if the number of free pages in the system is getting too low.

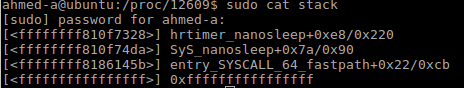
1. ( 3 marks) Use proc directory to analyze memory settings of an existing process as follows:
   1. Create any process (e.g. man, gedit, web browser or run one of the C programs with endless loop).
   2. Open a second terminal and find the PID of the process created.



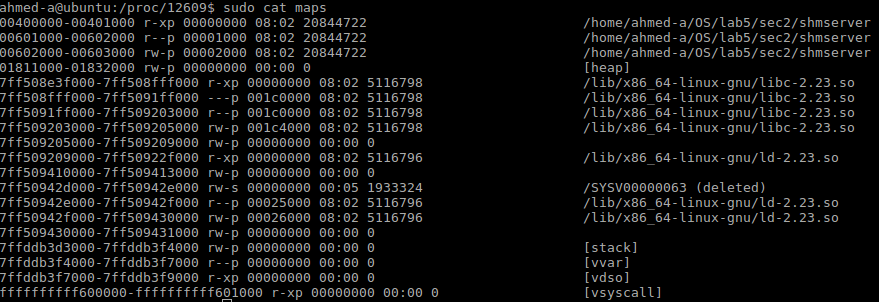
* 1. Access the directory **/proc/PID (replace PID with the respective PID of the process you created)** and type ***ls*** command to find and analyze memory files and settings for this process.



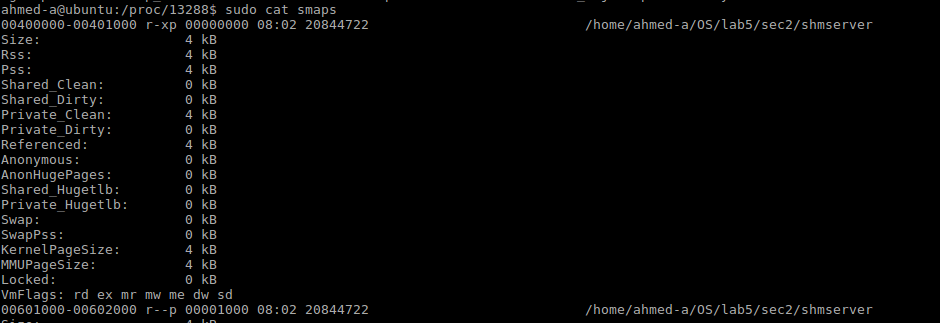
* 1. **Demo and explain** to the instructor the content of the following files:
     1. The ***stack*** file: addresses and the respective data within the stack



* + 1. ***maps*** file: Memory regions or areas and its respective address, permissions, offset

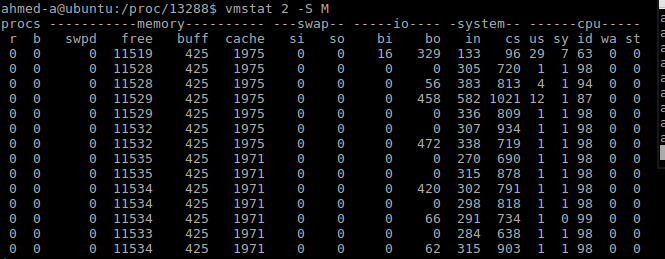


* + 1. ***smaps*** file: details about heap and stack settings. What is the MMUPagesize?



1. ( 2 marks ) Use **man vmstat** to learn how to use vmstat command and demo to the instructor how to display the report of memory statistics every 2 sec (delay) in Megabytes units

vmstat: Report virtual memory statistics - reports information about processes, memory, paging, block IO, traps, disks and cpu activity.



|  |  |
| --- | --- |
| **ASK THE INSTRUCTOR TO SIGN OFF** |  |

1. 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

****

1. Use gdb to debug the program with segmentation error.
2. **gdb run**
3. **gdb print myptr** - Verify if this is the correct address



1. **gdb x address (replace address with myptr address)**
2. **gdb info address myptr**



