Title of the Experiment: Buffer Overflow

Group Members' Names: Sujan Kumar Saha, Pankaj Bhowmik

Date: 08/27/2019

Abstract

In C, strcpy(), gets() functions copy strings without checking the length of the strings which can create a buffer overflow. Using buffer overflow, software attackers can steal passwords or can inject malicious code in the program. In this experiment, we studied buffer overflow attacks. We run the given codes, observed how buffer overflow occurred and modified the code to prevent those overflow attacks. In part 1 of the experiment, we ran a simple C code with no overflow checking. We observed how illegal access can be granted due to the overflow. Different interactions can be taken to prevent this kind of overflow. The solution is implemented by modifying the C code example and observed the change of the behaviour of the program. In the second part of the experiment, we ran another C code example where stack overflow has occured due to buffer overrun. In this code, it has been shown that how buffer overflow can be used to run a malicious code instead of a good code. We observed the impact of malicious code injection by exploiting the overflow. We replaced the unsafe code with a safe one to prevent this kind of attacks. We learned about buffer overflow in this experiment and learned how to code in C in a secured manner.

Experimental Results

Goals: The goal of this experiment is to investigate how to carry out a buffer overflow attack and how to prevent those attacks.

Experimental Setup: We used Linux operating system (Ubuntu and CentOS 7) on virtual box on our own computers. We compiled those C codes using gcc compiler and used 32 bit systems.

Part 1:
Experiment Steps:

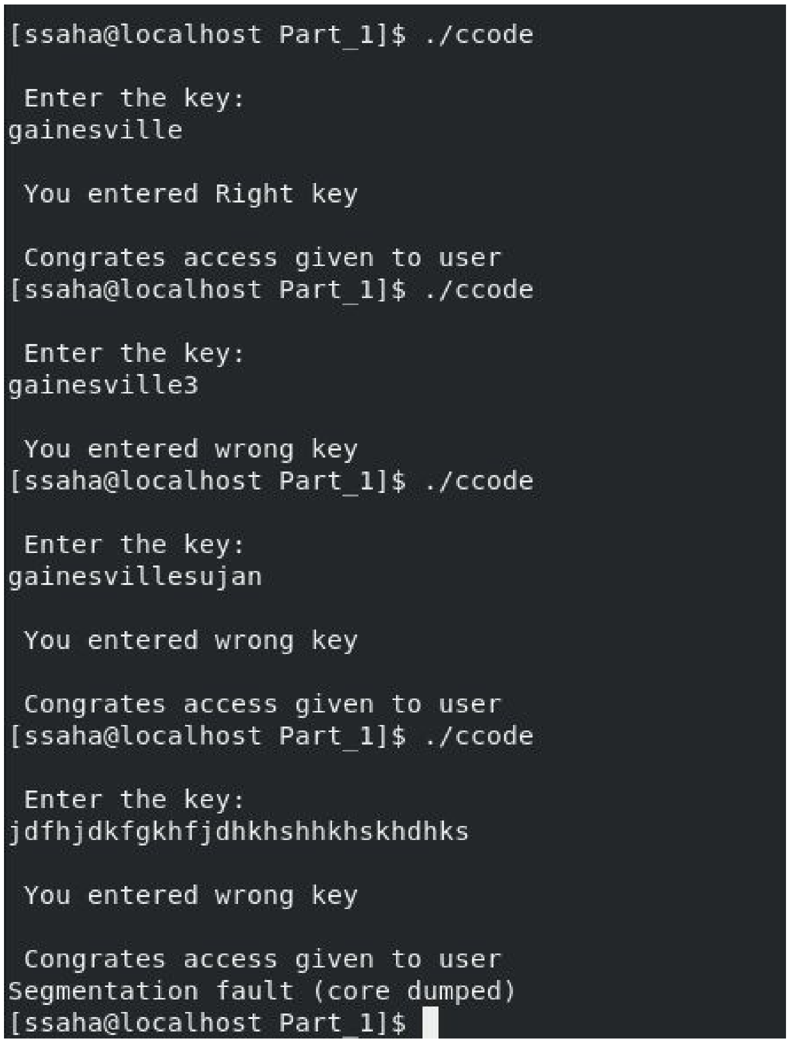
1. The C code ccode.c has been compiled with gcc compiler with the flag -fno-stack-protector.
2. The code has been run with correct key, wrong key, correct key with more characters and a long key

Results and Observations

The code of the first experiment has been done on Centos 7 machine. The outputs screenshot is given below.

