Computer and Network Security

Lecture 10:
Buffer Overflows & Binary Exploitation

COMP-5370/6370
Fall2024

Time for a Change

- Moving from crypto-block to host-block
  - Crypto-block: Very abstract and conceptual
  - Host-block: Much more "nuts and bolts"
- Still applying the same Security Mindset
- There is a **LOT** of foundational knowledge required to fully understand
  - Operating systems, data structures, concurrency, compilers, etc.

Buffer Overflows

Buffer overflows are class of memory corruption bugs where a program attempts to put **too-much data** into **too-small** of a memory allocation.

```
void print_name(char** argv) {
    char buf[10];
    strcpy(buf, argv[0]);
    printf("Running: %s", buf);
}
```

As the Morris Worm Turned

By Meghan Holohan

Twenty years later, cybersecurity remains a challenging set of problems

At half past midnight on November 3, 1988, subscribers to an email list for TCP-IP developers received an ominous message: "There may be a virus loose on the Internet," it warned.

A worm written by a Cornell grad student, Robert Tappan Morris, had been released from Massachusetts Institute of Technology. Designed to test the size of the Internet, it got access to computers by exploiting a variety of vulnerabilities, including easy-to-guess passwords and weaknesses in the sendmail and finger server processes that ran on most of the hosts.

As the Morris Worm Turned

By Meghan

Twenty

challen

At half p

an emai

message

warned.

A worm

Massach

to comp

weakne

Security Bulletin

Microsoft Security Bulletin MS02-039 - Critical

Buffer Overruns in SQL Server 2000 Resolution Service Could Enable Code Execution (Q323875)

Published: July 24, 2002 | Updated: January 31, 2003

Version: 1.2

Originally posted: July 24, 2002

Updated: January 31, 2003

As the Morris Worm Turned

By Meghar

Twenty
challen

At half p
an emai
message
warned.

A worm
Massach
to comp
weaknes

Security Bulletin

Microsoft Security Bulletin MS02-03? - Critical

Buffer Ove
Resolutior
Execution

Published: July 24, 20(

Version: 1.2

Originally posted: July

Updated: January 31,

D-Link routers contain buffer overflow vulnerability

Vulnerability Note VU#332115

Original Release Date: 2016-08-11 | Last Revised: 2016-08-12

Overview

D-Link DIR routers contain a stack-based buffer overflow vulnerability, which may allow a remote attack to execute arbitrary code.

Description

CWE-121: Stack-based Buffer Overflow - CVE-2016-5681

A stack-based buffer overflow occurs in the function within the cgibin binary which validates the session cookie.
This function is used by a service which is exposed to the WAN network on port 8181 by default.

As the Morris Worm Turned

By Meghar

Twenty
challen

At half p
an emai
message
warned.

A worm
Massach
to comp
weaknes

Security Bulletin

Microsoft Security Bulletin MS02-03

Buffer Ove
Resolutior
Execution

Published: July 24, 200

Version: 1.2

Originally posted: July

Updated: January 31,

D-Link routers contain buffer overflow vulnerability

Vulnerability Note VU#332115

Original Release Date: 2016-08-11 | Last Revised: 2016-08-12

Overview

D-Link DIR routers contain a stack-l
execute arbitrary code.

Description

CWE-121: Stack-based Buffer Ove

A stack-based buffer overflow occu
cookie.
This function is used by a service w

Pulse Connect Secure Samba buffer overflow

Vulnerability Note VU#667933

Original Release Date: 2021-05-24 | Last Revised: 2021-06-17

Overview

Pulse Connect Secure (PCS) gateway contains a buffer overflow vulnerability in Samba-related code that may allow an authenticated remote attacker to execute arbitrary code.

Description

CVE-2021-22908

PCS includes the ability to connect to Windows file shares (SMB). This capability is provided by a number of CGI scripts, which in turn use libraries and helper applications based on Samba 4.5.10. When specifying a long server name for some SMB operations, the `smbclt` application may crash due to either a stack buffer overflow or a heap buffer overflow, depending on how long of a server name is specified. We have confirmed that PCS 9.1R11.4 systems are vulnerable, targeting a CGI endpoint of: `/dana/fb/smb/wnf.cgi`. Other CGI endpoints may also trigger the vulnerable code.

Buffer Overflows

Buffer overflows are class of memory corruption bugs where a program attempts to put **too-much data** into **too-small** of a memory allocation.

```
void print_name(char** argv) {
    char buf[10];
    strcpy(buf, argv[0]);
    printf("Running: %s", buf);
}
```

Our World

PRETEND THE WORLD IS SIMPLE.

Our World

PRETEND THE WORLD IS SIMPLE.

Our World

- x86 ISA with Intel syntax
  - AT&T is similar but weirder

Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2A: Instruction Set Reference, A-L

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-L*, Order Number 253666; *Instruction Set Reference M-U*, Order Number 253667; *Instruction Set Reference V-Z*, Order Number 326018; *Instruction Set Reference*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers*, Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Our World

- x86 ISA with Intel syntax
  - AT&T is similar but weirder
- 32-bit CPU
- Concepts apply in non-32-bit CPUs
  - AH/AL: 8-bit
  - AX: 16-bit
  - EAX: 32-bit
  - RAX: 64-bit

CPU - Registers

- General Purpose Registers
  - EAX, EBX, ECX, EDX

CPU - Registers

- General Purpose Registers
  - EAX, EBX, ECX, EDX
- Semi-General Purpose Registers
  - EDI, ESI

CPU - Registers

- General Purpose Registers
  - EAX, EBX, ECX, EDX
- Semi-General Purpose Registers
  - EDI, ESI
- Special Purpose Registers:
  - EIP, ESP, EBP

CPU - Instructions

- Move a value to a register
  - **`mov eax, 0x34`**

CPU - Instructions

- Move a value to a register
  - `mov eax, 0x34`
- Add a value to a register
  - `add eax, 10`

CPU - Instructions

- Move a value to a register
  - `mov eax, 0x34`
- Add a value to a register
  - `add eax, 10`
- Load calculated address into register
  - `lea eax, [ebx+4]`

CPU - Instructions

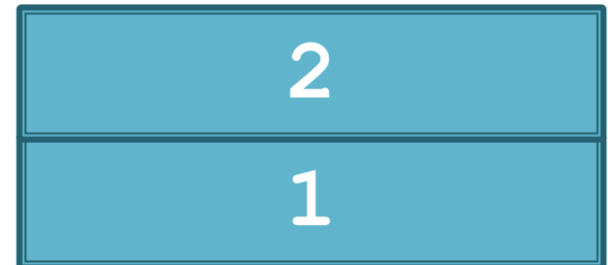
- Change execution path
  - `jmp 0x12345678    # can't return`
    - `je, jne, jg, jge, ...`
  - `call 0x12345678   # can return`
  - `ret # return from call instr.`

Stack in Normal-World

```
Stack myStack = Stack();
myStack.push(1);
```

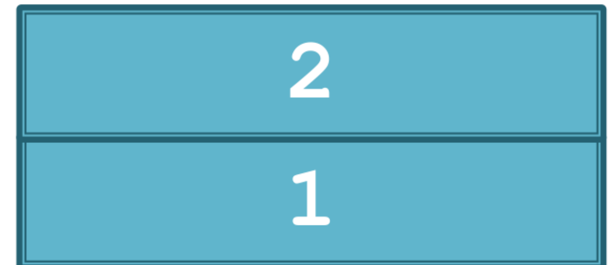
Stack in Normal-World

```
Stack myStack = Stack();
myStack.push(1);
myStack.push(2);
```



Stack in Normal-World

```
Stack myStack = Stack();
myStack.push(1);
myStack.push(2);
myStack.pop();      // returns 2
```



