Lab 4: Format Strings

MISM 470 - Dr. Sachin Shetty

Cole Baty, tbaty002@odu.edu, UIN: 01187002

OLD DOMINION UNIVERSITY

2023-11-28

Contents

List of Figures

1 Introduction

This lab entails exploiting format-string vulnerabilities. In the two major task areas, we will achieve the following goals:

- crashing the program
- reading the contents of the program's internal memory
- modifying the contents of the program's internal memory

1.1 Lab setup

This lab was conducted on a SEED Ubuntu 20.04 VM hosted in the author's home lab.

All tasks were conducted with address randomization turned off, per the lab instructions.

1.1.1 Quality of life

The following two `bash` functions were used to facilitate converting between decimal and hexadecimal bases

1.1.1.1 `hex()`

```
function hex() {
    printf "%#x\n" $1
}
```

Figure 1.1: Converting 3735928559 to hexadecimal

1.1.1.2 unhex()

```
function unhex() {
    echo $(( $1 ))
}
```

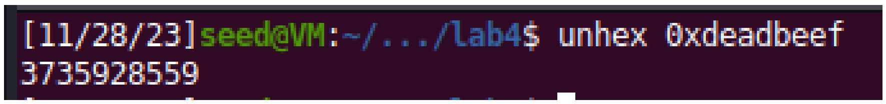


Figure 1.2: Converting 0xdeadbeef to base 10

1.1.1.3 Validating functions

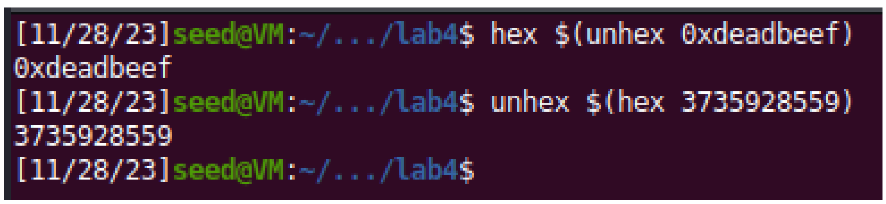


Figure 1.3: Validating that each function is the inverse of the other

Figure 1.3 shows that each of these functions is the inverse of each other.

2 Task 1: Exploit the vulnerability

2.1 Crash the program

One way to crash the program is to trigger a segmentation fault. A segmentation fault occurs when the program attempts to access memory which it is not permitted to access.

Through trial and error, we've determined that a format string consisting of ten `%x` format specifiers is sufficient to reach a memory address which is likely to cause a segmentation fault when accessed.

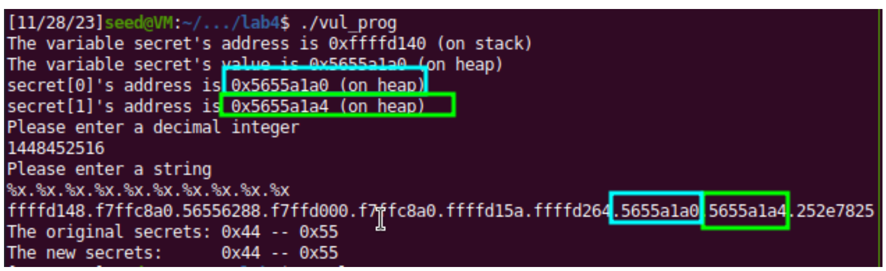


Figure 2.1: Exposing memory contents with format string

In Figure 2.1, we can see the contents of the program's memory exposed by the format string given below. In the blue boxes (top and left), we see the address corresponding to `secret[0]`. In the green boxes (bottom and right), we see the address corresponding to `secret[1]`. This was given in the user input by supplying the decimal representation of the (deliberately) leaked `secret[1]` address in the program output Figure 2.2.

```
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x # '%x.' * 10
```

Figure 2.2: Converting secret1 address to decimal

We note that the address following `secret[1]` is `0x25e7825`. This is a relatively low level address. If we attempt to access this address, e.g. by attempting to read the contents of this address as a string, we should be able to trigger a segmentation fault. We can accomplish this by changing the last format specifier in the format string to `%s`.

Figure 2.3 shows the results of applying this change, successfully resulting in a segmentation fault.