1. With the right key "gainesville", it printed the right key and access was given the user. as xyz value is 1.
2. With the wrong key "gainesville3", it printed the wrong key and access was not given. as xyz value is 0.
3. With the wrong key "gainesvillesujan", it printed wrong key but access was given!
4. With the long key, we got the same output as 3 but also got the segmentation fault.
5. **Our Conclusion:** We observed the outputs with different inputs and our conclusion is that if the key is long, buffer overflow happens and an illegal access can be granted even if the key is wrong.
6. **Explanation:** With the right key, access is not given because, xyz value is changed to 1. With the wrong key, xyz value is still 0. So, access is not given. But with long wrong key, access has been granted because

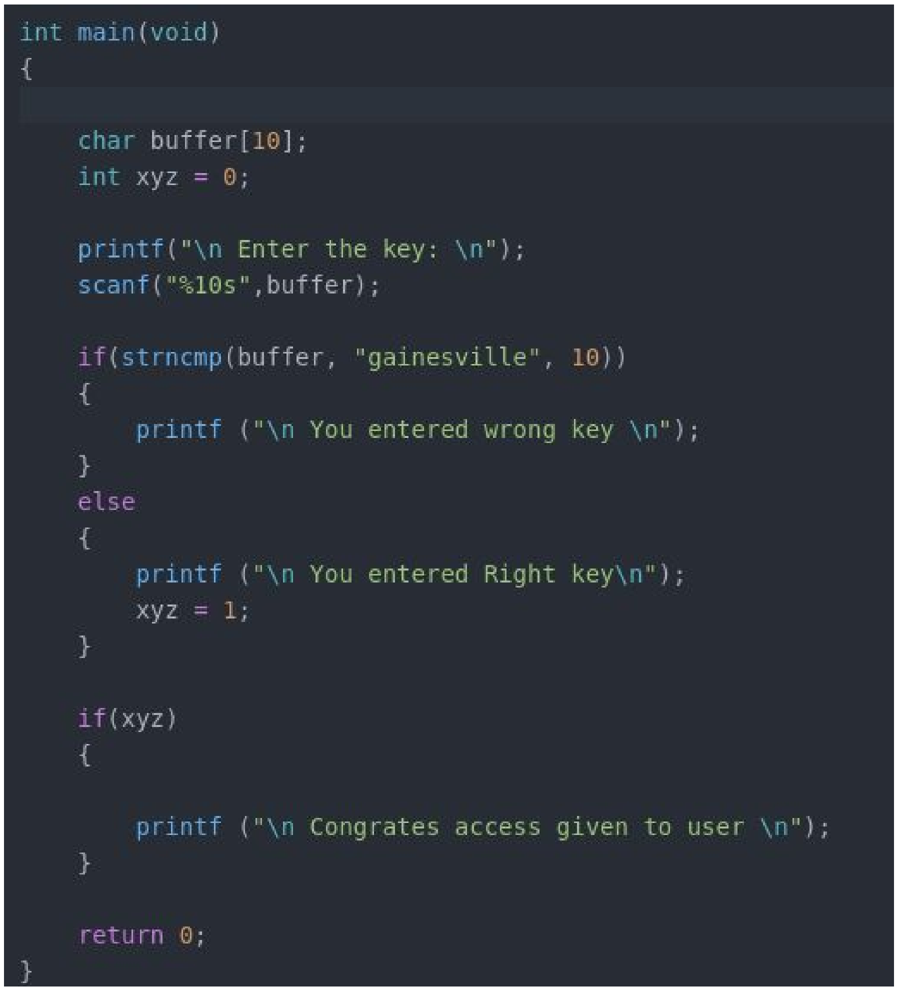
the string is longer. So it did not fit in the “buffer” variable and overflow occurred which replaced the xyz value. So, xyz value is no longer 0. That’s why, access has been granted in the later portion of the code.



```
[ssaha@localhost Part_1]$ ./ccode

 Enter the key:
gainesville

 You entered Right key

 Congrates access given to user
[ssaha@localhost Part_1]$ ./ccode

 Enter the key:
gainesville3

 You entered wrong key
[ssaha@localhost Part_1]$ ./ccode

 Enter the key:
gainesvillesujan

 You entered wrong key

 Congrates access given to user
[ssaha@localhost Part_1]$ ./ccode

 Enter the key:
jdfhjdkfgkhfjdhkhshhkhskhdhks

 You entered wrong key

 Congrates access given to user
Segmentation fault (core dumped)
[ssaha@localhost Part_1]$ 
```