Stack in Assembly-World

- Usually elements are CPU-sized values
  - Our World: 4-bytes each

Stack in Assembly-World

- Usually elements are CPU-sized values
  - Our World: 4-bytes each

- Add elements: **`push`** instruction

`push 0x00000000` adds 4 null-bytes to top

`push eax` adds EAX register value to top

Stack in Assembly-World

- Usually elements are CPU-sized values
  - Our World: 4-bytes each

- Add elements: **`push`** instruction

`push 0x00000000` adds 4 null-bytes to top

`push eax` adds EAX register value to top

- Remove elements: **`pop XXX`** instructions

`pop eax` stores removed value in EAX register and de-allocates 4-bytes

Assembly-Level Stack

```
push 0x0a
```

0a

Assembly-Level Stack

```
push 0x0000000a
```

`0000000a`

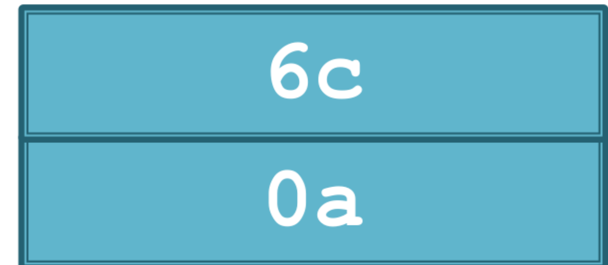
Assembly-Level Stack

```
push 0x0a
```

0a

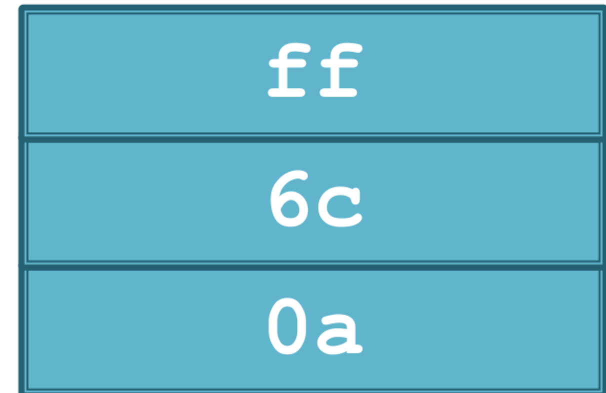
Assembly-Level Stack

```
push 0x0a
push 0x6c
```



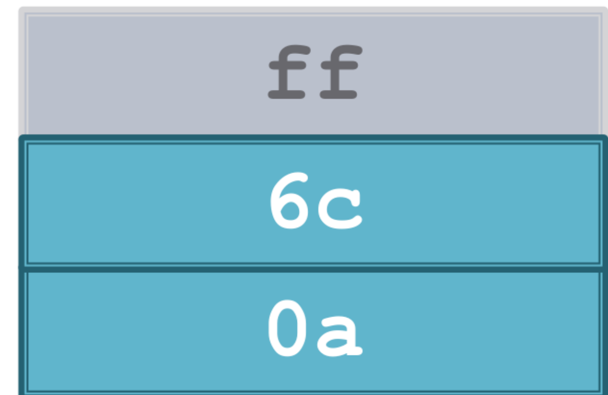
Assembly-Level Stack

```
push 0x0a
push 0x6c
push 0xff
```



Assembly-Level Stack

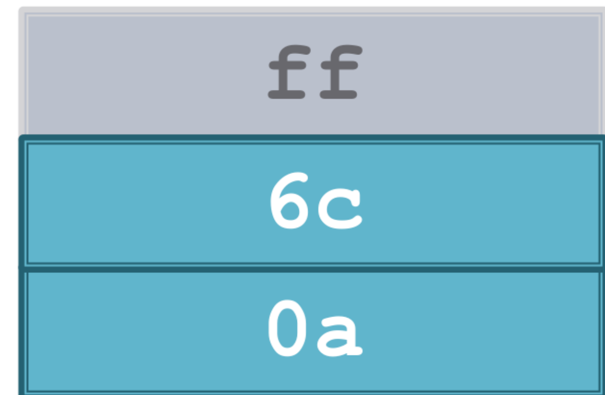
```
push 0x0a
push 0x6c
push 0xff
pop  eax    #0xff
```



Assembly-Level Stack

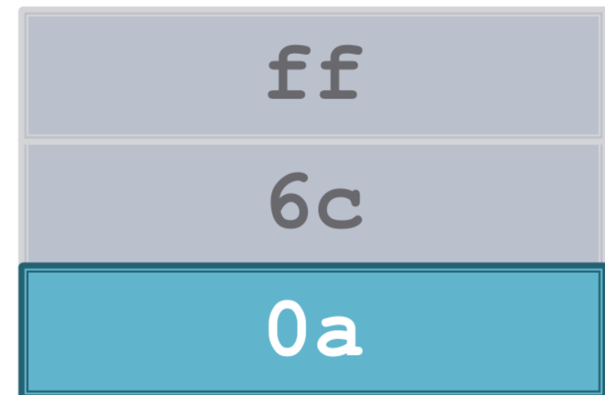
```
push 0x0a
push 0x6c
push 0xff
pop  eax     #0xff
```

Overwrites whatever exists in EAX.



Assembly-Level Stack

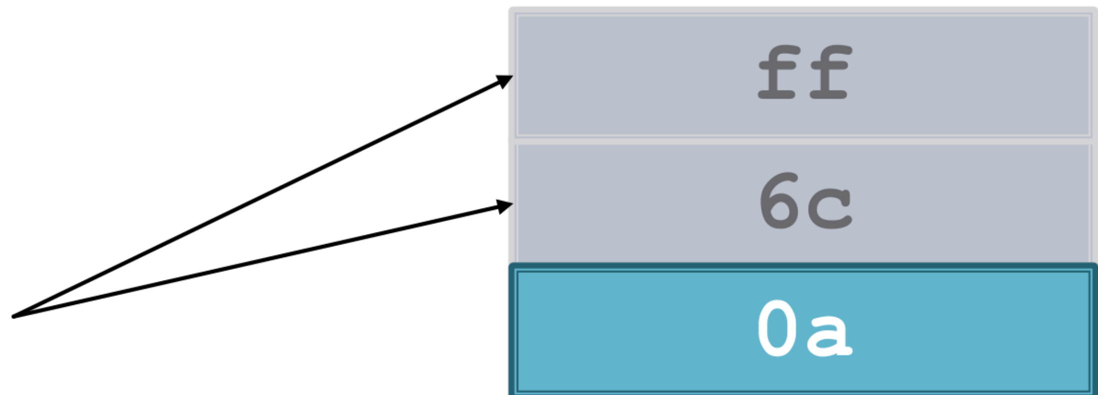
```
push 0x0a
push 0x6c
push 0xff
pop  eax     #0xff
pop  eax     #0x6c
```



Assembly-Level Stack

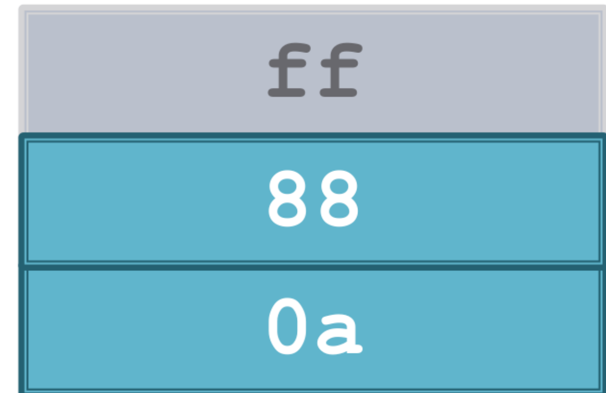
```
push 0x0a
push 0x6c
push 0xff
pop  eax    #0xff
pop  eax    #0x6c
```

Values still exist in memory



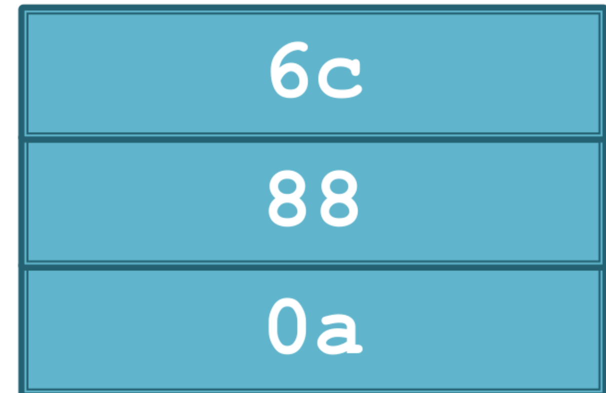
Assembly-Level Stack

```
push 0x0a
push 0x6c
push 0xff
pop  eax     #0xff
pop  eax     #0x6c
push 0x88
```



