Exercise 09

Apr 30, 2023

Overview

This exercise provides hands-on experience on how to launch Stack Smashing attack using Buffer overflow mechanism.

Learning Objectives

- Understand basics of Buffer Overflow mechanism
- Understand Use of Stacks in storing local variables, function parameters and return address.
- Understand basics of stack corruption i.e. stack smashing

Reading Material

1. C and Assembly programming
    - https://dev.to/yjdoc2/series/8954
2. Chapt 10: Buffer Overflow, Textbook- "Computer Security: Principles and Practice", Author: Stallings and Lawrie
    - https://linux.die.net/man/8/tcpdump
3. Using `gdb`
    - https://linux.die.net/man/1/gdb
    - https://developers.redhat.com/blog/2021/04/30/the-gdb-developers-gnu-debugger-tutorial-part-1-getting-started-with-the-debugger#compiler_options
    - https://developers.redhat.com/articles/2022/11/08/introduction-debug-events-learn-how-use-breakpoints#next_up
    - Many resources on internet on using GDB
4. Using od (octal dump)
    - https://linux.die.net/man/1/od
5. Book "Computer Systems: A Programmer's Perspective" 3rd ed, Authors: Randal E. Bryant, David R. O'Hallaron.
    - The book provides a detailed insight of high any C program is implemented in assembly and corresponding stack memory management.

Prerequisites and environment familiarity

- Familiarity with Basic C programming.
- Familiarity with using Gnu Debugger (`gdb`) and how to examine and set values.
- Familiarity with buffer overflow in programs.

Description

This project assignment is to be done **individually**. The assignment consists of writing a C program which consists or *main*() and 2 or more functions. A sample program `stack2.c` is given for the same. In this sample program, *main*() invokes function *foo*[1]() which in turns invokes *bar*(). Before returning from *bar*(), it prints its name i.e. "`bar`" and returns a value 2. Similarly, before returning from *foo*, it prints its name "`foo`" and returns the value 1. The main program checks the returned value and prints "`Returned properly`", when returned value is 1 other it prints "`Improper return`".

A normal invocation of this program stack2 outputs the following

```
rprustagi@crbase-ubuntu22:~/Prog/BufOverflow$ ./stack2
bar
foo
Returned properly
```

This is shown in screenshot as below in Figure 1 when it is run in `gdb`.

```
rprustagi@crbase-ubuntu22:~/Prog/BufOverflow$ gdb stack2
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack2...
(gdb) run
Starting program: /home/rprustagi/Prog/BufOverflow/stack2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
bar
foo
Returned properly
[Inferior 1 (process 3084988) exited normally]
(gdb)
```

Figure 1: Normal execution of program

[1] The function foo() and bar() are used as place holders, and you should use your own function names.

This assignment consists of 3 parts.
Part 1: It involves simple invocation of program and showing the proper execution and corresponding output.
Part 2: It involves running the program under GDB, setting a break point at *foo*() and *bar*() noting the values of registers `%rbp`, `%rsp` and return addresses respectively when the invoked function completes execution. The main task in this part of the assignment is to set the values on stack. The values to be set should correspond to the situation as if execution flow is returning from the first invoked function *foo*(). Thus, when execution resumes, the return from *bar*() directly goes to *main*) and it will output the following

```
Bar
Improper return
```

The use of `gdb` to examine and set the value on stack (i.e. corrupting the stack), and corresponding memory layout are shown in three screenshots as below.

- Figure 2 shows the memory layout of stack during program execution.
- Figure 3 shows how to examine the values on stack during execution.
- Figure 4 shows how to manually modify the values on stack during execution.

