I. INTRODUCTION

The goal of this project is to help students become familiar with memory protection facilities provided by operating systems. In particular, you will learn how to limit access to a certain region of memory (e.g., only read, write or execute access). The project has two parts. First, you will learn about mprotect(), a memory protection system call. In the second part, you will explore why it is a good idea to disable execution on the stack to safeguard a program's execution.

II. Task 1: Protection of Memory via mprotect()

This task is divided into two parts, (A) Understanding mprotect(), and (B) Experimenting with programs that use this call.

Part A: Understanding Memory Protection

1. Why do you get SIGSEGV when you execute the object code generated by compiling the mprotect.c program?

   A two-page buffer in memory is declared, initial protection is R/W:

   ```
   p = memalign(pagesize, size);    /*Allocating buffer'p' of size = two
                                    pages, where a page == 4096 bytes*/
   ```

   Protection for the second page (`p+pagesize`) is set to allow read, `PROT_READ`:

   ```
   if (mprotect(p+pagesize, pagesize, PROT_READ)==-1) {
         handle_error("mprotect");
   }
   ```

   When write access to the second page is attempted, a SIGSEGV error is thrown and handled by the program.

   ```
   i=4096;             /* i = 4096, so that i+4096 will point to 2nd page */
   *(buffer+i) = 'G';  /* Trying to Overwrite 1st byte on 2nd page */
   ```

2. How does mprotect() protect memory ? What is the minimum size requirement for this call, what happens if you pass the value of the .len. argument as 1?

   From `man mprotect` f:

   - If the calling process tries to access memory in a manner that violates the protection, then the kernel generates a SIGSEGV signal for the process.
   - The minimum size requirement for this call is: `addr+len-1`. I ran 3 experiments for the len = [1, 0, -1]:

When len===1, the 2nd page has protection, e.g., a SIGSEGV error is thrown. My experiment attempted to verify if protection is applied to only the 1st byte when len==1. However, it appears that protection is applied to the entire page==4096 bytes:

```
if (mprotect(p+pagesize, 1, PROT_READ)==-1) {
      handle_error("mprotect");
}

i=4096;      /* i = 4096, so that i+4096 will point to 2nd page */
// *(buffer+i) = 'G';     /* Trying to Overwrite 1st byte of 2nd
                          page, this is commented out in order to
                          see how modifying the value for len
                          affects the protection */
*(buffer+(i+1)) = 'A';

/* Output:
Second Page:
4097=A, 9d56000
4098=A, 9d56001
4099=A, 9d56002
8190=A, 9d56ffd
8191=A, 9d56ffe
8192=A, 9d56fff

Got SIGSEGV at address: 0x9d56001
*/
```

When len===0, the second page has no protection, e.g., no SIGSEGV error is thrown:

```
if (mprotect(p+pagesize, 0, PROT_READ)==-1) {
      handle_error("mprotect");
}

i=4096;      /* i = 4096, so that i+4096 will point to 2nd page */
// *(buffer+i) = 'G';     /* Trying to Overwrite 1st byte of 2nd
                          page, this is commented out in order to
                          see how modifying the value for len
                          affects the protection */
*(buffer+(i+1)) = 'A';

/* Output:
Second Page after overwriting:
4097=A, 82ab000
4098=A, 82ab001
4099=T, 82ab002
8190=E, 82abffd
8191=C, 82abffe
8192=H, 82abfff
Loop completed
*/
```

Setting len==-1 does not allocate any memory, thus the program crashes upon attempting to write to unallocated memory:

```
if (mprotect(p+pagesize, -1, PROT_READ)==-1) {
      handle_error("mprotect");
}

i=4096;      /* i = 4096, so that i+4096 will point to 2nd page */
// *(buffer+i) = 'G';     /* Trying to Overwrite 1st byte of 2nd
                          page, this is commented out in order to
                          see how modifying the value for len
                          affects the protection */
*(buffer+(i+1)) = 'A';

/* Output:
mprotect: Cannot allocate memory
*/
```

3. Sam is a programmer and he needs to protect a 0x380 byte block starting at memory address 0x1234000 and ending at address 0x1234379. This block should be protected from overwriting by some other internal function. Can Sam use the mprotect() function in his C program to protect this memory space? Explain your answer.