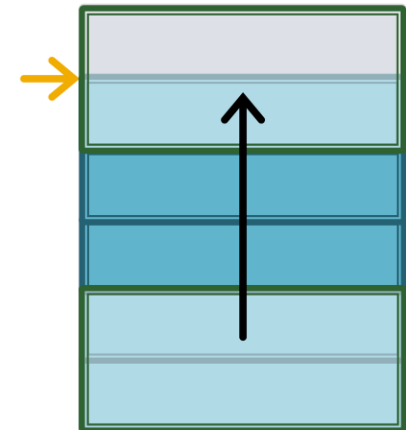
Assembly-Level Stack

```
push 0x0a
push 0x6c
push 0xff
pop  eax     #0xff
pop  eax     #0x6c
push 0x88
push eax     #0x6c
```



Call Stack

- Starts at `0xffffffff`
- Grows toward `0x00000000`
- ESP (→) points to top-of-stack
  - "Stack Pointer"



Assembly-Level Stack

```
push 0xaa
```

Assembly-Level Stack

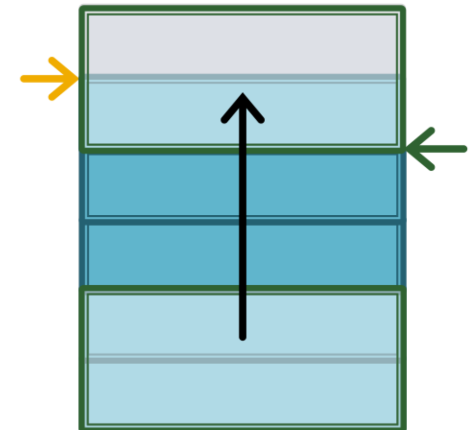
```
push 0xaa
push 0xffffffff
```

Assembly-Level Stack

```
push 0xaa
push 0xffffffff
pop  eax
```

Call Stack

- Starts at `0xffffffff`
- Grows toward `0x00000000`
- ESP (→) points to top-of-stack
  - "Stack Pointer"
- EBP (←) points to bottom of current frame
  - "Base Pointer" / "Frame Pointer"

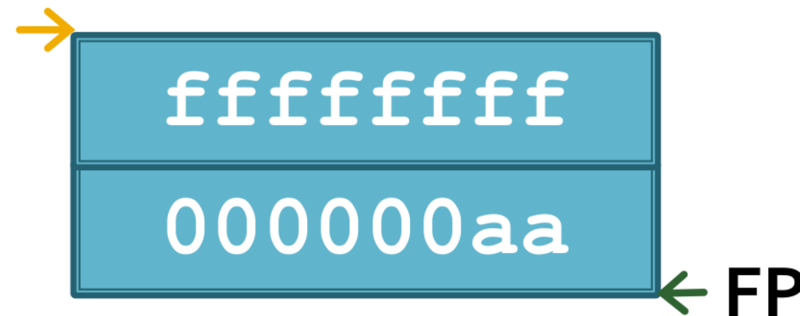


Assembly-Level Stack

```
push 0xaa
```

Assembly-Level Stack

```
push 0xaa
push 0xffffffff
```



Assembly-Level Stack

```
push 0xaa
push 0xffffffff
pop  eax
```

Stack Frames

A **stack frame** is a logical area of the call stack which is associated with a single function's execution instance.

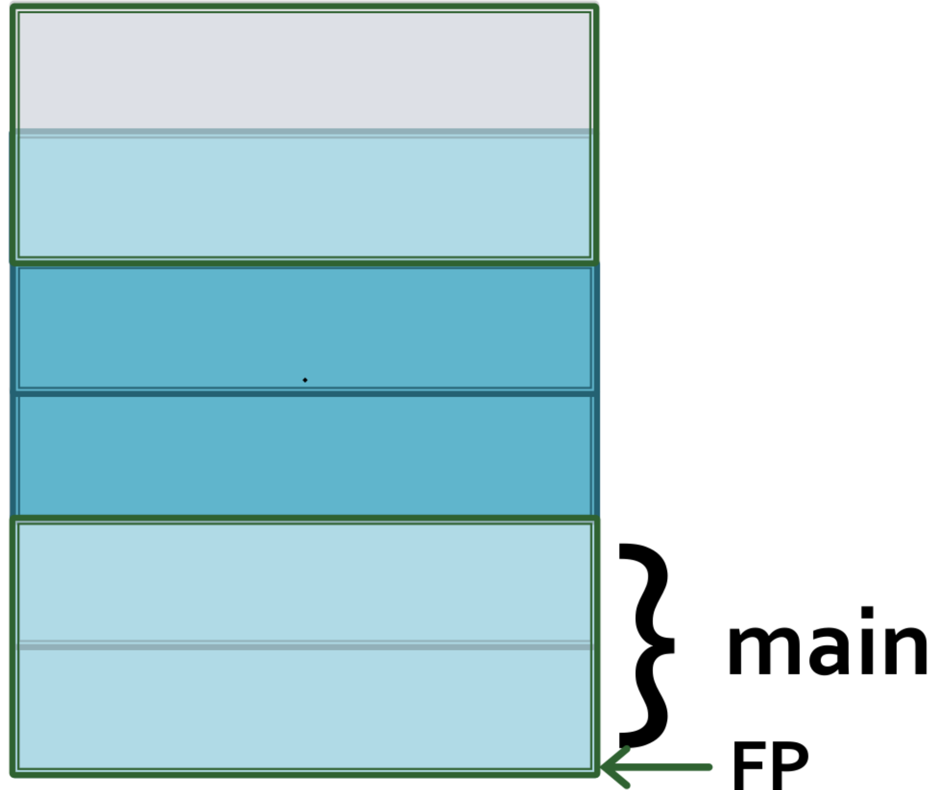
- Local variable are self-contained
- Function arguments and return address included (in stdcall)
- Marked by EBP and ESP

example1.c

```
void func_1() {
      ...
      ...
}

int main() {
      func_1();
      ...
}
```

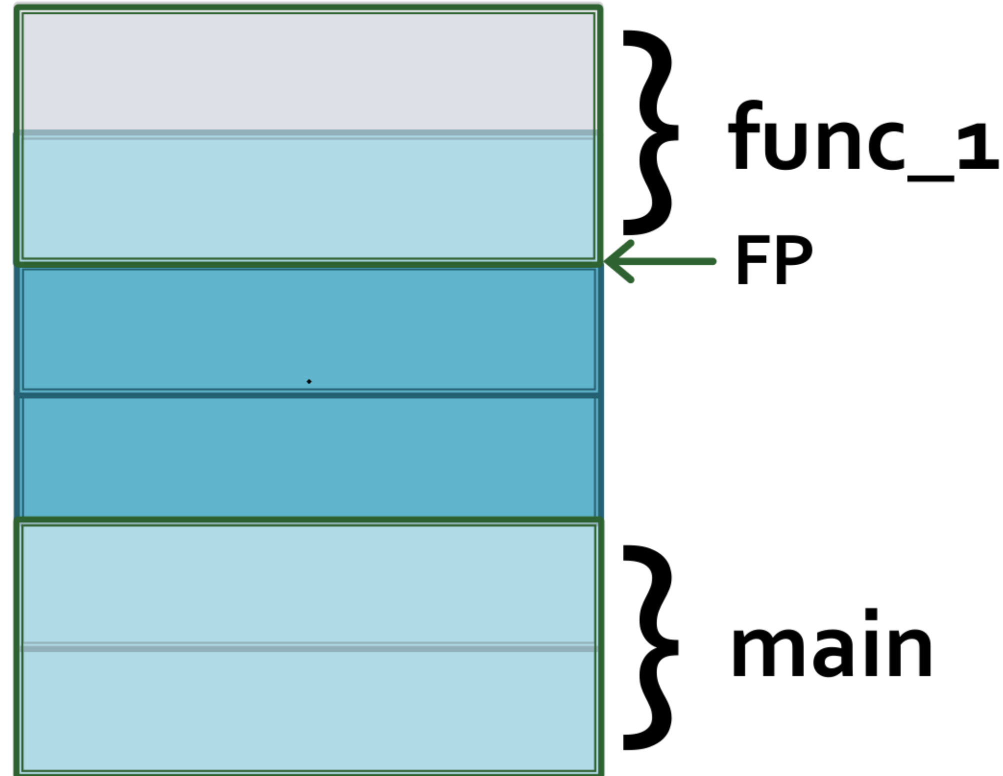
Stack frames

```
void func_1() {
      ...
      ...
}

int main() {
      func_1();
      ...
}
```