1. **gdb print myvar**



1. **gdb bt** (backtrace to verify the stack) .What function is in the stack

**10.** Demo results to the instructor

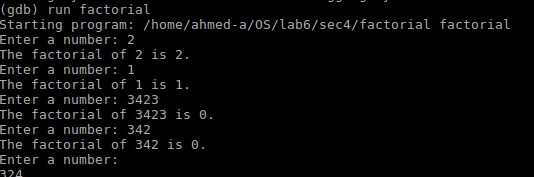
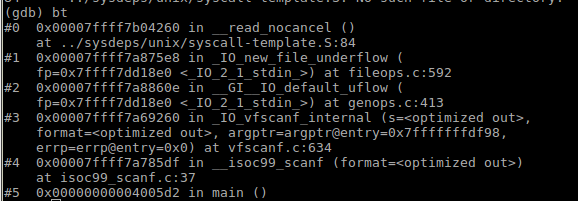
|  |  |
| --- | --- |
| **ASK THE INSTRUCTOR TO SIGN OFF** |  |

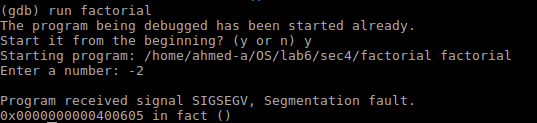
Stack buffer overflow \_\_\_\_/5

**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 security vulnerabilities.

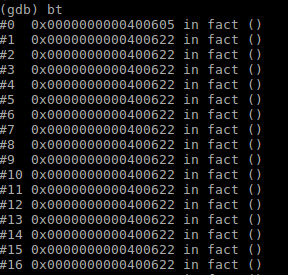
In the previous ***factorial.c*** program ***n*** was validated for **0**, but what about ***n < 0*** ?

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



1. Use ***gdb bt*** to verify and analyze stack results



1. Demo and explain the results to the instructor

|  |  |
| --- | --- |
| **ASK THE INSTRUCTOR TO SIGN OFF** |  |

1. Dynamic Memory Allocation-Heap \_\_\_\_/10
2. Read ***malloc*** and ***free*** from Linux manual ***man malloc*** to learn basics on how to allocate and free dynamic memory.

Malloc: The malloc() function allocates size bytes and returns a pointer to the

allocated memory.

ptr = (cast-type\*) malloc(byte-size)

EXAMPLE: ptr = (int\*) malloc(100 \* sizeof(int));

free: displays the total amount of free and used physical and swap memory in the system, as well as the buffers and caches used by the kernel.

Example: free(ptr);

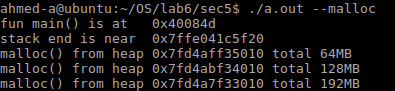
1. The following code ***memory.c*** allocates dynamically blocks of memory (known as **heap**) and fills up the **stack**. Download from D2L the program called ***memory.c*** and analyze the code. Identify the lines that contain the malloc( ) function. How big are the memory blocks assign by this function?

64mb

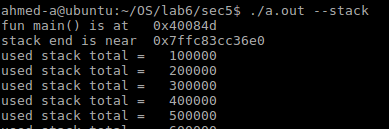
512 MB

1. Compile and execute the program with the following options and describe the results for:

./a.out --malloc



./a.out --stack

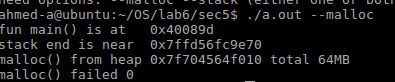


1. What is the purpose of ***free( )*** function?

You can call the malloc function at any time, and it will request a block of memory from the heap. The operating system will reserve a block of memory for your program, and you can use it in any way you like. When you are done with the block, you return it to the operating system for recycling by calling the free function

1. In the memory.c code in line 68 add the following:

***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. Explain and demo the results to the instructor.

|  |
| --- |
| **ASK THE INSTRUCTOR TO SIGN OFF** |

**4.0 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: \_\_\_\_\_\_\_



1. 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

864

1. Do NOT exit show program. Keep it running
2. Open a second terminal and compile pagetables.c program as follows:

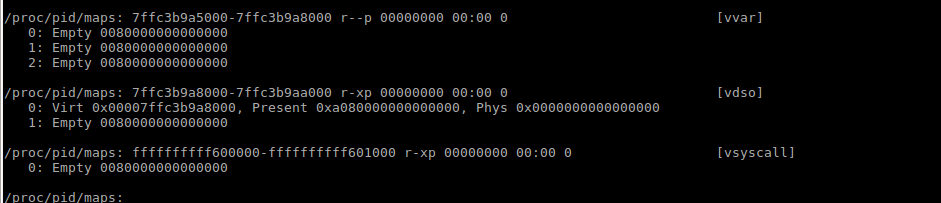
***gcc -w -g `getconf LFS\_CFLAGS` -o pagetables pagetables.c***

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 the respective physical address of the virtual address of main( ) function (address recorded above). Write down the physical memory location where main( ) function is located:
2. Find the respective physical address of the virtual address of the stack (address recorded above). Write down the physical memory location where the stack is located:
3. Demo and explain the results to the instructor