   Data in memory can be protected from writing using the mprotect() function by setting the protection to:

   - PROT_NONE The memory cannot be accessed at all.

```
if (mprotect(p+pagesize, pagesize, PROT_NONE)==-1) {
      handle_error("mprotect");
}

i=4096;      /* i = 4096, so that i+4096 will point to 2nd page */
// *(buffer+i) = 'G';     /* Trying to Overwrite 1st byte of 2nd
                          page, this is commented out in order to
                          see how modifying the value for len
                          affects the protection */
*(buffer+(i+1)) = 'A';

/* Output:
Second Page:
4097=A, 9bcb000
4098=A, 9bcb001
4099=A, 9bcb002
8190=A, 9bcbffd
8191=A, 9bcbffe
8192=A, 9bcbfff

Got SIGSEGV at address: 0x9bcb001
*/
```

- PROT_READ The memory can be read. As shown in Question 2, PROT_READ can be set, thus enabling read only access to memory.

```
if (mprotect(p+pagesize, pagesize, PROT_READ)==-1) {
      handle_error("mprotect");
}

i=4096;      /* i = 4096, so that i+4096 will point to 2nd page */
// *(buffer+i) = 'G';     /* Trying to Overwrite 1st byte of 2nd
                          page, this is commented out in order to
                          see how modifying the value for len
                          affects the protection */
*(buffer+(i+1)) = 'A';

/* Output:
Second Page:
4097=A, 8bc4000
4098=A, 8bc4001
4099=A, 8bc4002
8190=A, 8bc4ffd
8191=A, 8bc4ffe
8192=A, 8bc4fff

Got SIGSEGV at address: 0x8bc4001
*/
```

4. Review and report what data is being protected in mprotect.c by the mprotect() function. Moreover, discuss if any function can perform read or write on the protected data. If the mprotect call is used in multiple instances, write your observations for each of them.

   Experiments and observation for memory/data protection offered by the mprotect() function were discussed in Task 1, Part A, Questions [1-3]. The discussion for Question 3 allowed read access while providing write protection on a block of memory. The following experiment attempts to provide write protection, while allowing read access. In this experiment, read access did not throw a SIGSEGV error.

```
if (mprotect(p+pagesize, pagesize, PROT_WRITE)==-1) {
      handle_error("mprotect");
}

i=4096;      /* i = 4096, so that i+4096 will point to 2nd page */
// *(buffer+i) = 'G';     /* Trying to Overwrite 1st byte of 2nd
                          page, this is commented out in order to
                          see how modifying the value for len
                          affects the protection */
*(buffer+(i+1)) = 'A';

/* Output:
Second Page after overwriting:
4097=A, 83d5000
4098=A, 83d5001
4099=T, 83d5002
8190=E, 83d5ffd
8191=C, 83d5ffe
8192=H, 83d5fff
Loop completed
*/
```

Part B: Experimentation and Implementation

1. Write first n bytes of last two pages (9th and 10th page) with your first name where n is equal to number of characters in your first name.

```
// Task 1, Part B: Question 1 output:
Address of 1 Page: 8dcc000
Address of 2 Page: 8dcd000
Address of 3 Page: 8dce000
Address of 4 Page: 8dcf000
Address of 5 Page: 8dd0000
Address of 6 Page: 8dd1000
Address of 7 Page: 8dd2000
Address of 8 Page: 8dd3000
Address of 9 Page: 8dd4000
Address of 10 Page: 8dd5000

32768=R, 8dd4000
32769=o, 8dd4001
32770=b, 8dd4002
32771=e, 8dd4003
32772=r, 8dd4004
32773=t, 8dd4005

36864=R, 8dd5000
36865=o, 8dd5001
36866=b, 8dd5002
36867=e, 8dd5003
36868=r, 8dd5004
36869=t, 8dd5005
```