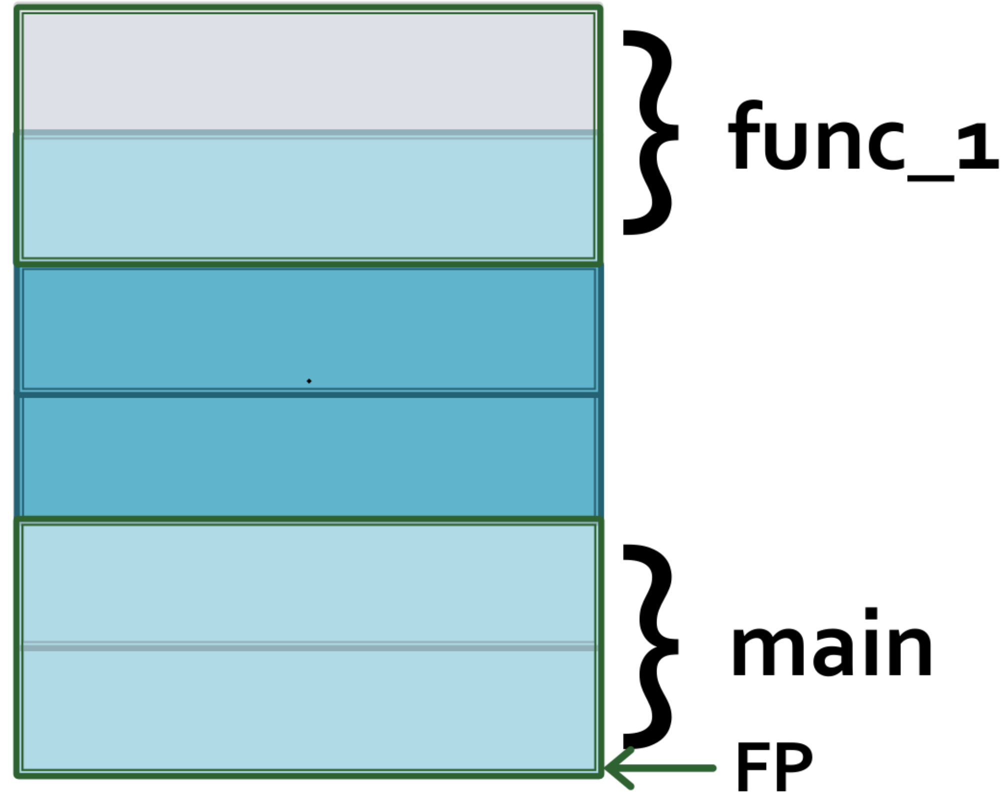
Stack frames

```
void func_1() {
    ...
    ...
}

int main() {
    func_1();
    ...
}
```



Stack frames

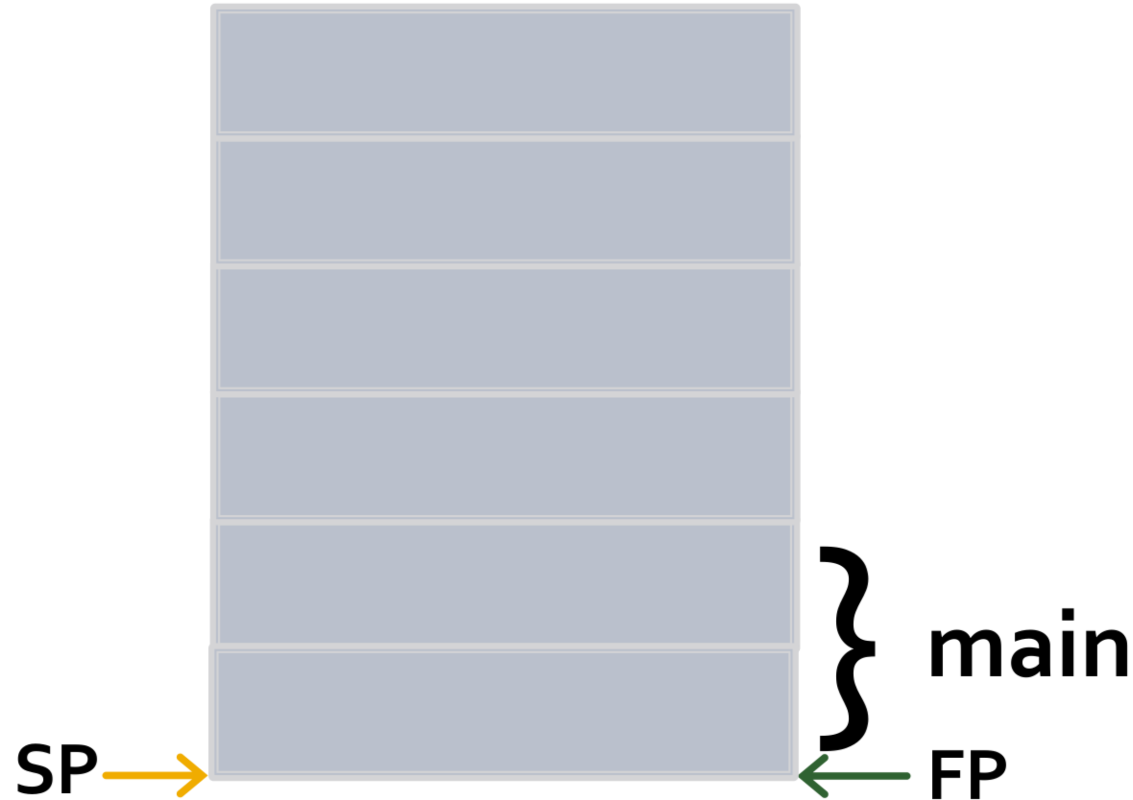
```
void func_1() {
     ...
     ...
}

int main() {
     func_1();
     ...
}
```



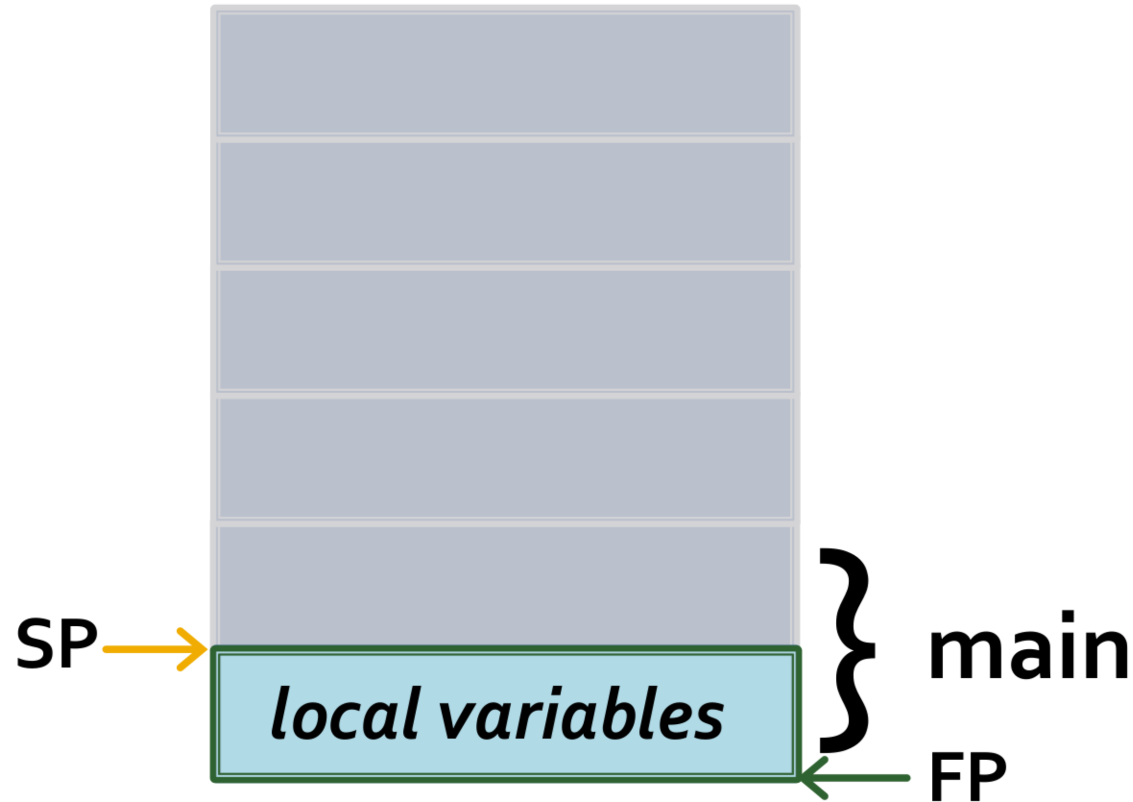
example.c

```
void func_2(int a) {
    ...
    ...
}

int main() {
    func_2(3);
    ...
}
```