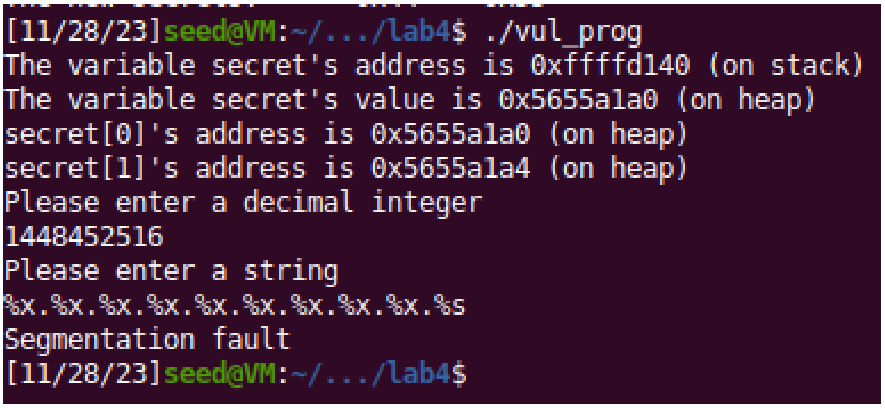


Figure 2.3: Successfully crashing the program

2.2 Print out the `secret[1]` value

In order to print the contents of `secret[1]`, we need to recognize that the base address of `secret` is stored on the stack, and the values stored in `secret[0]` and `secret[1]` are stored on the heap.

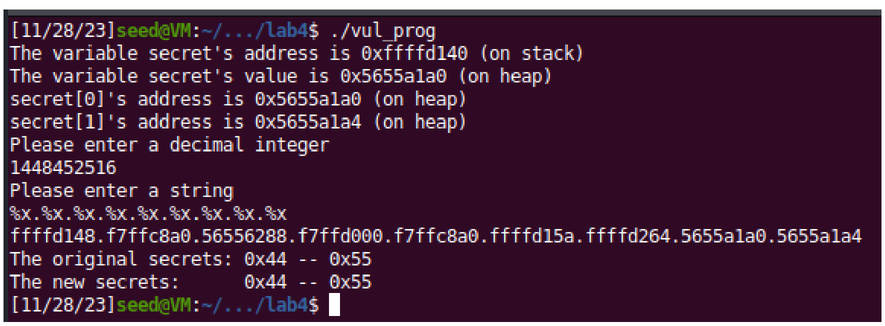


Figure 2.4: Vulnerable program output detailing memory locations of interest

Figure 2.4 shows the output of the vulnerable program provided with the lab. This program deliberately exposes the areas of memory we're concerned with. The variable `secret` is stored on the stack. Because the variable `secret` is instantiated dynamically, the value of this address is a pointer to another address on the heap. This is the base address of `secret`, which coincides with `secret[0]` by definition.

To get a better idea of where user input winds up in relationship to the stack, Figure 2.5 shows the output of the program when supplied with the decimal representation of the hexadecimal value `0xe10c`, which can be seen in the output.

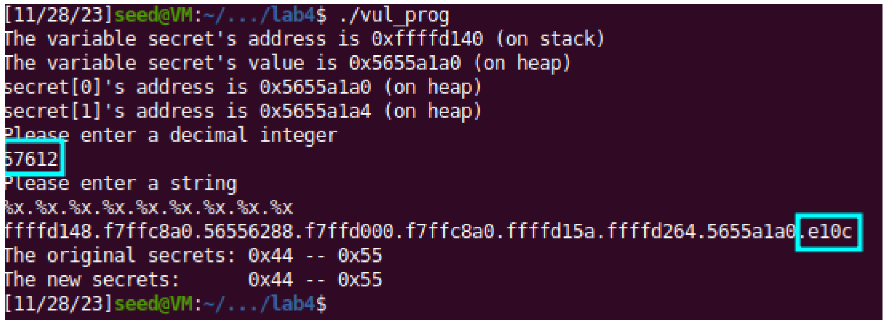


Figure 2.5: Exploring program memory with a known integer value

We note that this value appears directly after the base address for `secret`, shown as `0x5655a1a0` in the program output.

Therefore, to print the value of `secret[1]`, we need to perform supply the following information:

- the decimal representation of the address for `secret[1]`, which is 1448452516.
- a format string which will cause the contents of this memory address to be printed out

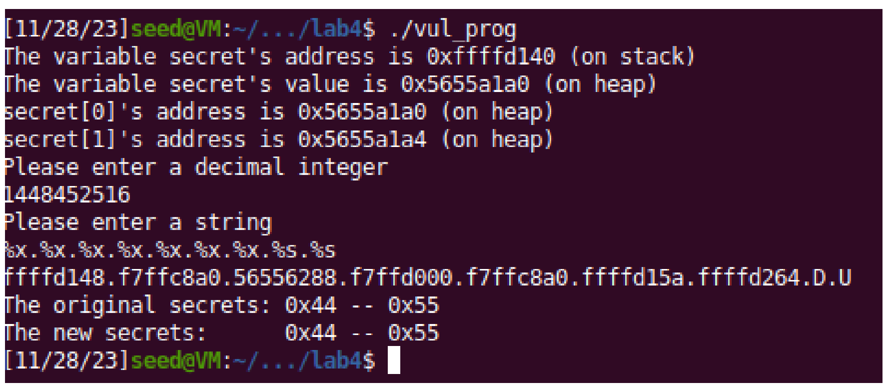


Figure 2.6: Successfully printing contents of secret variable

In Figure 2.6, we can see that the supplied format string prints the contents of both `secret[0]` and `secret[1]`. Note that the characters D and U are represented in ASCII by the hexadecimal values `0x44` and `0x55`, respectively (Fig. 2.7).

