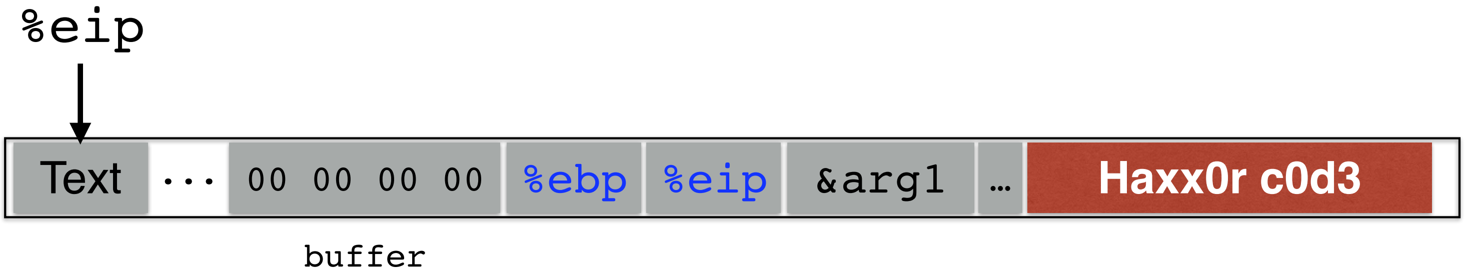
Code injection

Code Injection: Main idea

```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



%eip

Text	...	00 00 00 00	%ebp	%eip	&arg1	…	Haxx0r c0d3

buffer

(1) Load my own code into memory

(2) Somehow get `%eip` to point to it

Challenge 1
Loading code into memory

- It **must be the machine code** instructions (i.e., already compiled and ready to run)
- We have to be careful in how we construct it:
  - It **can't contain** any **all-zero bytes**
    - Otherwise, sprintf / gets / scanf / … will stop copying
    - How could you write assembly to never contain a full zero byte?
  - It **can't use the loader** (we're injecting)
  - It **can't use the stack** (we're going to smash it)

What code to run?

- Goal: **general-purpose shell**
  - Command-line prompt that gives attacker **general access to the system**
- The code to launch a shell is called **shellcode**

Shellcode

```
#include <stdio.h>
int main( ) {
   char *name[2];
   name[0] = “/bin/sh”;
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

```
“\x31\xc0”
“\x50”
“\x68””//sh”
“\x68””/bin”
“\x89\xe3”
“\x50”
...
```

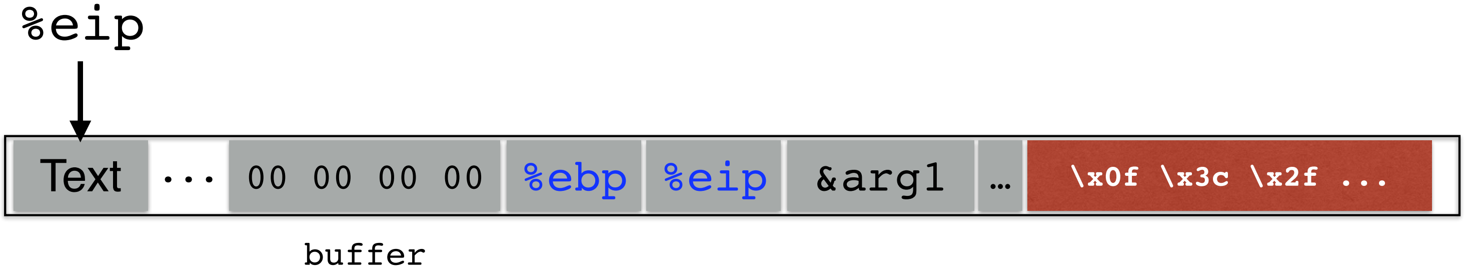
Machine code

(Part of) your input

Challenge 2

Getting injected code to run

- We can’t insert a “jump into my code” instruction
- We don’t know precisely where our code is



Recall

Memory layout summary

Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump to the function's address**