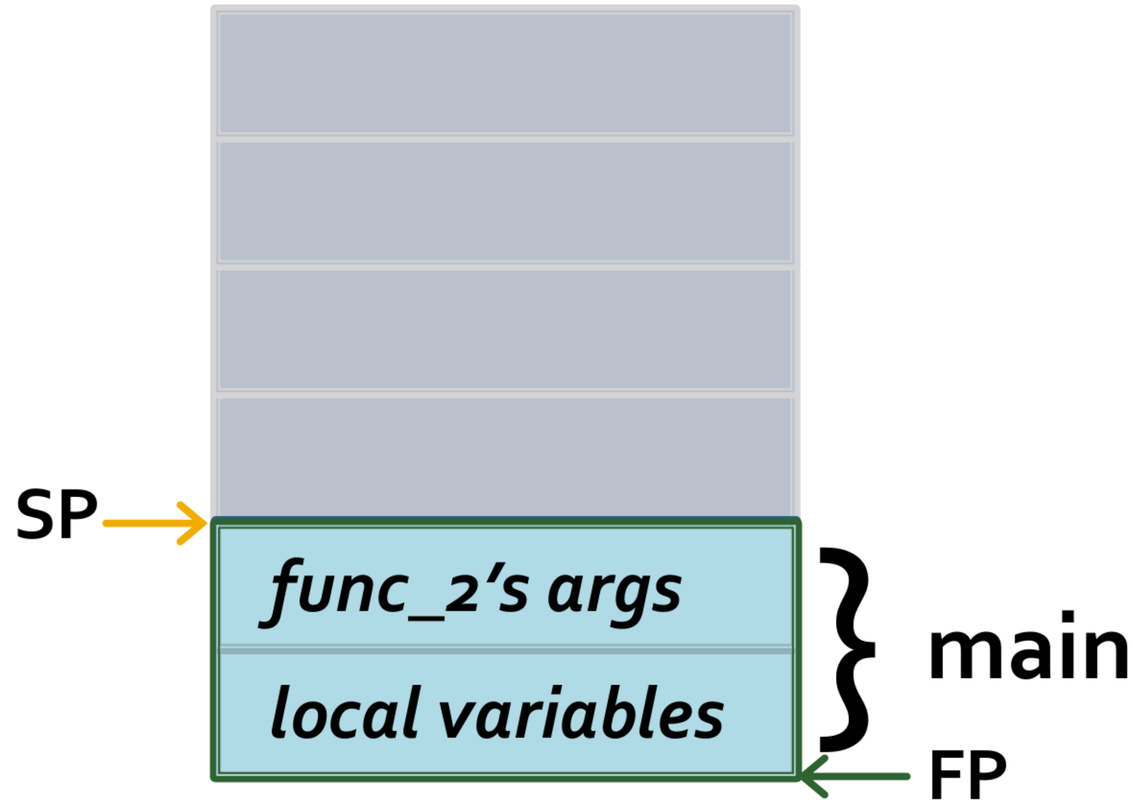
Standard Call Sequence



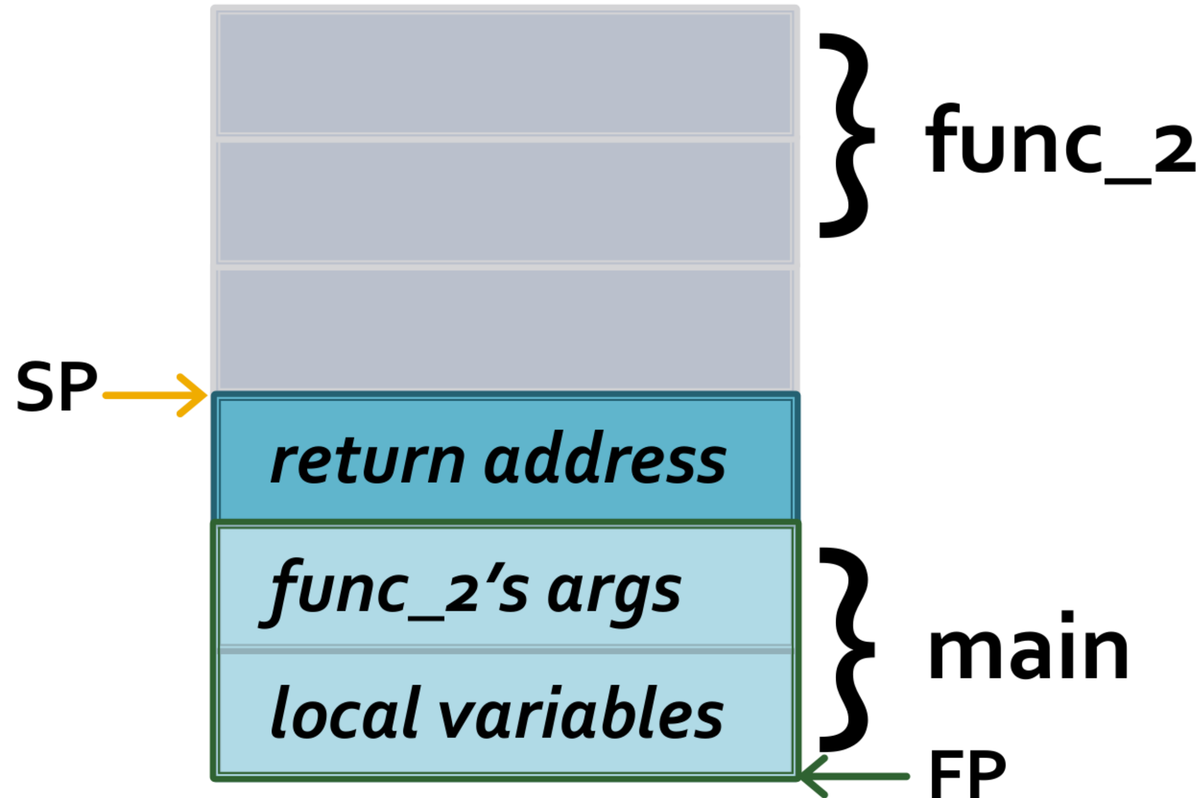
Standard Call Sequence



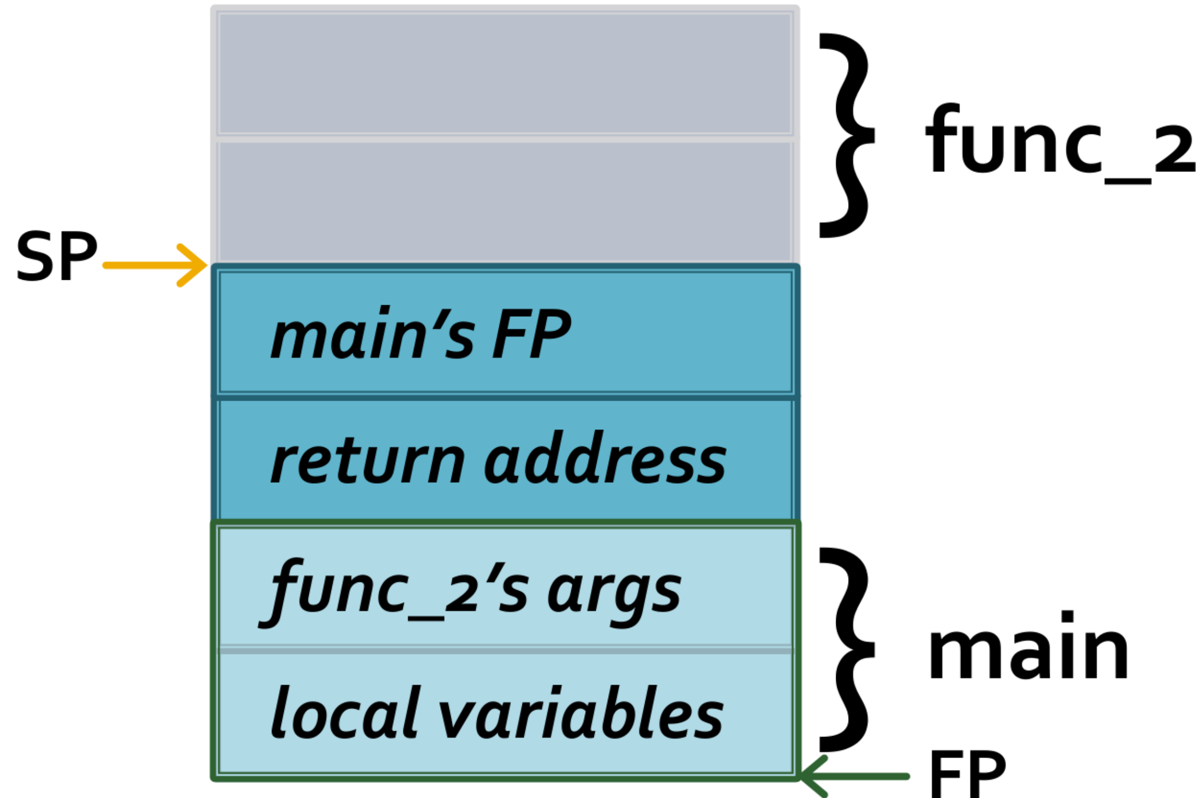
Standard Call Sequence



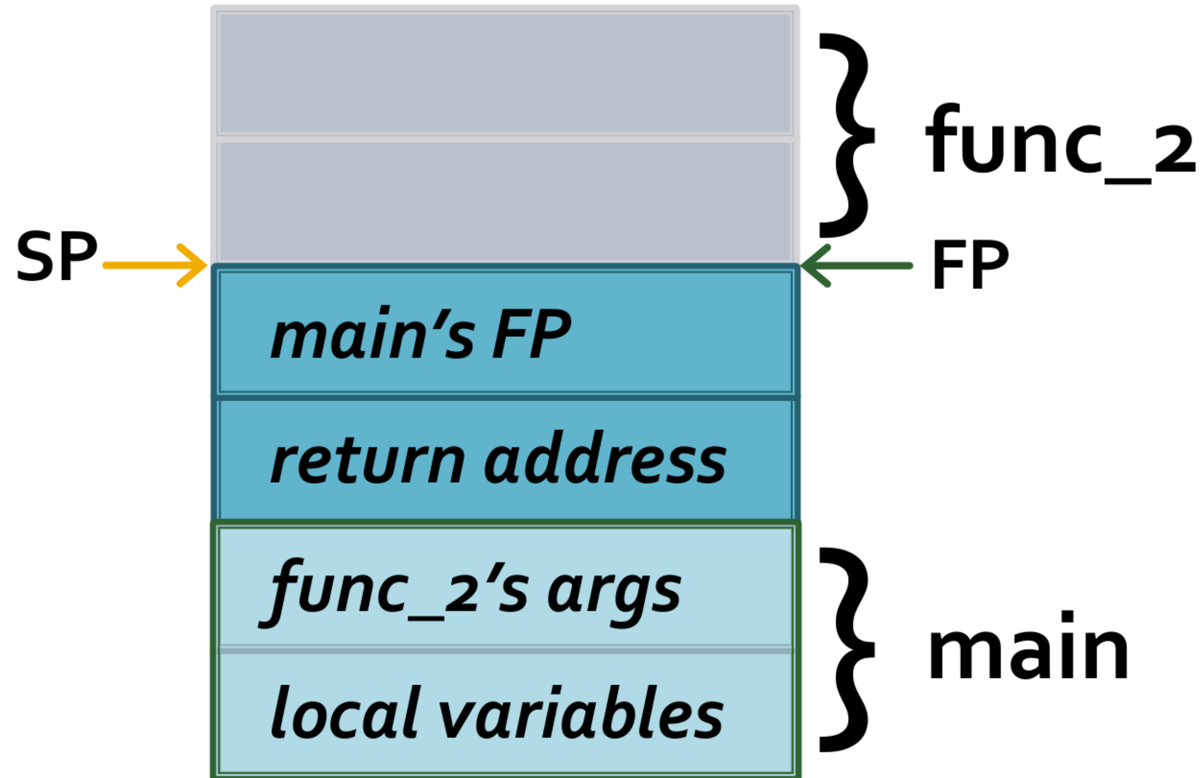
Standard Call Sequence



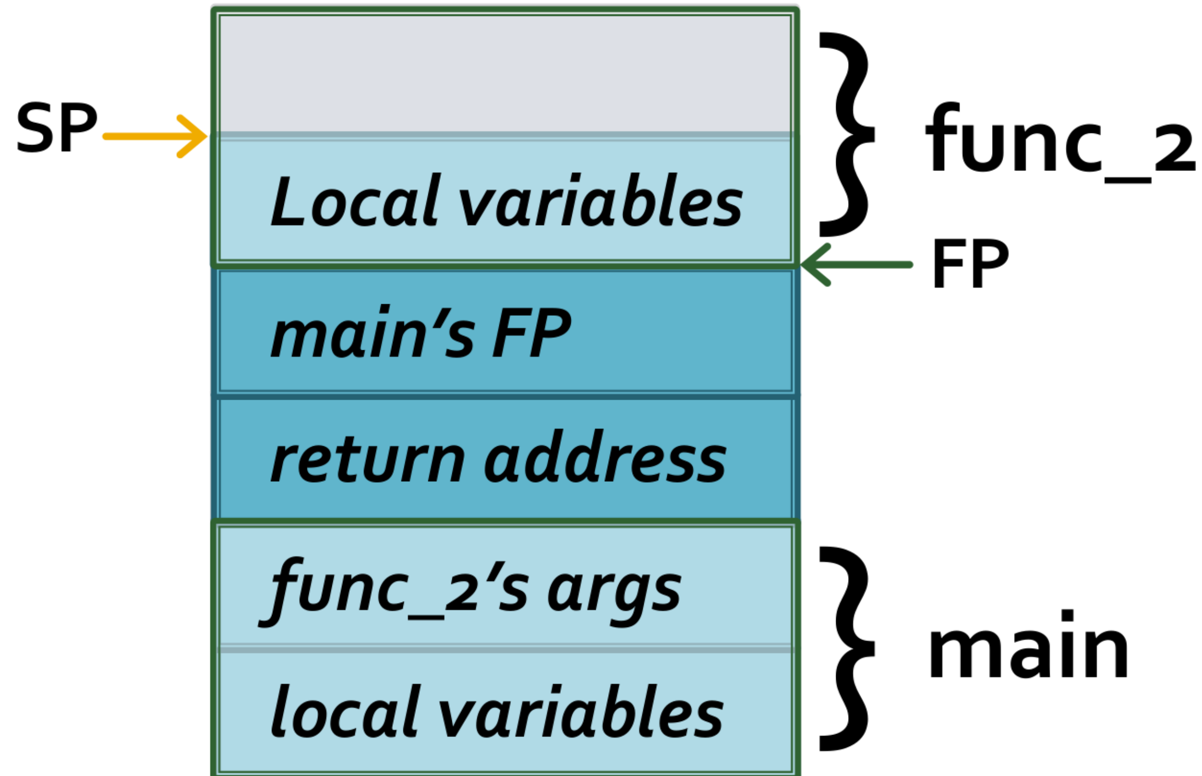
Standard Call Sequence



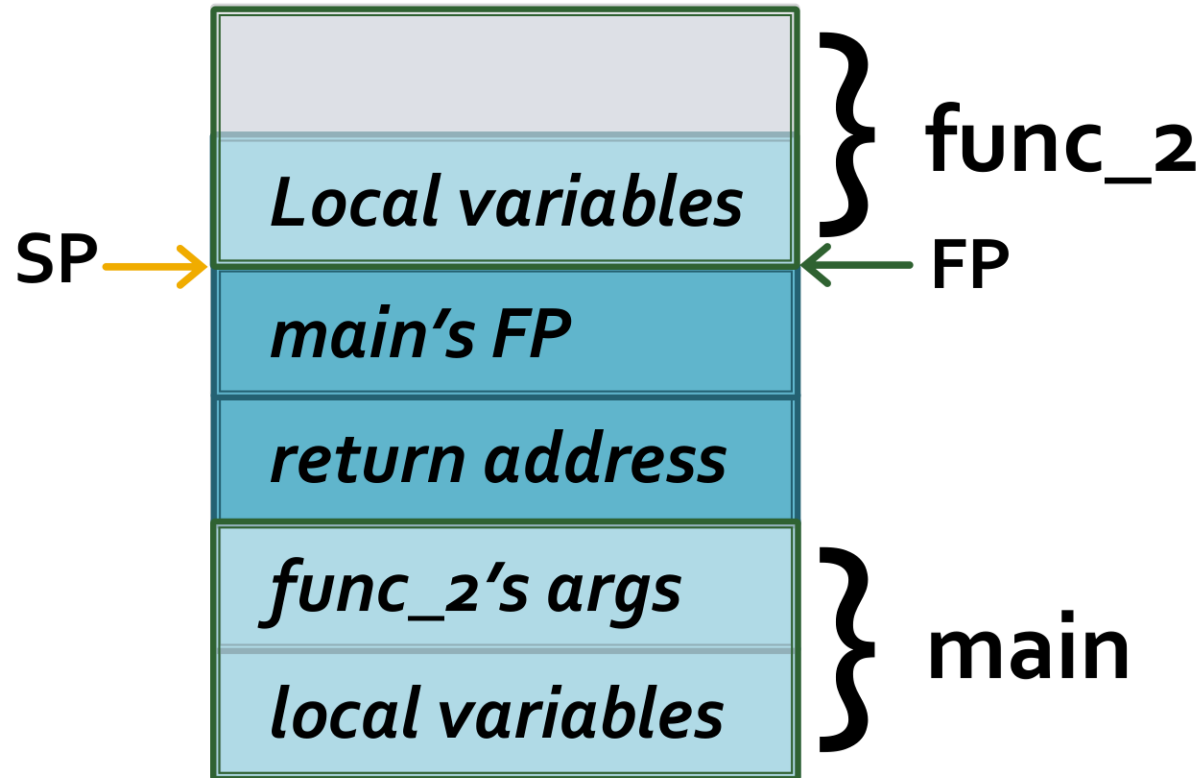
Standard Call Sequence