2. Now, use mprotect() to allow read and write access on 7th and 8th page. Write your last name in first n bytes of 7th and 8th pages, where n is equal to number of characters in your last name and then try to read it. You should display it on output screen (print to STDOUT).

```
if (mprotect(p+(pagesize*6), pagesize*2, PROT_READ|PROT_WRITE)==-1) {   /* R/W access */
      handle_error("mprotect");
}

// Task 1, Part B: Question 2 output:
24576=B, 8dd2000
24577=o, 8dd2001
24578=b, 8dd2002
24579=k, 8dd2003
24580=o, 8dd2004
24581=s, 8dd2005
24582=k, 8dd2006
24583=i, 8dd2007
24584=e, 8dd2008

28672=B, 8dd3000
28673=o, 8dd3001
28674=b, 8dd3002
28675=k, 8dd3003
28676=o, 8dd3004
28677=s, 8dd3005
28678=k, 8dd3006
28679=i, 8dd3007
28680=e, 8dd3008
```

3. Now, use mprotect on 6th and 5th page and only give write access through mprotect() function. Now write your n character Georgia Tech user-id like (adeshpande74), where n equal to number of characters in your user-id. After writing try to read it. Can you read it?

   After writing, the memory can be read. This was unexpected, as my assumption was that setting protection to `PROT_WRITE` would make the memory write only:

```
if (mprotect(p+(pagesize*4), pagesize*2, PROT_WRITE) == -1) {     /* W access Allowed /*
      handle_error("mprotect");
}

// Task 1, Part B: Question 3 output:
16384=r, 8dd0000
16385=b, 8dd0001
16386=o, 8dd0002
16387=b, 8dd0003
16388=k, 8dd0004
16389=o, 8dd0005
16390=s, 8dd0006
16391=k, 8dd0007
16392=i, 8dd0008
16393=e, 8dd0009
16394=3, 8dd000a

20480=r, 8dd1000
20481=b, 8dd1001
20482=o, 8dd1002
20483=b, 8dd1003
20484=k, 8dd1004
20485=o, 8dd1005
20486=s, 8dd1006
20487=k, 8dd1007
20488=i, 8dd1008
20489=e, 8dd1009
20490=3, 8dd100a
```

4. Now, create a buffer of 2 pages and try to copy 7th and 8th page into it. Can you copy it, if not why?

   For this experiment, both the 7th and 8th pages can be copied into a second buffer that had the default R/W access, e.g., no mprotect() applied to the two-page buffer. In addition, both the 7th and 8th pages that are being copied, have R/W access enabled.

```
char *p2,*buffer2;  /* Allocate a buffer; it will have the default
                       protection of PROT_READ|PROT_WRITE. */
size=pagesize*2;
p2 = memalign(pagesize,size);    /*Allocating buffer'p2' of size = two pages*/
if (p2 == NULL)
       handle_error("memalign");

memset(p2,0x42,size);     /* Copying 'B' to whole buffer */
buffer2=p2;               /* pointing buffer, 'buffer2' to starting address of p2 */
memcpy(buffer2+1, (buffer+((4096*6)+1)), pagesize*2);

// Task 1, Part B: Question 4 output:
Address of 1 Page: 8dd7000
Address of 2 Page: 8dd8000
0=B, 8dd7000
1=o, 8dd7001
2=b, 8dd7002
3=k, 8dd7003
4=o, 8dd7004
5=s, 8dd7005
6=k, 8dd7006
7=i, 8dd7007
8=e, 8dd7008

4096=B, 8dd8000
4097=o, 8dd8001
4098=b, 8dd8002
4099=k, 8dd8003
4100=o, 8dd8004
4101=s, 8dd8005
4102=k, 8dd8006
4103=i, 8dd8007
4104=e, 8dd8008
```

5. Now try to copy 6th page and 9th page into the previously created buffer. Are you able to do so? If not, when your code hit SIGSEGV, is it copying 6th or 9th page? Explain your answer.