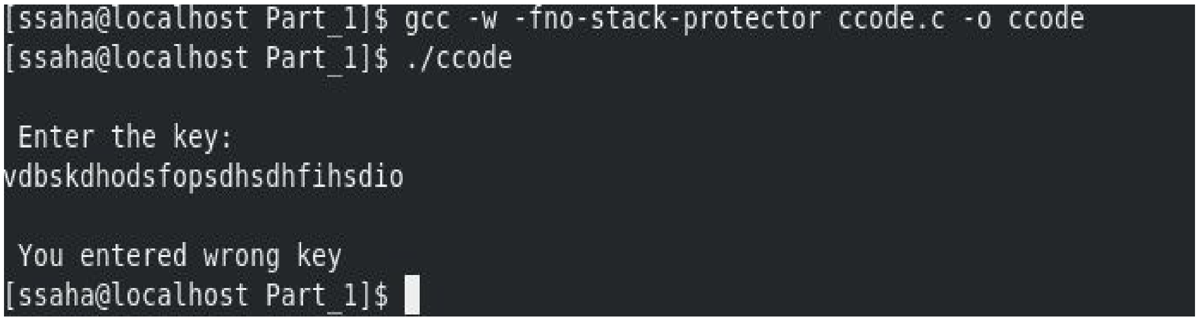
Figure 1: The execution of ccode.c using the right key, wrong key, long key

7. The code can be corrected by using strncmp() function with string size 10. The corrected code and output is given below. Here, even if a long key is given, the access has not been granted.



```
int main(void)
{

    char buffer[10];
    int xyz = 0;

    printf("\n Enter the key: \n");
    scanf("%10s",buffer);

    if(strncmp(buffer, "gainesville", 10))
    {
        printf ("\n You entered wrong key \n");
    }
    else
    {
        printf ("\n You entered Right key\n");
        xyz = 1;
    }

    if(xyz)
    {

        printf ("\n Congrates access given to user \n");
    }

    return 0;
}
```

Figure 2: The code modification of ccode.c to prevent buffer overflow



```
[ssaha@localhost Part_1]$ gcc -w -fno-stack-protector ccode.c -o ccode
[ssaha@localhost Part_1]$ ./ccode

 Enter the key:
vdbskdhodsfopsdhsdhfihsdio

 You entered wrong key
[ssaha@localhost Part_1]$ 
```

Figure 3: Output of ccode.c after using strncmp() instead of strcmp()

Part 2:

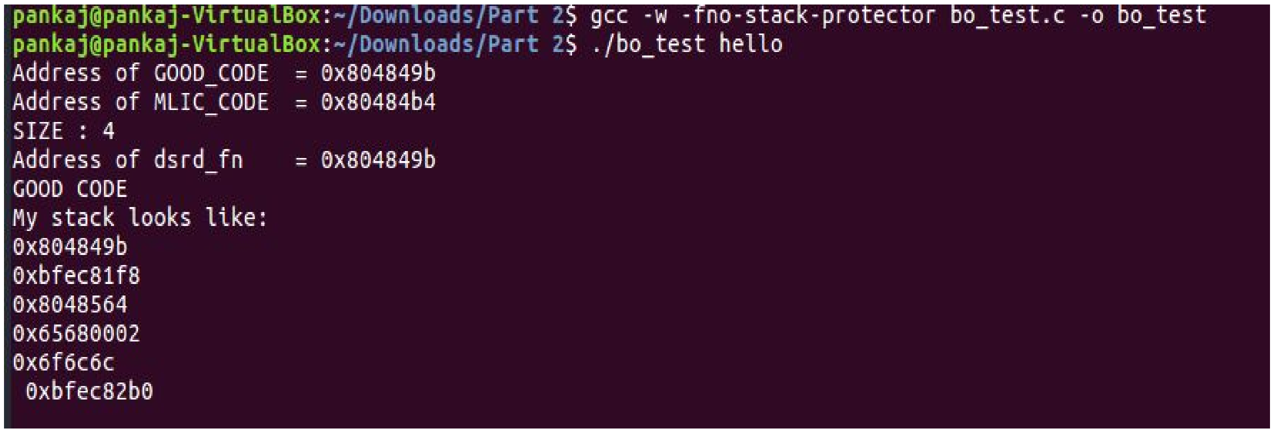
Experiment Steps:

1. The given C code bo_test.c has been compiled with the given flag -fno-stack-protector. We run this code on ubuntu instead of CentOS 7.
2. The code has been run with an argument less than 10 characters. The code runs normally and address of good code and malicious code is printed.

3. The address of malicious code is used as an input argument for the next run where we used a small perl script to run the code.

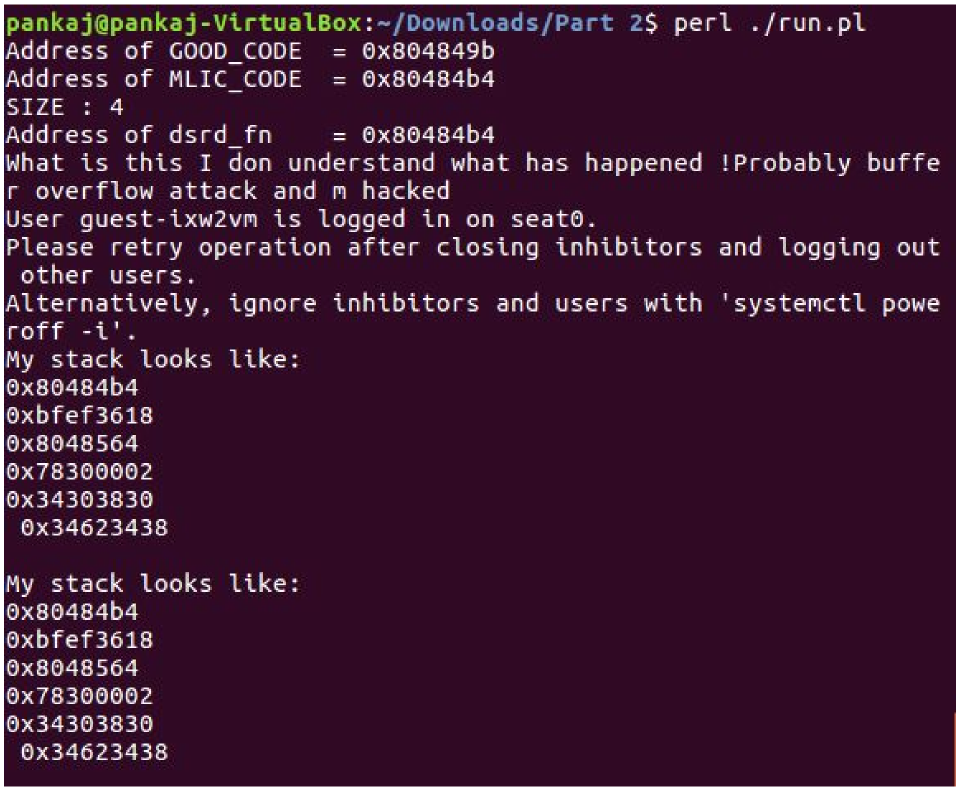
Results and Observations:

1. In the first run of the code, we can see that "GOOD CODE" has been printed and the address of dsrd_fn is same as GOOD_CODE. It is happening because the buffer input we are giving is less than the buffer size.
2. In the second run of the code, we are using the address of the malicious code as input argument. We are giving that argument using a small perl script. Here, the malicious code has been printed instead of good code and the address of dsrd_fn is same as MILC_CODE. It is happening because the dsrd_fn function pointer declaration is just after the data[10] variable declaration. So, when input string is long enough to make an overflow in data variable and replace the address of dsrd_fn to malicious code. That's why malicious code has been printed instead of good code.
3. The code can be modified by using strncpy() instead of strcpy(). The modified code is given below and the output prints good code.