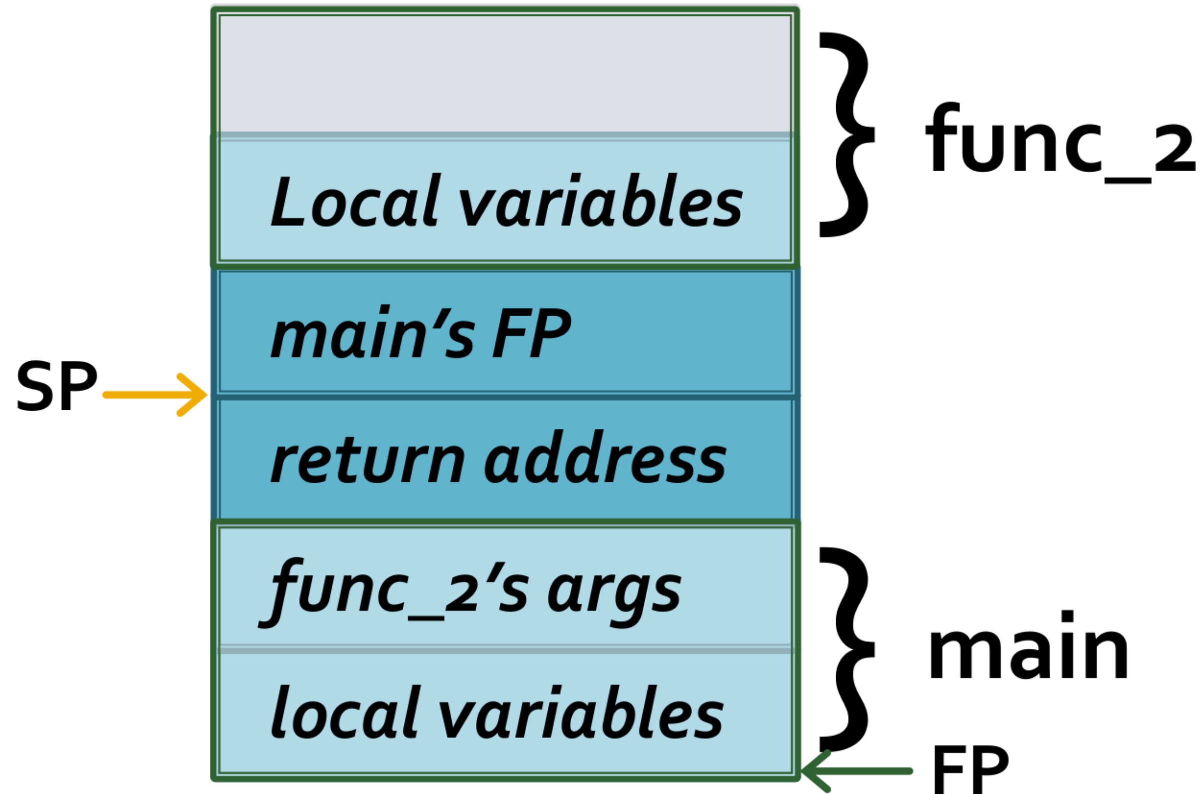
Standard Call Sequence



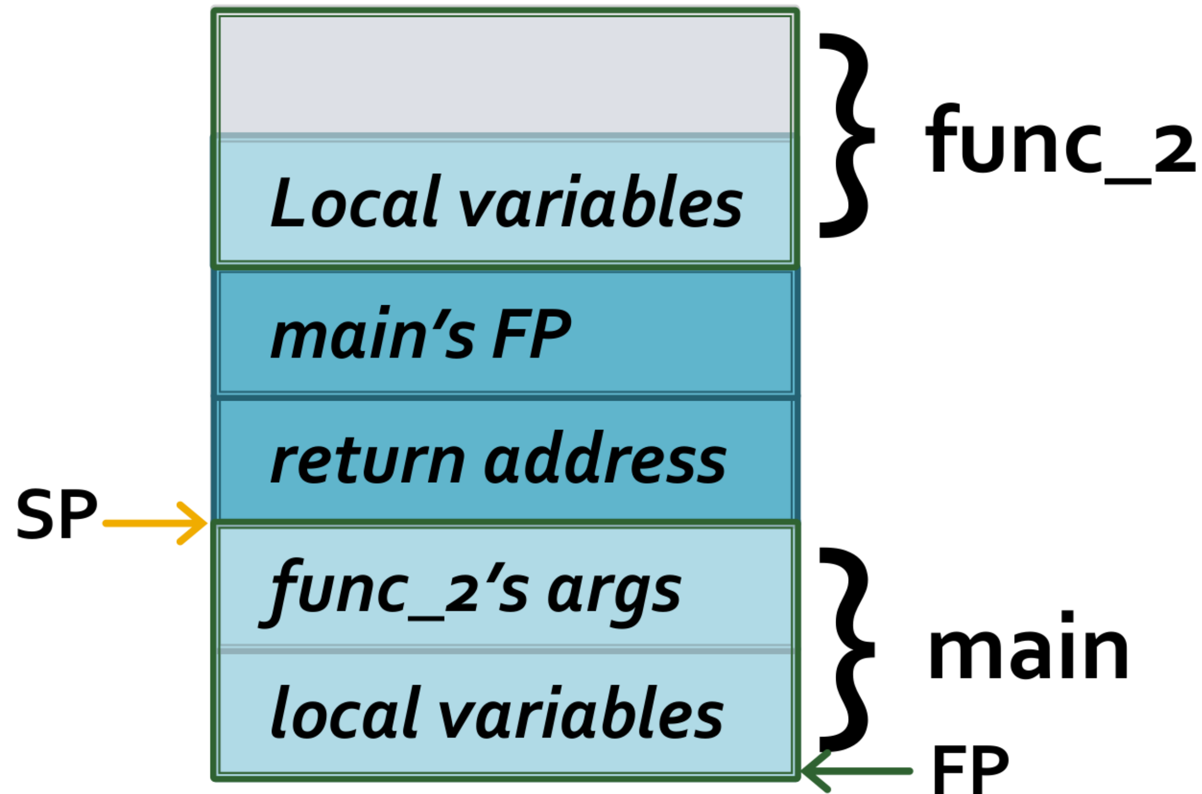
Standard Call Sequence



Standard Call Sequence



Standard Call Sequence



example2.c

```
void func_3(int a, int b) {
    char buf1[16];
    ...other logic...
}

int main() {
    func_3(3, 6);
}
```

example.s (x86)

```
void main() {
    func_3(3);}
```

```
void func_3(…) {
    char buf1[16];}
```

```