   For this experiment, running in this environment, both the 6th and 9th page can be copied into the second buffer. As was the case with Question 4, the 2nd buffer has R/W access. However, the data in the 6th page on the first buffer has W access enabled (see Question 3). Since R access was not declared in the mprotect() call to protect this chunk of memory, my assumption was this chuck of memory would be W only.

```
memcpy(buffer2, (buffer+((4096*5))), pagesize);
       memcpy(buffer2+4096, (buffer+((4096*8))), pagesize);

// Task 1, Part B: Question 5 output:
0=r, 8dd7000
1=b, 8dd7001
2=o, 8dd7002
3=b, 8dd7003
4=k, 8dd7004
5=o, 8dd7005
6=s, 8dd7006
7=k, 8dd7007
8=i, 8dd7008
9=e, 8dd7009
10=3, 8dd700a

4096=R, 8dd8000
4097=o, 8dd8001
4098=b, 8dd8002
4099=e, 8dd8003
4100=r, 8dd8004
4101=t, 8dd8005
4102=A, 8dd8006
4103=A, 8dd8007
4104=A, 8dd8008
4105=A, 8dd8009
4106=A, 8dd800a
4107=A, 8dd800b
4108=A, 8dd800c
4109=A, 8dd800d
4110=A, 8dd800e
```

An additional experiment was run by setting the protection on the 2nd buffer to R only. In this scenario, SIGSEGV was thrown on write access to 2nd buffer:

```
if (mprotect(p2, pagesize+2, PROT_READ)==-1) {      //Verify R access on 2 pages
      handle_error("mprotect");
}
printf("BUFFER-2: VERIFY READ ACCESS, 1st page\n");
for(i=0; i<15; i++) {     //print 1st page
      // for(i=pagesize; i<((pagesize)+15); i++) {        //print 2nd page
      printf("%d=%c, %lx\n",i,*(p2+i),p2+i);
}

// Task 1, Part B: Question 5 output:
Address of 1 Page: 9b16000
Address of 2 Page: 9b17000
BUFFER-2: VERIFY READ ACCESS, 1st page
0=B, 9b16000
1=B, 9b16001
Got SIGSEGV at address: 0x9b16001
```

III. Task 2: Non-Executable Stack

This task is also divided into two parts, (A) Understanding the importance of non-executable stack, and (B) Experimenting with vulnerable code with protected and unprotected stack.

Part A: Understanding Stack Protection

1. Stack smashing via buffer overflow.

   A stack buffer overflow occurs when a program writes to a memory address on the program's call stack that's outside of the intended memory allocation (buffer). A malicious actor (untrusted host) can compromise the Trusted Compute Base (TCB) Principle of Least Privilege and gain access to privileged data or protected system calls by smashing (overfilling) a buffer on the stack [1].

2. Stack canary.

   A stack canary is used to detect a stack buffer overflow before execution of malicious code can occur. This method works by placing a small randomly chosen integer in memory just before the stack return pointer.

   Since most buffer overflows overwrite memory from lower to higher memory addresses, overwriting the return pointer (and thus violating the TCB), the canary value must also be overwritten. Verifying the canary value prior to a function call that uses the return pointer on the stack can make it ore difficult for a malicious actor to implement the stack buffer overflow exploitation [2].

3. NX (Non-Executable Stack).

   Executable stack protection (NX) marks memory locations (buffers) as non-executable. This is typically accomplished by setting an (NX) bit in the operating systems hardware. NX offers protection from buffer-overflow attacks that rely on some part of memory that's both writeable and executable. Some examples are the Sasser and Blaster worms that inject executable code into the TCB [3].

4. Address space layout randomization (ASLR).

   Address space layout randomization (ASLR) provides security against memory exploitation by randomizing memory locations for data. Random allocation of memory-addresses obscures memory-addresses from attacks that violate the TCB. Malicious actors will need to guess the location of protected data and/or programs. This increases the work factor (Economics of Security) for malicious actors [4].