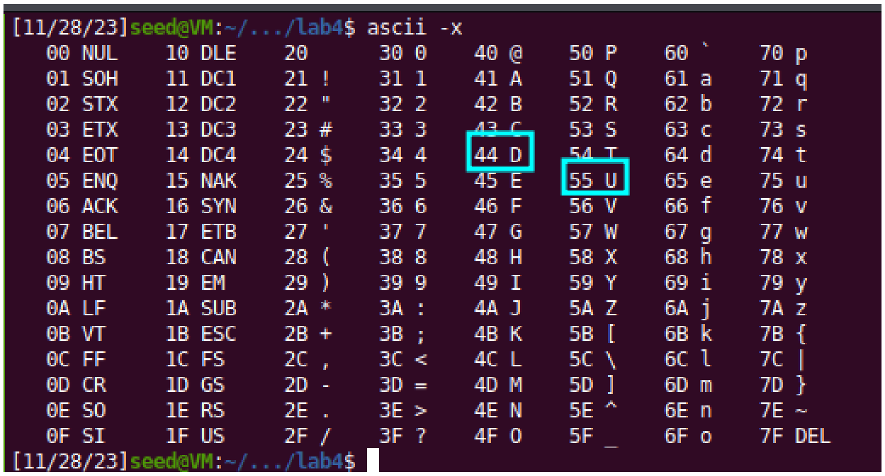


Figure 2.7: ASCII table with D and U shown in hexadecimal

2.3 Modify the `secret[1]` value to an arbitrary value

Modifying the value of `secret[1]` is similar to the approach of printing the value of `secret[1]`. Instead of using the `%s` format specifier in the position corresponding to `secret[1]`, instead we supply the `%n` format specifier. This will write the number of characters printed up to that point into the variable pointed to by the `va_list` pointer internal to the `printf()` function.

By supplying the (decimal) address of `secret[1]`, and a format string containing sufficient `%x` format specifiers, we can write an arbitrary value into `secret[1]`, as shown in 2.8.

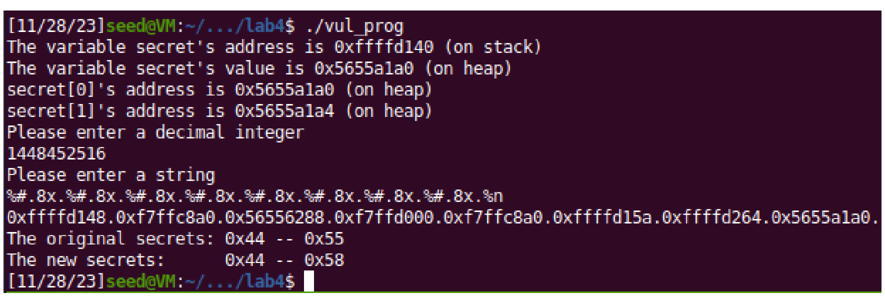


Figure 2.8: Successfully changing contents of secret1 to arbitrary value

Note that the value of `secret[1]` has been changed to `0x58`, corresponding to 88 in decimal. To verify why this should be, note that each group of memory address and trailing period consists of eleven characters. There are eight such groups, for a total of 88 printed characters.

2.4 Modify the `secret[1]` value to 0x500

Note that the hexadecimal value `0x500` corresponds to the decimal value 1280. In order to write this value into `secret[1]`, we will again use the `%n` format specifier, in combination with the field width option incorporated with the `%x` specifier.[1]

Using the field with option, we can specify arbitrary numbers of characters to be printed with the associated format specifier.

Using the format string from the previous section, we can calculate the number of characters we need to specify in a as a field width in order to reach a total number of 1280 characters printed before the `printf()` function processes the `%n` format specifier.

Recall from the previous section that we were able to write the value `0x58` into `secret[1]` with a format string shown in the code snippet below. Each of those groups of memory addresses and periods is eleven printed characters.