main:
   push   ebp
   mov    ebp, esp
   sub    esp, 8
   mov    [esp+4], 6
   mov    [esp], 3
   call   func_3
   leave
   ret
```

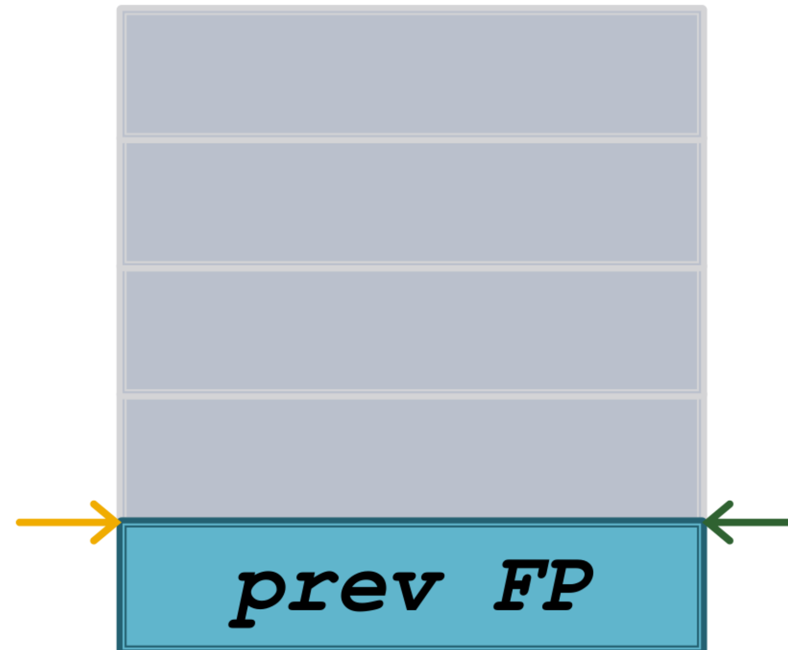
```
func_3:
   push   ebp
   mov    ebp, esp
   sub    esp, 0x10
   leave
   ret
```

example.s (x86)

```
main:
   push   ebp
   mov    ebp, esp
   sub    esp, 8
   mov    [esp+4], 6
   mov    [esp], 3
   call   func_3
   leave
   ret
```