Part B: Experiments with execution of code

vuln.c compiled with the following gcc options:

- `gcc -g -O0 -fno-stack-protector -z execstack -o vuln-nossp-exec vuln.c`
- `gcc -g -O0 -fno-stack-protector -o vuln-nossp-noexec vuln.c`
- `gcc -g -O0 -z execstack -o vuln-ssp-exec vuln.c`
- `gcc -g -O0 -o vuln-ssp-noexec vuln.c`

1. Explaining the functionality of -fno-stack-protector and -z execstack options of gcc. Explain the differences you see in the binaries when the binaries are created with and without these options.

   - The -fno-stack-protector disables Stack Guard protection.
   - The `-z execstack` flag marks the stack executable. By default, gcc will mark the stack as non-executable.
   - The `ls -l` command shows the binaries that have `stack-protector` enabled, e.g., not disabled are slightly larger than the binaries where `stack-protector` is disabled. The `execstack` option has no effect on the size or file permissions for the binaries. However, running `md5sum` against the binary files shows they are unique.

```
$ ls -l
-rwxrwxr-x 1 project1 8460 Jan 27 05:26 vuln-nossp-exec
-rwxrwxr-x 1 project1 8460 Jan 27 05:26 vuln-nossp-noexec
-rwxrwxr-x 1 project1 8512 Jan 27 05:27 vuln-ssp-exec
-rwxrwxr-x 1 project1 8512 Jan 27 05:27 vuln-ssp-noexec

$ md5sum vuln-nossp-exec
b3652b2edf7040f1846f3b4538e4d9bb  vuln-nossp-exec
$ md5sum vuln-nossp-noexec
0dae752622f829e5b2372097df34dc87  vuln-nossp-noexec
p $ md5sum vuln-ssp-exec
0100b9cec2ac3c0b047c07548999d469  vuln-ssp-exec
$ md5sum vuln-ssp-noexec
e9fdd37cd3a0fbad71eb4d2dcecddb64  vuln-ssp-noexec
```