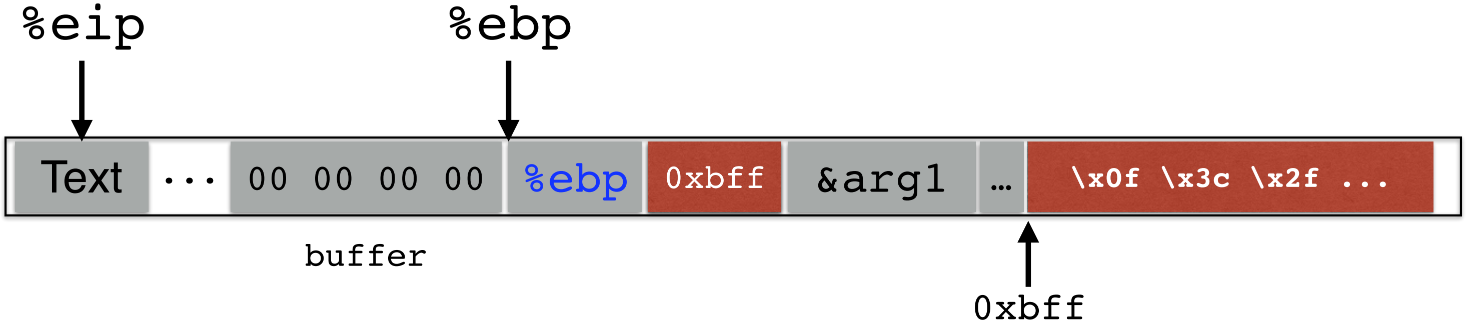
Called function:

4. **Push the old frame pointer** onto the stack (%ebp)
5. **Set frame pointer** (%ebp) to where the end of the stack is right now (%esp)
6. **Push local variables** onto the stack

Returning function:

7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)

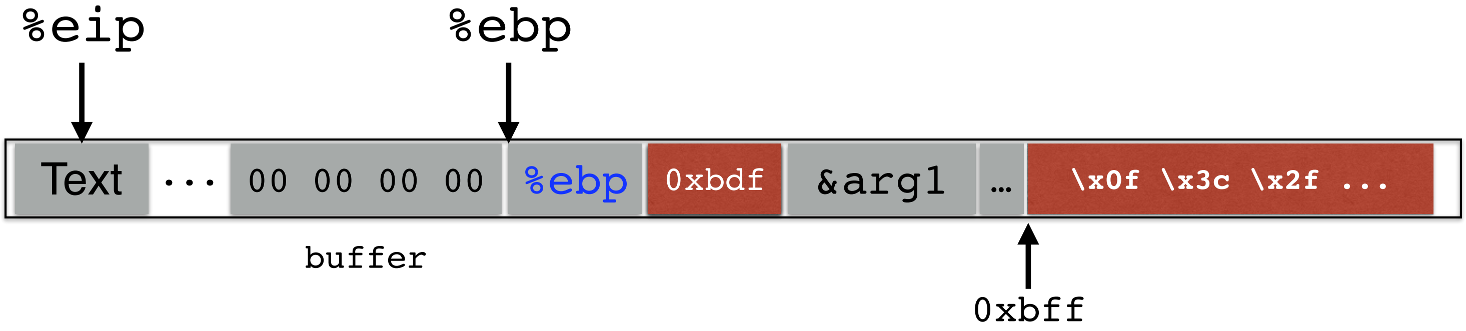
Hijacking the saved `%eip`



But how do we know the address?

Hijacking the saved `%eip`

What if we are wrong?



