# FIT 3173 Software Security Week 5 Lab: Format String Vulnerability and Symmetric Encryption

## 1 Lab Tasks

The learning objective of this lab is for students to gain the first-hand experience on format-string vulnerability by putting what they have learned about the vulnerability from class into actions. The format-string vulnerability is caused by code like `printf(user_input)`, where the contents of variable of `user_input` is provided by users. When this program is running with privileges (e.g., `Set-UID` program), this `printf` statement becomes dangerous, because it can lead to one of the following consequences: (1) crash the program, (2) read from an arbitrary memory place, and (3) modify the values of in an arbitrary memory place. The last consequence is very dangerous because it can allow users to modify internal variables of a privileged program, and thus change the behavior of the program.

In the following program, you will be asked to provide an input, which will be saved in a buffer called `user_input`. The program then prints out the buffer using `printf`. The program is a `Set-UID` program (the owner is `root`), i.e., it runs with the root privilege. Unfortunately, there is a format-string vulnerability in the way how the `printf` is called on the user inputs. We want to exploit this vulnerability and see how much damage we can achieve.

The program has two secret values stored in its memory, and you are interested in these secret values. However, the secret values are unknown to you, nor can you find them from reading the binary code (for the sake of simplicity, we hardcode the secrets using constants 0x44 and 0x55). Although you do not know the secret values, in practice, it is not so difficult to find out the memory address (the range or the exact value) of them (they are in consecutive addresses), because for many operating systems, the addresses are exactly the same anytime you run the program. In this lab, we just assume that you have already known the exact addresses. To achieve this, the program "intentionally" prints out the addresses for you. With such knowledge, your goal is to achieve the following tasks (not necessarily at the same time):

- Task 1: Crash the program.
- Task 2: Print out the secret[1] value.
- Task 3: Modify the secret[1] value.

Note that the binary code of the program (`Set-UID`) is only readable/executable by you, and there is no way you can modify the code. Namely, you need to achieve the above objectives without modifying the vulnerable code. However, you do have a copy of the source code, which can help you design your attacks.

```
/* vul_prog.c */

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  int int_input;
  int a, b, c, d; /* other variables, not used here.*/
```

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

  printf("Please enter a decimal integer\n");
  scanf("%d", &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;
}
```

**Complie the program:**

```
$ gcc -o vul_prog vul_prog.c
```

**Make it as** `Set-UID` **program:**

```
$ chmod 4755 vul_prog
```

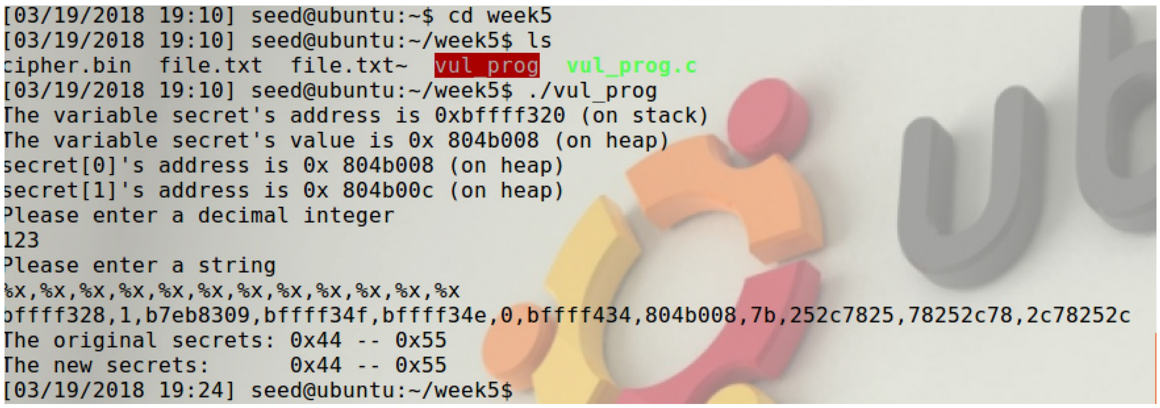
From the printout, you will find out that `secret[0]` and `secret[1]` are located on the heap, i.e., the actual secrets are stored on the heap. We also know that the address of the first secret (i.e., the value of the variable `secret`) can be found on the stack, because the variable `secret` is allocated on the stack. In other words, if you want to overwrite `secret[0]`, its address is already on the stack; your format string can take advantage of this information. However, although `secret[1]` is just right after `secret[0]`, its address is not available on the stack. This poses a major challenge for your format-string exploit, which needs to have the exact address right on the stack in order to read or write to that address.

Below are some format parameters which can be used as your input to complete the tasks:

```
%x Read data from the stack

%s Read character strings from the process' memory
```

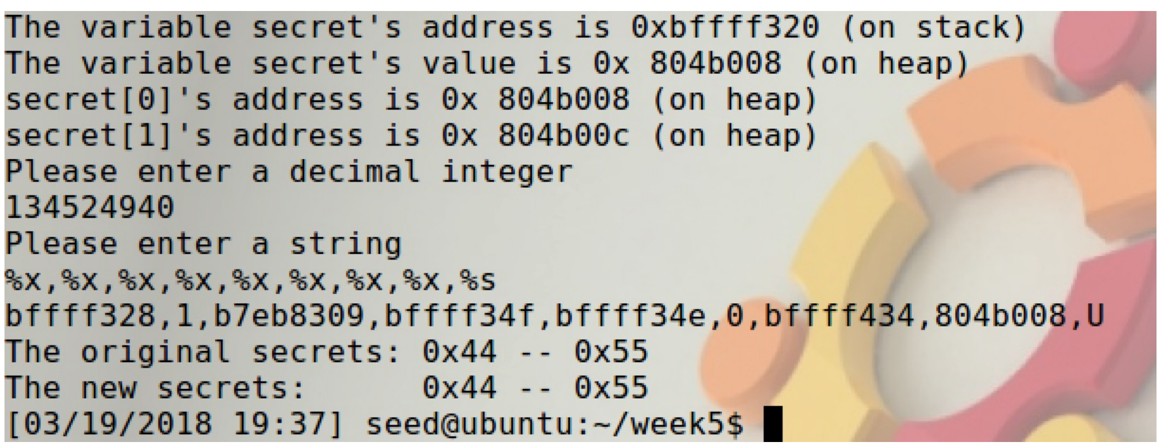
`%n Write an integer to locations in the process' memory`

**Hints for Task 2 and Task 3:** To print out the secret value, we aim to figure out where the input decimal integer is saved. This can be accomplished by inputting a specific number and then using an input string of a relatively large number of "%x" to figure the address of the input number. As seen, 7b (i.e., 123) is in the 9th position.



```
[03/19/2018 19:10] seed@ubuntu:~$ cd week5
[03/19/2018 19:10] seed@ubuntu:~/week5$ ls
cipher.bin  file.txt  file.txt~  vul_prog  vul_prog.c
[03/19/2018 19:10] seed@ubuntu:~/week5$ ./vul_prog
The variable secret's address is 0xbffff320 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
Please enter a decimal integer
123
Please enter a string
%x,%x,%x,%x,%x,%x,%x,%x,%x,%x,%x,%x
bffff328,1,b7eb8309,bffff34f,bffff34e,0,bffff434,804b008,7b,252c7825,78252c78,2c78252c
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[03/19/2018 19:24] seed@ubuntu:~/week5$
```

After the address is obtained, we re-run this program. The address of secret [1] (0x804b00c) is equivalent to 134524940, which is used as input number. Note that 0x804b00c could be different in your machine. Then pass $\%s$ in the 9th position, and the content of secret [1] is displayed as a string "U" (aka 0x55 in ASCII).



```
The variable secret's address is 0xbffff320 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
Please enter a decimal integer
134524940
Please enter a string
%x,%x,%x,%x,%x,%x,%x,%x,%s
bffff328,1,b7eb8309,bffff34f,bffff34e,0,bffff434,804b008,U
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[03/19/2018 19:37] seed@ubuntu:~/week5$
```

To modify the value, we may replace $\%s$ with $\%n$ at the 9th position. The content of secret [1] will be replaced by the number of characters fetched out.

## 2 File Encryption

In this task, we will play with an encryption algorithm via OpenSSL. Create a text file (file.txt) and then do the following:

1. Use the following `openssl enc` command to encrypt/decrypt the file. To see the manuals, you can type `man openssl` and `man enc`.