Stack Space		Values		
0x00007FFF	FFFFE320			
0x00007FFF	FFFFE31C	0x00000000	int n (main)	
0x00007FFF	FFFFE318		Unused	
0x00007FFF	FFFFE314		FILE *fd (main)	
0x00007FFF	FFFFE310			
0x00007FFF	FFFFE30C		char x[128] (main)	
:	:			
0x00007FFF	FFFFE290			
0x00007FFF	FFFFE28C	5555	return addr after calling foo(x)	
0x00007FFF	FFFFE288	555552A8		
0x00007FFF	FFFFE284	7FFF	prev value of register %rbp	
0x00007FFF	FFFFE280	FFFFE320		
0x00007FFF	FFFFE27C	0x00000001	int n(foo)	
0x00007FFF	FFFFE278		Unused for 16 byte aligned address	
0x00007FFF	FFFFE274			
0x00007FFF	FFFFE270			
0x00007FFF	FFFFE26C		char fx[64] (foo)	
:	:			
0x00007FFF	FFFFE230			
0x00007FFF	FFFFE22C	7FFF	Addr of char *str (parameter to fun foo)	
0x00007FFF	FFFFE228	FFFFE290		
0x00007FFF	FFFFE224		Unused	
0x00007FFF	FFFFE220			
0x00007FFF	FFFFE21C	5555	return addr after calling bar(x)	For stack corruption, change it to 0x5555555552A8, ret addr after foo()
0x00007FFF	FFFFE218	555551CA		
0x00007FFF	FFFFE214	7FFF	prev value of register %rbp	For stack corruption, change it to 0x7FFFFFFFE320, prev value of %rbp
0x00007FFF	FFFFE210	FFFFE280		
0x00007FFF	FFFFE20C	2	int n (bar)	
0x00007FFF	FFFFE208		Unused for 16 byte aligned address	
0x00007FFF	FFFFE204			
0x00007FFF	FFFFE200			
0x00007FFF	FFFFE1FC		char bx[64] (bar)	
:	:			
0x00007FFF	FFFFE1C0			
0x00007FFF	FFFFE1BC	7FFF	Addr of char *str (parameter to fun foo)	
0x00007FFF	FFFFE1B8	FFFFE290		
0x00007FFF	FFFFE1B4			
0x00007FFF	FFFFE1B0			

Figure 2: memory layout of program

~/OneDrive - Advanced Computing and Communications Society/UMBC/CMSC-626/Prog — rprustagi@crbase-ubuntu22: ~/Prog/B

...crbase-ubuntu22: ~/Prog/BufOverflow — -bash | ...crbase-ubuntu22: ~/Prog/BufOverflow — -bash | ...fOverflow — ssh rprustagi@crbase-ubur

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack2...
(gdb) b foo
Breakpoint 1 at 0x11b7: file stack2.c, line 17.
(gdb) b bar
Breakpoint 2 at 0x1179: file stack2.c, line 8.
(gdb) run
Starting program: /home/rprustagi/Prog/BufOverflow/stack2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, foo (str=0x7fffffffe290 "ABCDEFGHIJKLMNOPQRSTUVWXYZ") at stack2.c:17
17          int n = 1;
(gdb) p $rbp
$1 = (void *) 0x7fffffffe280
(gdb) p $rsp
$2 = (void *) 0x7fffffffe220
(gdb) x/2 0x7fffffffe280
0x7fffffffe280: -7392   32767
(gdb) x/2wx 0x7fffffffe280
0x7fffffffe280: 0xffffe320      0x00007fff
(gdb) x/2wx 0x7fffffffe288
0x7fffffffe288: 0x555552a8      0x00005555
(gdb) c
Continuing.

Breakpoint 2, bar (str=0x7fffffffe290 "ABCDEFGHIJKLMNOPQRSTUVWXYZ") at stack2.c:8
8           int n = 2;
(gdb) p $rbp
$3 = (void *) 0x7fffffffe210
(gdb) p $rsp
$4 = (void *) 0x7fffffffe1b0
(gdb) x/2 0x7fffffffe210
0x7fffffffe210: 0xffffe280      0x00007fff
(gdb) x/2 0x7fffffffe218
0x7fffffffe218: 0x555551ca      0x00005555
(gdb)
```

Figure 3: Examining value of registers and respective locations

```
~/OneDrive - Advanced Computing and Communications Society/UMBC/CMSC-626/Prog — rprustagi@crbase-ubuntu22: ~/Prog/
...crbase-ubuntu22: ~/Prog/BufOverflow — -bash    ...crbase-ubuntu22: ~/Prog/BufOverflow — -bash    ...fOverflow — ssh rprustagi@crbase-ubu
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, foo (str=0x7fffffffe290 "ABCDEFGHIJKLMNOPQRSTUVWXYZ") at stack2.c:17
17          int n = 1;
(gdb) p $rbp
$1 = (void *) 0x7fffffffe280
(gdb) p $rsp
$2 = (void *) 0x7fffffffe220
(gdb) x/2 0x7fffffffe280
0x7fffffffe280: -7392   32767
(gdb) x/2wx 0x7fffffffe280
0x7fffffffe280: 0xffffe320      0x00007fff
(gdb) x/2wx 0x7fffffffe288
0x7fffffffe288: 0x555552a8      0x00005555
(gdb) c
Continuing.