   Using gdb to examine the binaries shows that compiling with stack protection adds two parts to the assembly code. For this example, and for brevity, I chose to look at stack protection and not the z `execstack` option. Analysis of the assembler code points to the introduction of some variable (**`0x14`**), most likely the canary bit in the first part of the code. Toward the end of the dump, there seems to be a comparison to verify the canary bit (**`0x14`**) using the function `stack_chk_fail`:

```
$ gdb ./vuln-ssp-exec
(gdb) disas main
Dump of assembler code for function main:
   0x080484eb <+0>:     lea    0x4(%esp),%ecx
   0x080484ef <+4>:     and    $0xfffffff0,%esp
   0x080484f2 <+7>:     pushl  -0x4(%ecx)
   0x080484f5 <+10>:    push   %ebp
   0x080484f6 <+11>:    mov    %esp,%ebp
   0x080484f8 <+13>:    push   %ecx
   0x080484f9 <+14>:    sub    $0x34,%esp
   0x080484fc <+17>:    mov    %ecx,%eax
   0x080484fe <+19>:    mov    0x4(%eax),%eax
   0x08048501 <+22>:    mov    %eax,-0x2c(%ebp)
   0x08048504 <+25>:    mov    %gs:0x14,%eax
   0x0804850a <+31>:    mov    %eax,-0xc(%ebp)
   0x0804850d <+34>:    xor    %eax,%eax
   0x0804850f <+36>:    sub    $0xc,%esp
   0x08048512 <+39>:    push   $0x8048610
   0x08048517 <+44>:    call   0x80483b0 <puts@plt>
   0x0804851c <+49>:    add    $0x10,%esp
   0x0804851f <+52>:    sub    $0x8,%esp
   0x08048522 <+55>:    lea    -0x24(%ebp),%eax
   0x08048525 <+58>:    push   %eax
   0x08048526 <+59>:    push   $0x804861c
   0x0804852b <+64>:    call   0x80483d0 <__isoc99_scanf@plt>
   0x08048530 <+69>:    add    $0x10,%esp
   0x08048533 <+72>:    sub    $0x8,%esp
   0x08048536 <+75>:    push   $0x804861f
   0x0804853b <+80>:    lea    -0x24(%ebp),%eax
   0x0804853e <+83>:    push   %eax
   0x0804853f <+84>:    call   0x8048390 <strcmp@plt>
   0x08048544 <+89>:    add    $0x10,%esp
---Type <return> to continue, or q <return> to quit---
   0x08048547 <+92>:    test   %eax,%eax
   0x08048549 <+94>:    jne    0x804855d <main+114>
   0x0804854b <+96>:    sub    $0xc,%esp
   0x0804854e <+99>:    push   $0x8048626
   0x08048553 <+104>:   call   0x80483b0 <puts@plt>
   0x08048558 <+109>:   add    $0x10,%esp
   0x0804855b <+112>:   jmp    0x804856d <main+130>
   0x0804855d <+114>:   sub    $0xc,%esp
   0x08048560 <+117>:   push   $0x804863a
   0x08048565 <+122>:   call   0x80483b0 <puts@plt>
   0x0804856a <+127>:   add    $0x10,%esp
   0x0804856d <+130>:   mov    $0x0,%eax
   0x08048572 <+135>:   mov    -0xc(%ebp),%edx
   0x08048575 <+138>:   xor    %gs:0x14,%edx
   0x0804857c <+145>:   je     0x8048583 <main+152>
   0x0804857e <+147>:   call   0x80483a0 <__stack_chk_fail@plt>
   0x08048583 <+152>:   mov    -0x4(%ebp),%ecx
   0x08048586 <+155>:   leave
   0x08048587 <+156>:   lea    -0x4(%ecx),%esp
   0x0804858a <+159>:   ret
End of assembler dump.
```

```
$ gdb ./vuln-nossp-exec
(gdb) disas main
Dump of assembler code for function main:
   0x0804848b <+0>:     lea    0x4(%esp),%ecx
   0x0804848f <+4>:     and    $0xfffffff0,%esp
   0x08048492 <+7>:     pushl  -0x4(%ecx)
   0x08048495 <+10>:    push   %ebp
   0x08048496 <+11>:    mov    %esp,%ebp
   0x08048498 <+13>:    push   %ecx
   0x08048499 <+14>:    sub    $0x24,%esp
   0x0804849c <+17>:    sub    $0xc,%esp
   0x0804849f <+20>:    push   $0x8048590
   0x080484a4 <+25>:    call   0x8048350 <puts@plt>
   0x080484a9 <+30>:    add    $0x10,%esp
   0x080484ac <+33>:    sub    $0x8,%esp
   0x080484af <+36>:    lea    -0x20(%ebp),%eax
   0x080484b2 <+39>:    push   %eax
   0x080484b3 <+40>:    push   $0x804859c
   0x080484b8 <+45>:    call   0x8048370
<__isoc99_scanf@plt>
   0x080484bd <+50>:    add    $0x10,%esp
   0x080484c0 <+53>:    sub    $0x8,%esp
   0x080484c3 <+56>:    push   $0x804859f
   0x080484c8 <+61>:    lea    -0x20(%ebp),%eax
   0x080484cb <+64>:    push   %eax
   0x080484cc <+65>:    call   0x8048340 <strcmp@plt>
   0x080484d1 <+70>:    add    $0x10,%esp
   0x080484d4 <+73>:    test   %eax,%eax
   0x080484d6 <+75>:    jne    0x80484ea <main+95>
   0x080484d8 <+77>:    sub    $0xc,%esp
   0x080484db <+80>:    push   $0x80485a6
   0x080484e0 <+85>:    call   0x8048350 <puts@plt>
   0x080484e5 <+90>:    add    $0x10,%esp
---Type <return> to continue, or q <return> to quit---
   0x080484e8 <+93>:    jmp    0x80484fa <main+111>
   0x080484ea <+95>:    sub    $0xc,%esp
   0x080484ed <+98>:    push   $0x80485ba
   0x080484f2 <+103>:   call   0x8048350 <puts@plt>
   0x080484f7 <+108>:   add    $0x10,%esp
   0x080484fa <+111>:   mov    $0x0,%eax
   0x080484ff <+116>:   mov    -0x4(%ebp),%ecx
   0x08048502 <+119>:   leave
   0x08048503 <+120>:   lea    -0x4(%ecx),%esp
   0x08048506 <+123>:   ret
End of assembler dump.
```