```
% openssl enc ciphertype -e -in file.txt -out cipher.bin \
               -K  00112233445566778889aabbccddeeff \
               -iv 0102030405060708
```

Please replace the `ciphertype` with `-aes-128-cbc`. You can find the meaning of the command-line options and all the supported cipher types by typing ``` ``man enc'' ``` (check the `supported ciphers` section). We include some common options for the `openssl enc` command in the following:

```
  -in <file>     input file
  -out <file>    output file
  -e             encrypt
  -d             decrypt
  -K/-iv         key/iv in hex is the next argument
  -[pP]          print the iv/key (then exit if -P)
```

2. Use the same encryption key and IV to encrypt the file, and compare two encrypted files to see if they are same. (using `Ghex` on the desktop of SEEDVM to check the content of the encrypted file)

3. Use the same encryption key with a new IV to encrypt the file, and compare the encrypted file with the previous one to see if they are same.

4. Is the size of the encrypted file the same as the size of the original text file?

# 3 Guidelines for Format String Vulnerability

## 3.1 What is a format string?

```
printf ("The magic number is: %d\n", 1911);
```

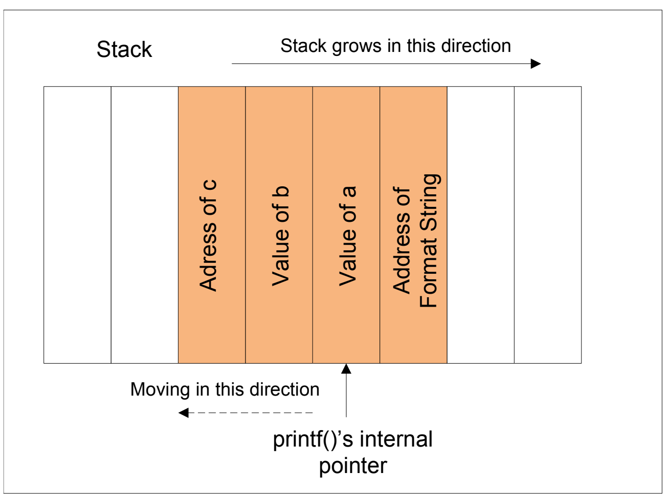
The text to be printed is "The magic number is:", followed by a format parameter '%d', which is replaced with the parameter (1911) in the output. Therefore the output looks like: The magic number is: 1911. In addition to %d, there are several other format parameters, each having different meaning. The following table summarizes these format parameters:

```
Parameter       Meaning                                     Passed as
-------------------------------------------------------------------
   %d        decimal (int)                                   value
   %u        unsigned decimal (unsigned int)                 value
   %x        hexadecimal (unsigned int)                      value
   %s        string ((const) (unsigned) char *)              reference
   %n        number of bytes written so far, (* int)         reference
```

## 3.2 The Stack and Format Strings

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

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                          a, b, &c);
```




## 3.3 What if there is a miss-match

What if there is a miss-match between the format string and the actual arguments?

```
printf ("a has value %d, b has value %d, c is at address: %08x\n",
                          a, b);
```

- In the above example, the format string asks for 3 arguments, but the program actually provides only two (i.e. $a$ and $b$).
- Can this program pass the compiler?
  - The function `printf()` is defined as function with variable length of arguments. Therefore, by looking at the number of arguments, everything looks fine.
  - To find the miss-match, compilers needs to understand how `printf()` works and what the meaning of the format string is. However, compilers usually do not do this kind of analysis.
  - Sometimes, the format string is not a constant string, it is generated during the execution of the program. Therefore, there is no way for the compiler to find the miss-match in this case.
- Can `printf()` detect the miss-match?
  - The function `printf()` fetches the arguments from the stack. If the format string needs 3 arguments, it will fetch 3 data items from the stack. Unless the stack is marked with a boundary, `printf()` does not know that it runs out of the arguments that are provided to it.
  - Since there is no such a marking. `printf()` will continue fetching data from the stack. In a miss-match case, it will fetch some data that do not belong to this function call.
- What trouble can be caused by `printf()` when it starts to fetch data that is meant for it?

## 3.4 Viewing Memory at Any Location

- We have to supply an address of the memory. However, we cannot change the code; we can only supply the format string.
- If we use `printf(%s)` without specifying a memory address, the target address will be obtained from the stack anyway by the `printf()` function. The function maintains an initial stack pointer, so it knows the location of the parameters in the stack.
- Observation: the format string is usually located on the stack. If we can encode the target address in the format string, the target address will be in the stack. In the following example, the format string is stored in a buffer, which is located on the stack.