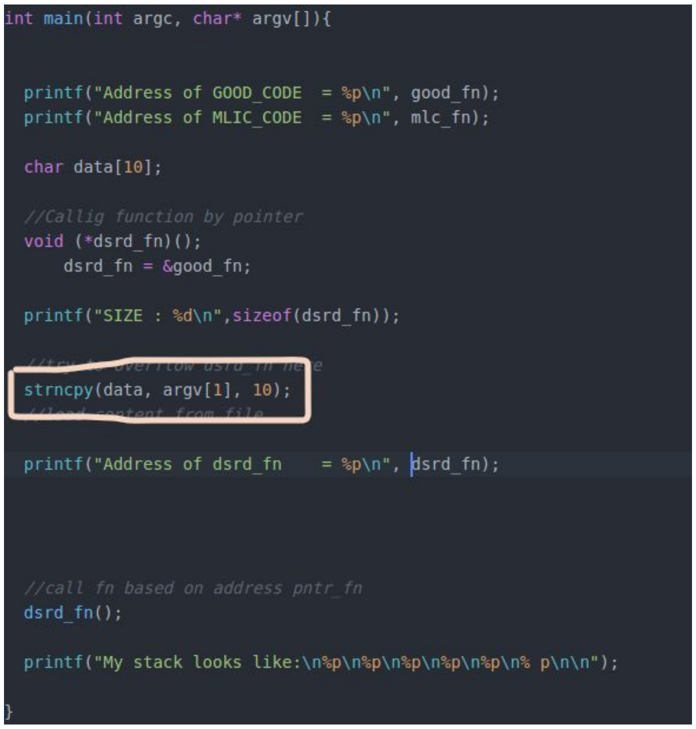
```
pankaj@pankaj-VirtualBox:~/Downloads/Part 2$ gcc -w -fno-stack-protector bo_test.c -o bo_test
pankaj@pankaj-VirtualBox:~/Downloads/Part 2$ ./bo_test hello
Address of GOOD_CODE  = 0x804849b
Address of MLIC_CODE  = 0x80484b4
SIZE : 4
Address of dsrd_fn    = 0x804849b
GOOD CODE
My stack looks like:
0x804849b
0xbfec81f8
0x8048564
0x65680002
0x6f6c6c
 0xbfec82b0
```