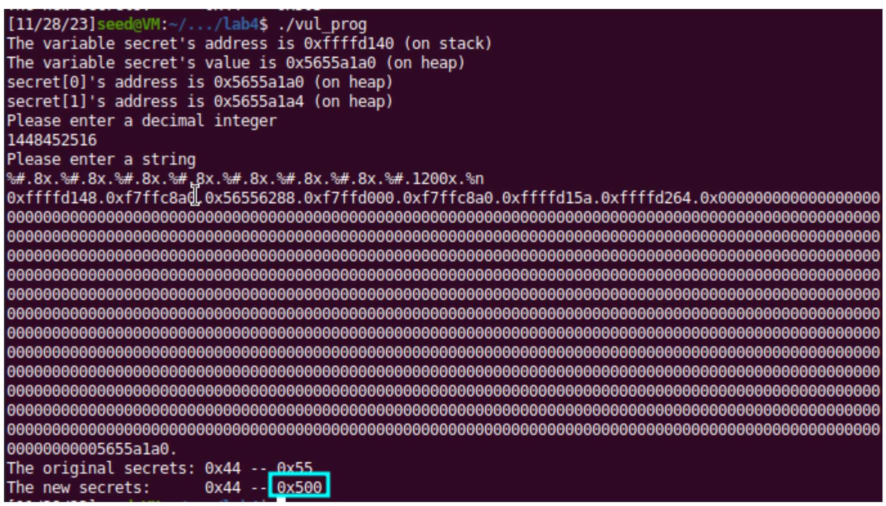


Figure 2.9: Successfully writing 0x500 into secret1

Figure 2.9 shows the results of successfully writing the value `0x500` into `secret[1]`. The structure of the format string is described in the following Python-flavored pseudocode codeblock.

```
7 * 11 = 77 # regular memory addresses printed by %#.8x
3 = len('0x') + len('.') # 'fixed' characters printed by format string

77 + 3 = 80 # total characters

1280 - 80 = 1200 # characters remaining to print
```

[1] “Field width”, printf(3)

3 Task 2: Memory randomization

Although this section of the lab is entitled "Memory randomization", the instructions specify that we are to turn off the address randomization.[1]

Additionally, the instructions for this section specify to remove the `scanf()` statements corresponding to user input for the memory address of `secret[1]` (Fig. 3.1).

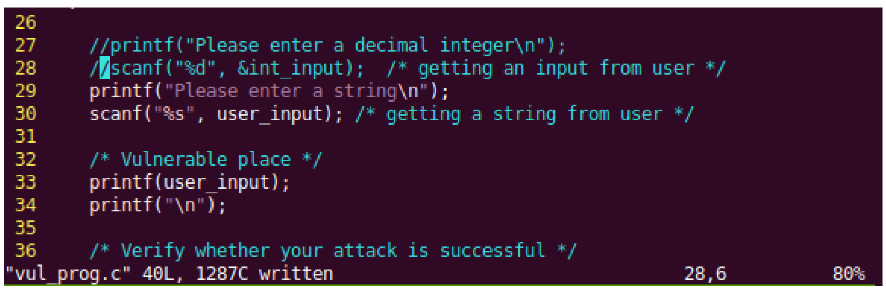


Figure 3.1: Removing scanf code

Since the only input will be the format string, it will be necessary to encode the target memory address in the format string. Although the lab instructions provide a C program to assist in this, previous labs accomplished this same task with the use of native Linux utilities; this is the approach we will use for the remainder of the lab to generate input for the vulnerable program.

3.1 Crash the program

Similar to the previous task, we can explore memory addresses with the `%x` format specifier to identify candidates likely to trigger a segmentation fault.

[1] Lab instructions, page 3.

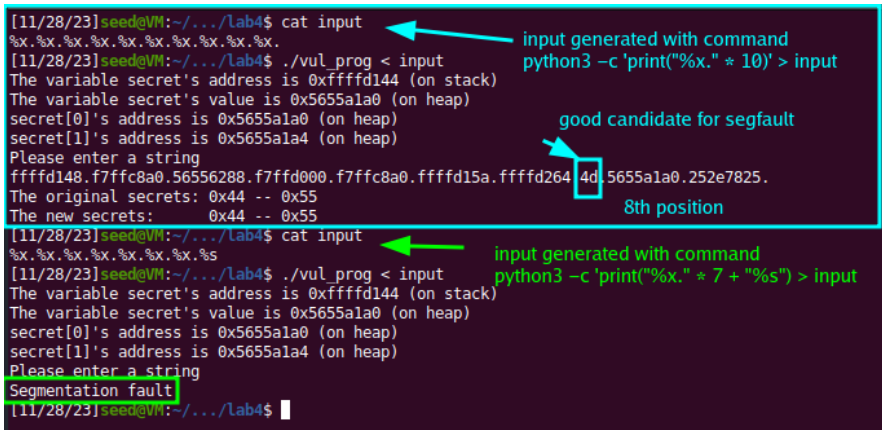


Figure 3.2: Targeting memory address to trigger segfault

In Figure 3.2, we see the process of identifying a memory address likely to trigger a segmentation fault (upper blue box). Note that the memory address shown in the 8th position of the output resulting from the format string corresponds to a very low memory address.