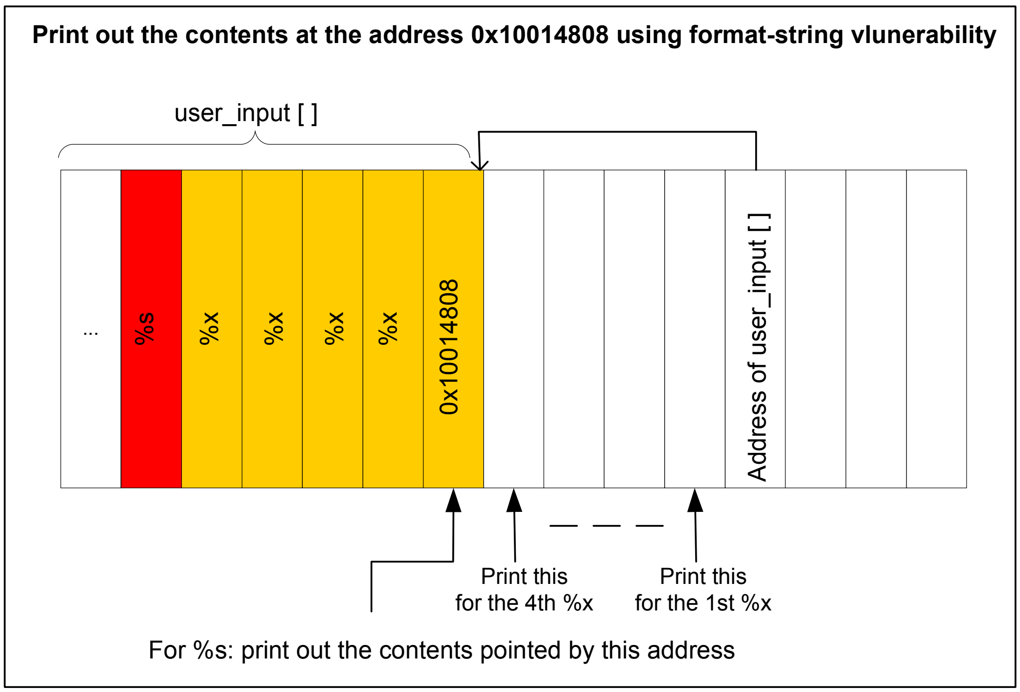
```
int main(int argc, char *argv[])
{
  char user_input[100];
  ... ... /* other variable definitions and statements */

  scanf("%s", user_input); /* getting a string from user */
  printf(user_input); /* Vulnerable place */

  return 0;
}
```

- If we can force the printf to obtain the address from the format string (also on the stack), we can control the address.

```
printf ("\x10\x01\x48\x08  %x %x %x %x %s");
```

- `\x10\x01\x48\x08` are the four bytes of the target address. In C language, `\x10` in a string tells the compiler to put a hexadecimal value `0x10` in the current position. The value will take up just one byte. Without using `\x`, if we directly put `"10"` in a string, the ASCII values of the characters `'1'` and `'0'` will be stored. Their ASCII values are 49 and 48, respectively.
- `%x` causes the stack pointer to move towards the format string.
- Here is how the attack works if `user_input[]` contains the following format string:

```
"\x10\x01\x48\x08  %x %x %x %x %s".
```




- Basically, we use four `%x` to move the `printf()`'s pointer towards the address that we stored in the format string. Once we reach the destination, we will give `%s` to `print()`, causing it to print out the contents in the memory address `0x10014808`. The function `printf()` will treat the contents as a string, and print out the string until reaching the end of the string (i.e. 0).
- The stack space between `user_input[]` and the address passed to the `printf()` function is not for `printf()`. However, because of the format-string vulnerability in the program, `printf()` considers them as the arguments to match with the `%x` in the format string.
- The key challenge in this attack is to figure out the distance between the `user_input[]` and the address passed to the `printf()` function. This distance decides how many `%x` you need to insert into the format string, before giving `%s`.

## 3.5 Writing an Integer to Memory

- `%n`: The number of characters written so far is stored into the integer indicated by the corresponding argument.

```
        int i;
        printf ("12345%n", &i);
```

- It causes `printf()` to write 5 into variable $i$.
- Using the same approach as that for viewing memory at any location, we can cause `printf()` to write an integer into any location. Just replace the `%s` in the above example with `%n`, and the contents at the address `0x10014808` will be overwritten.
- Using this attack, attackers can do the following:
  - Overwrite important program flags that control access privileges
  - Overwrite return addresses on the stack, function pointers, etc.
- However, the value written is determined by the number of characters printed before the `%n` is reached. Is it really possible to write arbitrary integer values?
  - Use dummy output characters. To write a value of 1000, a simple padding of 1000 dummy characters would do.
  - To avoid long format strings, we can use a width specification of the format indicators.