**This is most likely data,
so the CPU will panic
(Invalid Instruction)**

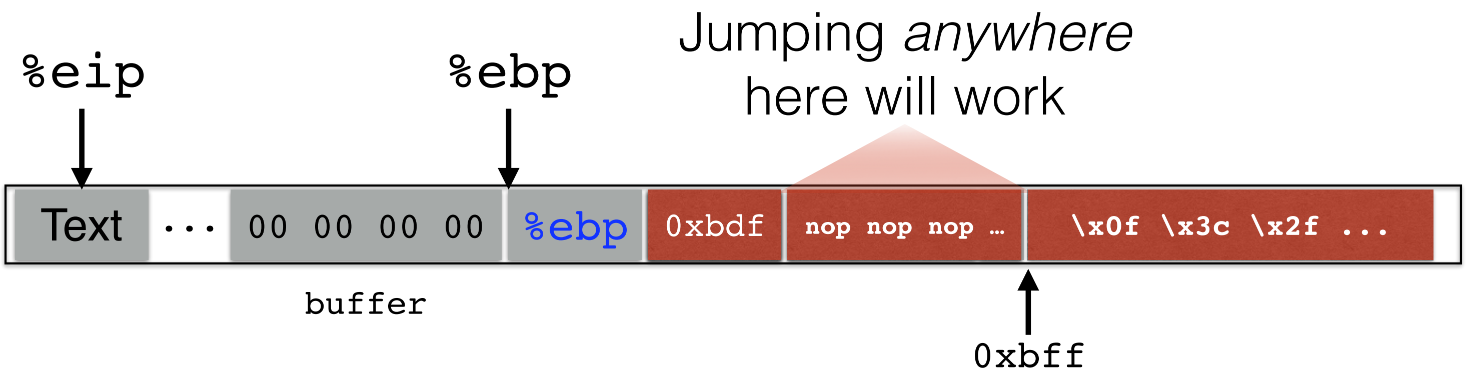
Challenge 3

Finding the return address

- If we don't have access to the code, we don't know how far the buffer is from the saved `%ebp`
- One approach: just try a lot of different values!
  - Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers
- Without address randomization (discussed later):
  - The **stack always starts** from the same **fixed address**
  - The stack will grow, but usually it **doesn't grow very deeply** (unless the code is heavily recursive)

Improving our chances: nop sleds

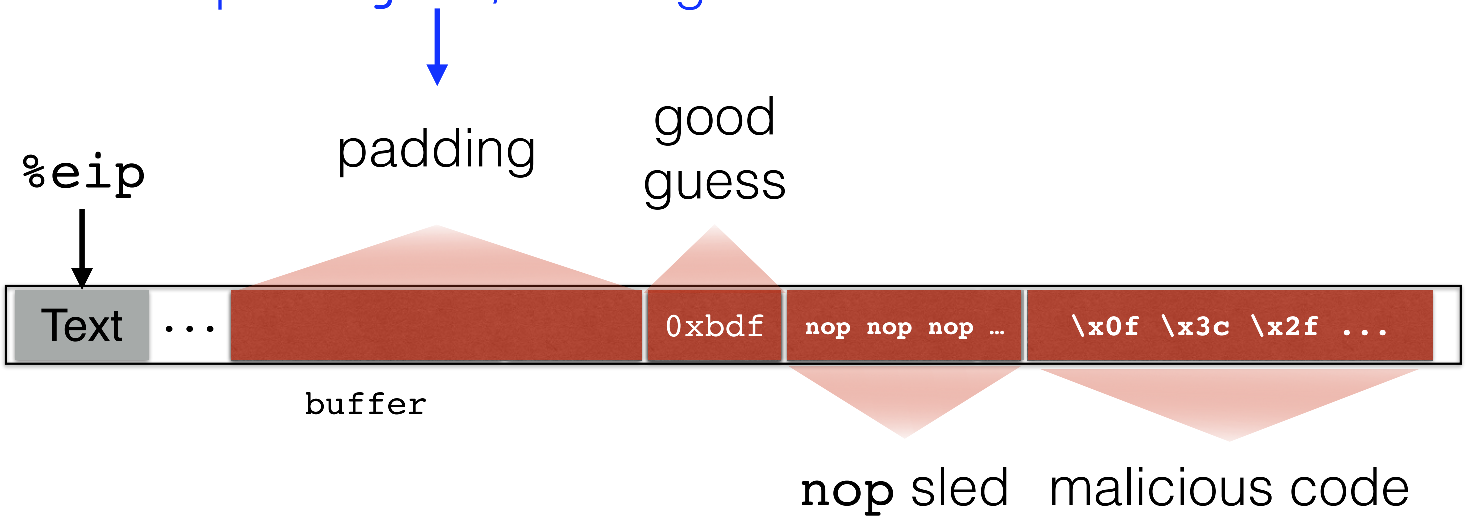
nop is a single-byte instruction
(just moves to the next instruction)



Now we improve our chances of guessing by a factor of #nops

Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to **gets**/etc. begins.



%eip
padding
good guess
Text
...
0xbdf
nop nop nop …
\x0f \x3c \x2f ...
buffer
nop sled
malicious code