Breakpoint 2, bar (str=0x7fffffffe290 "ABCDEFGHIJKLMNOPQRSTUVWXYZ") at stack2.c:8
8           int n = 2;
(gdb) p $rbp
$3 = (void *) 0x7fffffffe210
(gdb) p $rsp
$4 = (void *) 0x7fffffffe1b0
(gdb) x/2 0x7fffffffe210
0x7fffffffe210: 0xffffe280      0x00007fff
(gdb) x/2 0x7fffffffe218
0x7fffffffe218: 0x555551ca      0x00005555
(gdb) set *0x7fffffffe210=0x5555555552a8
(gdb) set *0x7fffffffe210=0x7fffffffe320
(gdb) set *0x7fffffffe218=0x5555555552a8
(gdb) c
Continuing.
bar
Improper return
[Inferior 1 (process 3084970) exited normally]
(gdb)
```

Figure 4: Setting the value in stack memory (stack corruption)

Part 3: It involves constructing the input data file `badfile` such that when its contents are read into local buffer variable, it corrupts the stack in such a way that when it returns from *bar*(), it directly returns to *main()* bypassing the return from *foo*() and produces outputs as is seen in part 2.

Assignment details

Part 1:

Write a program similar to `stack2.c`. Replace the function name *foo*() and *bar*() with your first and last name. e.g., if the name is "`ram rustagi`", then *foo*() should be renamed as *ram*() and *bar*() should be renamed as *rustagi*(). Accordingly change the content of `printf` statements that prints the function name.

The size of local variables buffer `bx[]` and `fx[]` should be changed to equal to last 3 digits of UMBC ID aligned to boundary of 16 bytes. For example, if your UMBC ID IA38490, then last 3 digits are 490, then next boundary aligned to 16 bytes would be 496. Thus declare these arrays as

`char bx[496];` and
`char fx[496]`;

When this program is compiled, it should function correctly, and produce expected output.

Part 2:

Compile the program with options which enables stack area to contain code which can be executed e.g.

```
gcc -g -o stack2 -z execstack -fno-stack-protector stack2.c
```

To ensure that program always gets the same stack location on each execution, disable the kernel configuration parameter `kernel.randomize_va_space` by setting its value to 0, i.e., issue the command