In the lower box, we can see that by adjusting the format string to target this position, we successfully crashed the program.

3.2 Print out the `secret[1]` value

To print out the contents of `secret[1]`, we need to identify where the user input is stored on the stack. To accomplish this, we will craft a format string that contains a value we can easily view, e.g. `0xdeadbeef`. Figure 3.3 shows the process of crafting the format string.

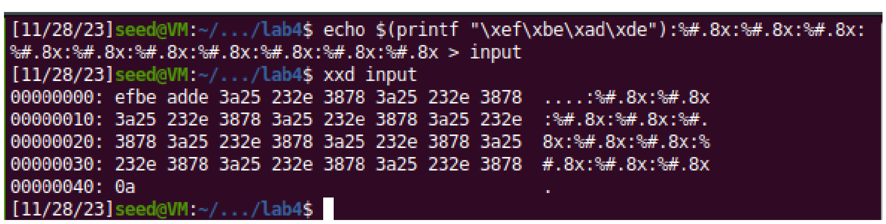


Figure 3.3: Creating format string with known value

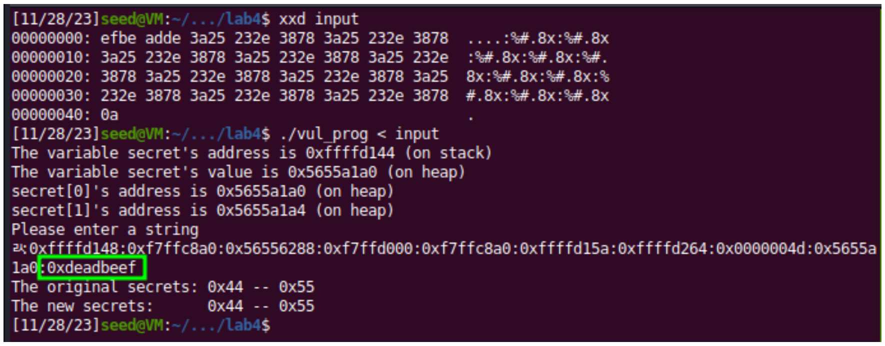


Figure 3.4: Locating supplied known value in output

In Figure 3.4, we can see the value `0xdeadbeef` printed in the eleventh position of the output resulting from the format string. Note that in the tenth position, we see the base address for `secret` of `0x5655a1a0`.

Therefore, if we craft a format string that begins with the address for `secret[1]`, we can adjust the format string to print the value stored there. This is shown in Figure 3.5.