2. Which of the above four binaries can you exploit, by smashing the stack and overflowing a buffer?

   The binaries compiled without stack protection [`vuln-nossp-exec, vuln-nossp-noexec`] can be exploited by smashing the stack and overflowing the buffer[5]. Note that stack protection is enabled by default and "`*** stack smashing detected ***`" by the binaries: `vuln-ssp-exec` and `vuln-ssp-noexec`:

```
// Task 2, Part B: Question 2 output:

/* ================================== */
$ ./vuln-nossp-exec
Vulnerable
abcdefghijklmnopqrstuvwxyz
Invalid Pass, Try Again!
$ ./vuln-nossp-exec
Vulnerable
abcdefghijklmnopqrstuvwxyz-ab
Invalid Pass, Try Again!
Segmentation fault (core dumped)

/* ================================== */
$ ./vuln-nossp-noexec
Vulnerable
abcdefghijklmnopqrstuvwxyz
Invalid Pass, Try Again!
$ ./vuln-nossp-noexec
Vulnerable
abcdefghijklmnopqrstuvwxyz-ab
Invalid Pass, Try Again!
Segmentation fault (core dumped)

/* ================================== */
$ ./vuln-ssp-exec
Vulnerable
abcdefghijklmnopqrstuvwxyz
Invalid Pass, Try Again!
*** stack smashing detected ***: ./vuln-ssp-exec terminated
Aborted (core dumped)

/* ================================== */
$ ./vuln-ssp-noexec
Vulnerable
abcdefghijklmnopqrstuvwxyz
Invalid Pass, Try Again!
*** stack smashing detected ***: ./vuln-ssp-noexec terminated
Aborted (core dumped)

/* ================================== */
$ gcc -w vuln.c -o vuln
$ ./vuln
Vulnerable
abcdefghijklmnopqrstuvwxyz
Invalid Pass, Try Again!
*** stack smashing detected ***: ./vuln terminated
Aborted (core dumped)
```

3. Attempt to find the stack canary value in vuln-ssp-exec binary, by using a debugger like gdb. Does the canary value change or remain the same across multiple compilations/ executions? Post screenshots of your attempts.

   To accomplish canary identification, I ran experiments utilizing gdb against a binary (vuln-ssp-exec) compiled with stack protection enabled. These experiments indicate that the value for the canary bit changes across execution and by extension compilation:

```
/* 1. Run binary and locate a point to set a break: The mov function right before the xor compare function.
 * 2. Set the break point: break *0x0804850a
 * 3. Run the binary: r
 * 4. Examine the eax register: info r eax
 * Repeat from step 3. */

$ gdb vuln-ssp-exec
(gdb) disas main
Dump of assembler code for function main:
   0x080484eb <+0>:     lea    0x4(%esp),%ecx
   0x080484ef <+4>:     and    $0xfffffff0,%esp
   0x080484f2 <+7>:     pushl  -0x4(%ecx)
   0x080484f5 <+10>:    push   %ebp
   0x080484f6 <+11>:    mov    %esp,%ebp
   0x080484f8 <+13>:    push   %ecx
   0x080484f9 <+14>:    sub    $0x34,%esp
   0x080484fc <+17>:    mov    %ecx,%eax
   0x080484fe <+19>:    mov    0x4(%eax),%eax
   0x08048501 <+22>:    mov    %eax,-0x2c(%ebp)
   0x08048504 <+25>:    mov    %gs:0x14,%eax
   0x0804850a <+31>:    mov    %eax,-0xc(%ebp)
   0x0804850d <+34>:    xor    %eax,%eax
End of assembler dump.
(gdb) break *0x0804850a
Breakpoint 1 at 0x804850a: file vuln.c, line 7.
(gdb) r
Starting program: /home/project1/Desktop/Project_1/Stack_protection/vuln-ssp-exec

Breakpoint 1, 0x0804850a in main (argc=1, argv=0xbffff614) at vuln.c:7
7       {
(gdb) info r eax
eax            0xb1f7cf00       -1309159680
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/project1/Desktop/Project_1/Stack_protection/vuln-ssp-exec

Breakpoint 1, 0x0804850a in main (argc=1, argv=0xbffff614) at vuln.c:7
7       {
(gdb) info r eax
eax            0x7fe03100       2145399040
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/project1/Desktop/Project_1/Stack_protection/vuln-ssp-exec

Breakpoint 1, 0x0804850a in main (argc=1, argv=0xbffff614) at vuln.c:7
7       {
(gdb) info r eax
eax            0xfb977100       -73961216
(gdb)
```