```
sudo sysctl -w kernel.randomize_va_space=0
```

Run the program under `gdb` (Gnu Debugger), put a break point at *foo()* and bar(), and run the program. When it breaks at *foo*, look at the value of registers `%rbp` and `%rsp` and content of memory location pointed to by `%rbp`. This has been shown in Figure 3. For example, when break point at *foo()* is hit, value of `%rbp` is `0x7FFFFFFFe280`, and memory content of this location is `0x7FFFFFFFE320` corresponding to previous value of `%rbp` register.. Similarly the return address of the location where *foo*() will return is `0x5555555552A8` (corresponds to line number 35 in `stack2.c` of *main*() function) is stored on the stack at location `0x7FFFFFFFE288`. The stack address range `0x7FFFFFFFE290-0x7FFFFFFFE310` corresponds to local variable char x[128] of *main*() function. The stack address `0x7FFFFFFFE310` corresponds to local variable `FILE *fd` of *main*() and similarly, the address `0x7FFFFFFFE318` corresponds local variable `int n` of *main*().

You need to identify these values for your program. These values will change since size of your local variable character array `fx[]` has been changed from 64 to a value corresponding to your UMBC Id.

Continue the execution and program will stop at *bar*(). Examine the value of `%rbp`, `%rsp` and corresponding memory location. This is also shown in Figure 3. Value of `%rbp` is `0x7FFFFFFFE210` which contains the value `0x7FFFFFFFE280` which is previous value of

%rbp (as seen when execution hit the breat point at *foo*()) Similarly, the content of location 0x7FFFFFFFE218 is 0x5555555551CA which corresponds to return address (code segment) corresponding to line 20 (stack2.c) where it will return to program code in *foo*() after *bar*() completes execution and return.

To launch our stack smashing attack, we would like stack to be modified in such a way that when execution returns from *bar*(), it should directly return back to *main*() function (at line #35) where it would have as if returned from *foo()*. Thus, we essentially note the stack value when execution hits the break point *foo*() and copy these values to stack area when execution hits the break point at *bar*(). This is shown in Figure 4. This is called stack corruption or stack smashing.

Your task is to change the value of stack location for your program causing stack corruption. Essentially, identify the return address in *foo*() and replace these value to that of return address in *main*.

After changing the value on stack, also called stack smashing, continue running the program. On its return from *bar*(), execution will directly return to *main* and continue its execution. Thus, it should not print "foo" (equivalent function name e.g. "ram")but directly prints the output of *main*().

Part 3:

Objective of this part is to implement Stack smashing via buffer overflow from reading the malicious values from a file. Consider the modified implementation of stack2.c as given in stack3.c. The function *bar*() is modified to read data from a file "badfile" and writes the read values into local buffer bx[]. The local buffer size is 64 and file badfile size is 96 bytes and thus this reading of malicious data results in stack corruption. When the value of "badfile" are so chosen so as to corrupt the stack in the desired way, return from *bar*() will directly return to execution from *main()*.

To create our badfile, we load the program stack3 in gdb, put a break point at *bar*() and note down the values of register %rbp which is 0x7FFFFFFFE210 (same as in the case of stack2). The return address of next instruction in *foo*() (after *bar*() is called) is stored in stack at location 0x7FFFFFFFE18. The content of this memory is 0x7FFFFFFFE280 which is previous value of %rbp. The previous to previous value of %rbp (when *foo()* is called from *main*) is at this stack memory which is 0x7FFFFFFFE320 (same as before in stack2.c). Similarly, return address of instruction in *main*() after *foo*() is called is at location is 0x7FFFFFFFE288 which in this case happens to be 0x55555555530F. So to corrupt the stack we need to copy the value 0x7FFFFFFFE320 at stack location 0x7FFFFFFFE210 (can be accessed from local variable buffer range bx[80] to bx[87]), and value 0x55555555530F at location 0x7FFFFFFFE218 (can be accessed from bx[88] to bx[95]). Since *foo* returns 1, we need to change the return value of *bar()* from 2 to 1 and thus we need to set the value `0x00000001` at the stack location `0x7FFFFFFF20C` (can be accessed from `bx[76]` to `bx[79]`).

To help understand the overall task, a program to carefully set these values is given in `genbadfile`.c which writes the malicious data in `badfile`. Now when we run this program in stack3 in `gdb`, its output will not execute the statement `printf("foo\n")` as flow will directly return from *bar*() to *main* and it will display "`Returned properly`".

Your task is create your own version of `stack3.c`, understand the storage management in stack, call return addresses and function parameters. You should first run the program `stack3` in `gdb` to note the required memory location, modify the program `genbadfile.c` with corresponding applicable values, generate malicious content "`badfile`" and then run the program `stack3` in debugger to show the impact of malicious buffer overflow changing the program execution flow.

Explanation and Hints

a. Revise your C programming and memory management on stack.
b. Refresh/revise your `gdb` skills.

Assessment and Rubric

Please do submit the following

1. `Readme.txt` file which will contain the following information
    a. Your details i.e. UMBC Id and Name.
    b. Details on computation of buffer size `bx[]` as per UMBC Id.
    c. Explanation of how did you compute the return addresses in debugger (`gdb`).
    d. Explanation of values being used in `genbadfile.c` to generate malicious output "`badfile`".
    e. Commands used to compile the C program, and setting of required kernel parameters.
    f. Challenges faced and how did you resolve these challenges.
    g. Summary of your learning.
    h. References: Any website/resource that you used to took help.

2. `stack2.c` with comments explaining the changes made.
3. `stack3.c` with comments explaining the changes.
4. `genbadfile.c` with comments and generated "`badfile`".
5. Screen capture of `gdb` work window where you computed/identifies the values on stack to be used for stack smashing.

Rubric for assessment (40 marks)

a. 10 marks for a `Readme.txt` file containing all the required information
b. 10 marks for `stack2.c` explaining the changes as comments.
c. 10 marks for screen capture of `gdb`.
d. 5 marks for `stack3.c` explaining the changes as comments.
e. 5 marks for `genbadfile.c` and "`badfile`" explaining the specific values.

Bonus marks: 10 marks

a. Write a new program `stack4.c` that programmatically computes the stack address of return value of *foo* and implements stack smashing. This would require computing the stack address of local variables in *bar*() as well as stack address of return address of *foo* and then corrupting the stack values. This program should not be required to read any file as malicious input, but simply implements stack smashing within the program automatically. You can make the assumption that both *foo*() and *bar*() are using same type and number of local variables.

Note

- Any plagiarism activity will result in penalties of being awarded 0 marks. If you are using the sample program as shown in the class, please attribute the same.
- This project exercise should be carried out in the ubuntu-22 VM where you login using GlobalProtect VPN. You are neither required nor expected to use any other VM system.

<End of Exercise 9>