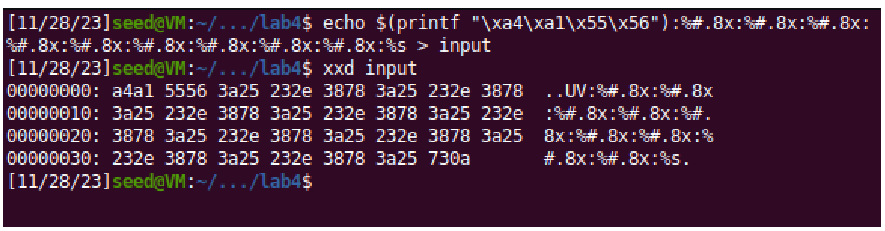


Figure 3.5: Crafting payload to print secret1

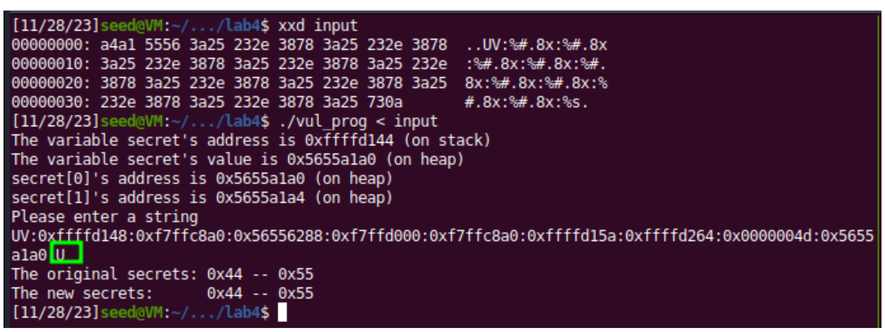


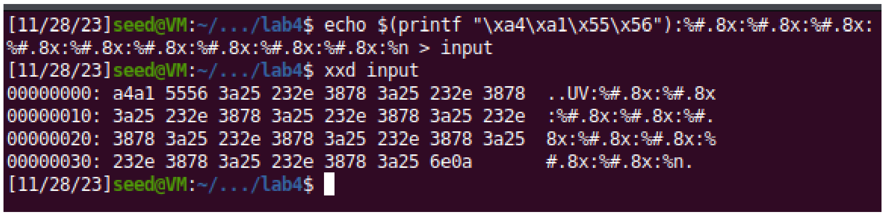
Figure 3.6: Printing value of secret1

Figure 3.6 shows successfully printing the value of `secret[1]` with the format string. Note that the character U in ASCII corresponds to the hexadecimal value `0x55`, as shown above in Figure 2.7.

3.3 Modify the `secret[1]` value to an arbitrary value

In a manner similar to the previous task, we can modify the value of `secret[1]` to an arbitrary value by using %n format specifier. Building on the previous step, we craft a payload which begins with the address of `secret[1]`, and terminates with a %n format specifier.

This process is shown in the following figures.



```
[11/28/23]seed@VM:~/.../lab4$ echo $(printf "\xa4\xa1\x55\x56"):%#.8x:%#.8x:%#.8x:
%#.8x:%#.8x:%#.8x:%#.8x:%#.8x:%#.8x:%n > input
[11/28/23]seed@VM:~/.../lab4$ xxd input
00000000: a4a1 5556 3a25 232e 3878 3a25 232e 3878  ..UV:%#.8x:%#.8x
00000010: 3a25 232e 3878 3a25 232e 3878 3a25 232e  :%#.8x:%#.8x:%#.
00000020: 3878 3a25 232e 3878 3a25 232e 3878 3a25  8x:%#.8x:%#.8x:%
00000030: 232e 3878 3a25 232e 3878 3a25 6e0a       #.8x:%#.8x:%n.
[11/28/23]seed@VM:~/.../lab4$
```

Figure 3.7: Generating format string to overwrite secret1 with arbitrary data

```
[11/28/23]seed@VM:~/.../lab4$ xxd input
00000000: a4a1 5556 3a25 232e 3878 3a25 232e 3878  ..UV:%#.8x:%#.8x
00000010: 3a25 232e 3878 3a25 232e 3878 3a25 232e  :%#.8x:%#.8x:%#.
00000020: 3878 3a25 232e 3878 3a25 232e 3878 3a25  8x:%#.8x:%#.8x:%
00000030: 232e 3878 3a25 232e 3878 3a25 6e0a       #.8x:%#.8x:%n.
[11/28/23]seed@VM:~/.../lab4$ ./vul_prog < input
The variable secret's address is 0xffffd144 (on stack)
The variable secret's value is 0x5655a1a0 (on heap)
secret[0]'s address is 0x5655a1a0 (on heap)
secret[1]'s address is 0x5655a1a4 (on heap)
Please enter a string
UV:0xffffd148:0xf7ffc8a0:0x56556288:0xf7ffd000:0xf7ffc8a0:0xffffd15a:0xffffd264:0x0000004d:0x5655
a1a0:
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x68
[11/28/23]seed@VM:~/.../lab4$
```

Figure 3.8: Overwriting secret1 with format string

Figure 3.8 shows the successful modification of the `secret[1]` value. Note that the value has changed to `0x68`.

3.4 Modify the `secret[1]` value to `0x500`

The approach to writing the value `0x500` into `secret[1]` is very similar to the approach used in the previous task. The format string must again begin with the address of `secret[1]`, and to write exactly the value `0x500` into that address, we must account for the number of characters printed before the %n format specifier is reached.

The Python-flavored pseudocode below describes the structure of the format string which achieves writing the value `0x500` into `secret[1]`.

```
5 = 4 bytes + len(':') # 4 bytes of memory address + colon separator
88 = 8 * (len(address) + len(':'))
3 = len('0x') + len(':')

96 = 5 + 88 + 3 # number of bytes written before %n

1184 = 1280 - 96 # remaining bytes to write
```