IV. CONCLUSION

The experiments in Task 1, Parts [A, B] demonstrated:

- Read access with write protection on a block of memory:

```
if (mprotect(p+pagesize, pagesize, PROT_READ)==-1) {
       handle_error("mprotect");
}
```

- Read and write protection on a block of memory:

```
if (mprotect(p+pagesize, pagesize, PROT_NONE)==-1) {
       handle_error("mprotect");
}
```

However, write only access with read protection was not able to be set (see Task 1, Part B, Questions[3, 5] for experiments):

```
if (mprotect(p+pagesize, pagesize, PROT_WRITE)==-1) {
       handle_error("mprotect");
}
```

The reason why write only access was not working could be a function of the operating system or kernel. From `man mprotect` f:

```
Whether PROT_EXEC has any effect different from PROT_READ depends on
processor architecture, kernel version, and process state.   If
READ_IMPLIES_EXEC is  set in  the process's personality flags (see
personality(2)), specifying PROT_READ will implicitly add PROT_EXEC.

On some hardware architectures (e.g., i386), PROT_WRITE implies
PROT_READ.
```

I. APPENDIX

This section contains miscellaneous experiments and some use gdb commands.

```
/* ================================
 * Some useful gdb commands
 * ================================ */
(gdb) break main
(gdb) r
(gdb) ni
(gdb) disas main
(gdb) p $esp
(gdb) info registers
(gdb) x/x $ecx


/* ================================
 * Some additional experiments
 * ================================ */
$ python -c 'print "B"*30' > LongPass
$ gdb vuln-ssp-exec
(gdb) r <LongPass
Starting program: /home/project1/Desktop/Project_1/Stack_protection/vuln-ssp-exec
<LongPass
Vulnerable
Invalid Pass, Try Again!
*** stack smashing detected ***: /home/project1/Desktop/Project_1/Stack_protection/vuln-
ssp-exec terminated

Program received signal SIGABRT, Aborted.
0xb7fd9d05 in __kernel_vsyscall ()
(gdb)

$ gdb vuln-nossp-exec
(gdb) r <LongPass
Starting program: /home/project1/Desktop/Project_1/Stack_protection/vuln-nossp-exec
<LongPass
Vulnerable
Invalid Pass, Try Again!

Program received signal SIGSEGV, Segmentation fault.
0x08048506 in main (
    argc=<error reading variable: Cannot access memory at address 0xbf004242>,
    argv=<error reading variable: Cannot access memory at address 0xbf004246>)
    at vuln.c:17
17      }
(gdb)
```

[1] https://en.wikipedia.org/wiki/Stack_buffer_overflow
https://en.wikipedia.org/wiki/Buffer_overflow
https://www.tenouk.com/Bufferoverflowc/bufferoverflowvulexploitdemo32.html
[2] https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries
[3] https://en.wikipedia.org/wiki/Executable_space_protection
[4] https://en.wikipedia.org/wiki/Address_space_layout_randomization
[5] https://www.youtube.com/watch?v=crZtO32pHq8
https://www.youtube.com/watch?v=V9lMxx3iFWU
https://www.youtube.com/watch?v=1S0aBV-Waeo