Figure 4: The execution of bo_test with less than 10 input characters



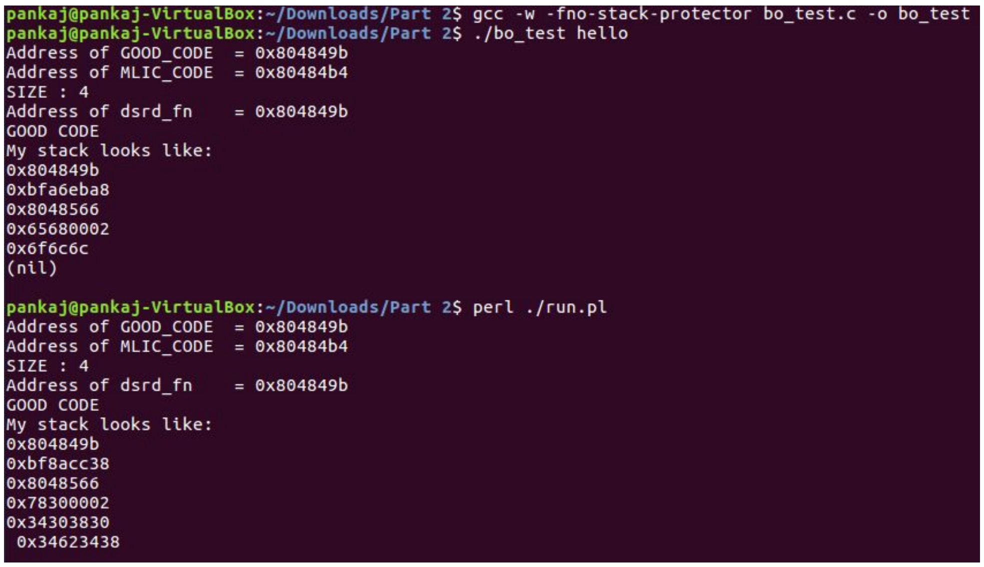
```
pankaj@pankaj-VirtualBox:~/Downloads/Part 2$ perl ./run.pl
Address of GOOD_CODE  = 0x804849b
Address of MLIC_CODE  = 0x80484b4
SIZE : 4
Address of dsrd_fn    = 0x80484b4
What is this I don understand what has happened !Probably buffe
r overflow attack and m hacked
User guest-ixw2vm is logged in on seat0.
Please retry operation after closing inhibitors and logging out
 other users.
Alternatively, ignore inhibitors and users with 'systemctl powe
roff -i'.
My stack looks like:
0x80484b4
0xbfef3618
0x8048564
0x78300002
0x34303830
 0x34623438

My stack looks like:
0x80484b4
0xbfef3618
0x8048564
0x78300002
0x34303830
 0x34623438
```

Figure 5: The execution of run.pl with malicious code address



```
int main(int argc, char* argv[]){


  printf("Address of GOOD_CODE  = %p\n", good_fn);
  printf("Address of MLIC_CODE  = %p\n", mlc_fn);

  char data[10];

  //Callig function by pointer
  void (*dsrd_fn)();
      dsrd_fn = &good_fn;

  printf("SIZE : %d\n",sizeof(dsrd_fn));

  //try to overflow dsrd_fn here
  strncpy(data, argv[1], 10);
  //load content from file

  printf("Address of dsrd_fn    = %p\n", dsrd_fn);




  //call fn based on address pntr_fn
  dsrd_fn();

  printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n% p\n\n");

}
```

Figure 6: The code modification of bo_test.c to prevent buffer overflow



```
pankaj@pankaj-VirtualBox:~/Downloads/Part 2$ gcc -w -fno-stack-protector bo_test.c -o bo_test
pankaj@pankaj-VirtualBox:~/Downloads/Part 2$ ./bo_test hello
Address of GOOD_CODE  = 0x804849b
Address of MLIC_CODE  = 0x80484b4
SIZE : 4
Address of dsrd_fn    = 0x804849b
GOOD CODE
My stack looks like:
0x804849b
0xbfa6eba8
0x8048566
0x65680002
0x6f6c6c
(nil)

pankaj@pankaj-VirtualBox:~/Downloads/Part 2$ perl ./run.pl
Address of GOOD_CODE  = 0x804849b
Address of MLIC_CODE  = 0x80484b4
SIZE : 4
Address of dsrd_fn    = 0x804849b
GOOD CODE
My stack looks like:
0x804849b
0xbf8acc38
0x8048566
0x78300002
0x34303830
 0x34623438
```

Figure 7: The output of bo_test after using strncpy()

Discussion and Summary:

Interesting observation: We observed an interesting scenario regarding this experiment. We ran that on CentOS 7 and the system was shut down. There was no way to print the output. It was happening due to the shutdown command in the malicious code and there is no default mechanism to prevent that shut down on CentOS 7. We ran the code on ubuntu and it ran the malicious code but the system did not shut down.

Summary: By running this experiment, we learned what is a buffer overflow and how an attacker can exploit these unsafe C functions (e.g. gets(), strcpy() etc.) to get an access illegally and insert malicious code. We also learned how to take countermeasures to prevent these kind of attacks such as using strncpy(), strncmp() or fgets(). Another way is to check the string length manually while copying the data to buffers. There are also some tools like Valgrind to check buffer overflow situations in a code. Overall, this experiment is a good learning.