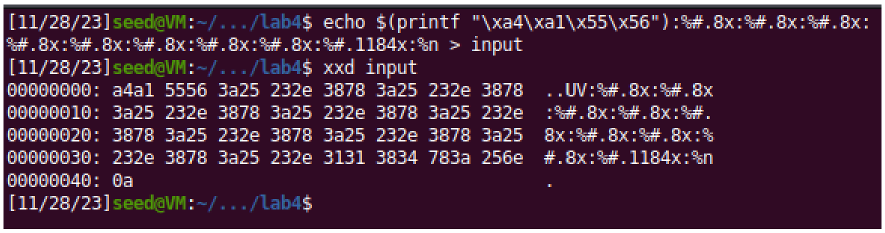


Figure 3.9: Constructing format string to write 0x500 into secret1

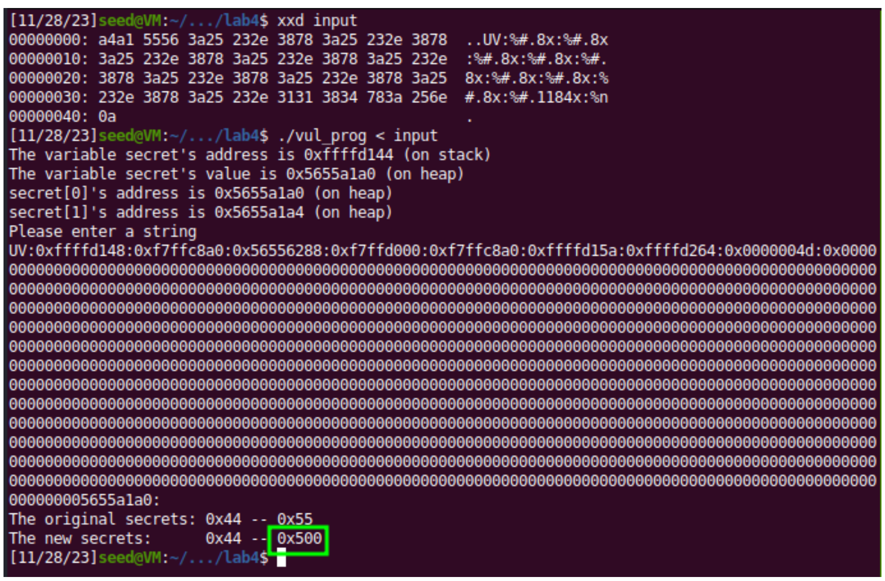


Figure 3.10: Successfully writing 0x500 into secret1 with format string

In Figure 3.10, we see that we have successfully written the value `0x500` into `secret[1]` using a format string.

3.5 Modify the `secret[1]` value to `0xFF990000`

To modify the value of `secret[1]` to `0xff990000`, we applied the techniques described in Chapter 6 of the textbook.

The technique applied in the previous step cannot be directly applied here. This is because, according to the `man` page for `printf()`, the field width option for format specifiers is an `int` value. That is, it is a four-byte signed integer, which is capable of holding the numbers from -2,147,483,648 to 2,147,483,647.[2]

The value `0xff990000` in decimal is 42,882,170,88, which lies outside of this range, so it is not possible to simply print out the ~42 billion characters required to achieve the goal.

Instead, applying the technique described in the textbook, we will use the `%hn` format specifier to write smaller values into each of the "higher" and "lower" memory addresses which comprise `secret[1]` to achieve the goal. According to the `man` page for `printf()`, adding an `h` to the format specifier indicates that "A following integer conversion corresponds to a `short int`...argument". That is, a two-byte integer capable of holding values on the range [-32,768, 32,767] (signed) or [0, 65,535] (unsigned).[3]

In the little-endian system, the "higher" two bytes for `secret[1]` start at `0x5655a1a6`, and the "lower" two bytes begin at `0x5655a1a4`. The problem now becomes writing `0xff99` into the higher two bytes, and `0x0000` into the lower two bytes.

3.5.1 Writing to upper bytes

The hexadecimal value `0xff99` corresponds to 65433 in decimal, which lies on the unsigned range for a `short int`. The Python-flavored pseudocode below describes the structure of the format string which will supply this value in the target address.

```
# addresses at beginning of format string output
# the '@@@@' string is included for output readability
13 = len(4 bytes) + len('@@@@') + len(4 bytes) + len(':')
88 = 8 * (len(address) + len(':')
3 = len('0x') + len(':')


65433 = 65329 + 13 + 88 + 3 # total to write to upper bytes
```

Figure 3.11 shows the process of creating the format string which writes `0xff99` into the upper two bytes of `secret[1]`. Figure 3.12 shows the output of the program with this format string. Since the output contains more than 65 thousand zeroes, the output is truncated using the Linux `tail` utility.

[2] Two's complement: https://en.m.wikipedia.org/wiki/Two%27s_complement
[3] https://en.m.wikipedia.org/wiki/Primitive_data_type#Integer_numbers

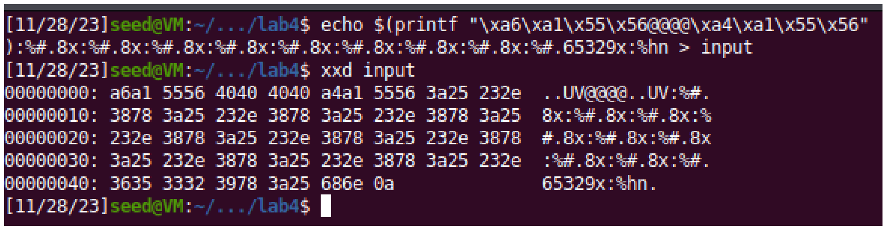


Figure 3.11: Generating format string to write to upper two bytes of secret1

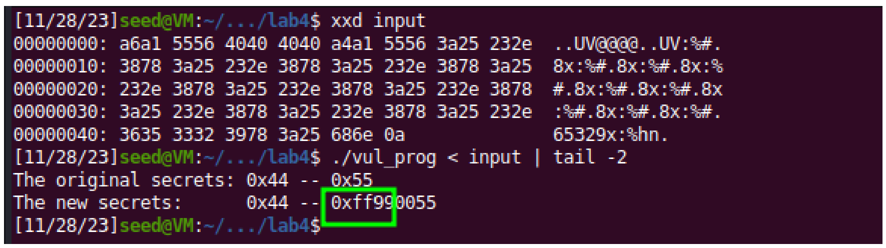


Figure 3.12: Sucessfully writing to upper bytes of secret1

3.5.2 Writing to lower bytes

Writing `0x0000` to the lower bytes of `secret[1]` employs the same technique as writing to the upper bytes, with the added challenge that it's not possible to simply write a string of null bytes, as the null byte is a string terminator which will cause `scanf()` to stop reading the user input string.

In order to write the value `0x0000` to the lower two bytes, we will exploit a fundamental property of storing binary numbers: the overflow.

For example, if we add the numbers `0xffff` and `0x0001`, the sum is `0x10000`, or 65,536 in decimal. However, since we will be writing only to a `short int` that holds only two bytes, it is not possible to store `0x10000` in this space. The result is an 'overflow' which will leave the value `0x0000` in the target address.

The Python-flavored pseudocode below describes the structure of the format string required to accomplish this technique. Note that this includes the structure required to also write to the higher bytes of the target address. Recall that, by definition of the %n format specifier, until this point, we've written `0xff99` or 65,433 characters.

```
#### upper bytes ####
# addresses at beginning of format string output
# the '@@@@' string is included for output readability
13 = len(4 bytes) + len('@@@@') + len(4 bytes) + len(':')
88 = 8 * (len(address) + len(':')
3 = len('0x') + len(':')
```

```
65433 = 65329 + 13 + 88 + 3 # total to write to upper bytes
#### end upper bytes ####

#### lower bytes ####
103 = 65536 - 65433 # target total minus total written so far
3 = len('0x') + len(':') # 'fixed' bytes of remaining %x specifier

100 = 103 - 3 # field width needed to reach `0x10000`

#### end lower bytes ####
```

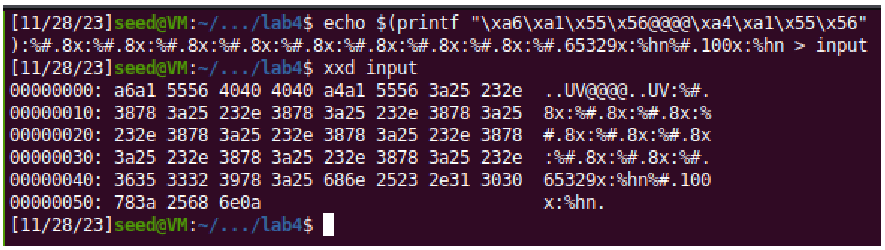


Figure 3.13: Creating format string to populate lower bytes

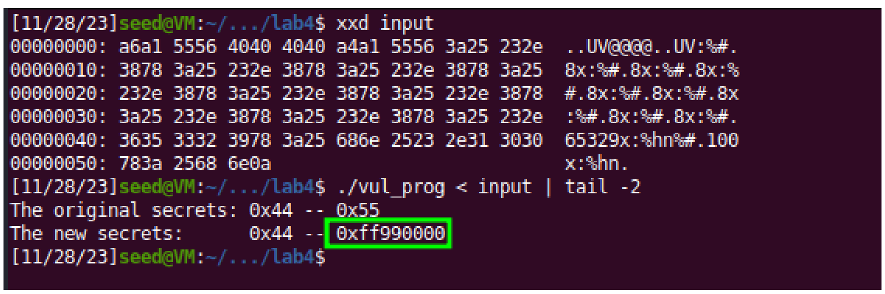


Figure 3.14: Successfully writing 0xff990000 to secret1

Figures 3.13 and 3.14 show the creation of the format string and the successful exploitation of the program with the format string, respectively.