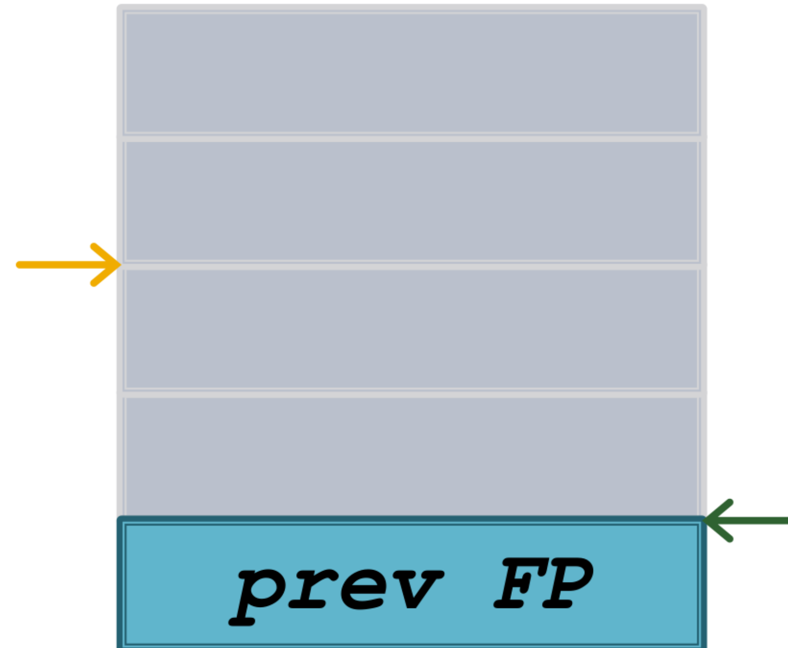
example.s (x86)

```
main:
   push   ebp
   mov    ebp, esp
   sub    esp, 8
   mov    [esp+4], 6
   mov    [esp], 3
   call   func_3
   leave
   ret
```



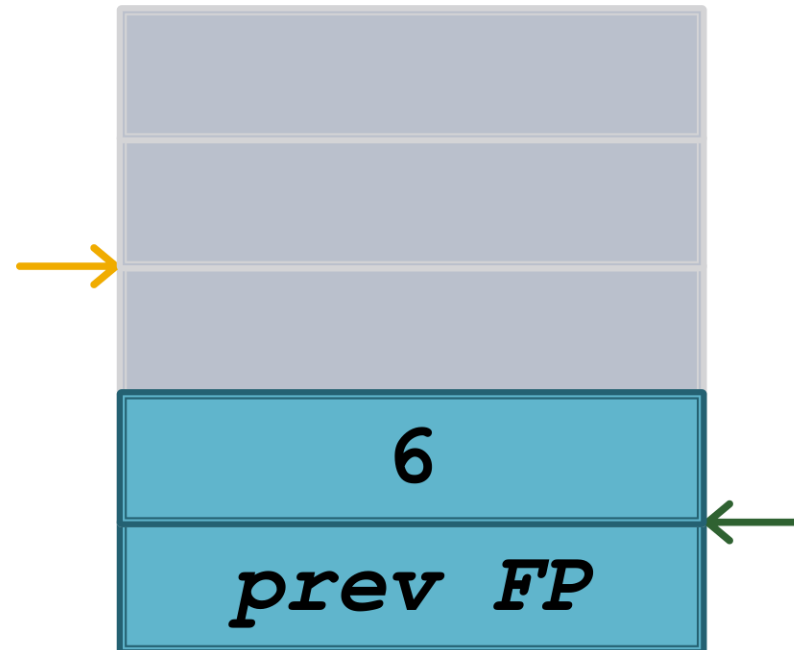
example.s (x86)

```
main:
    push    ebp
    mov     ebp, esp
    sub     esp, 8
    mov     [esp+4], 6
    mov     [esp], 3
    call    func_3
    leave
    ret
```



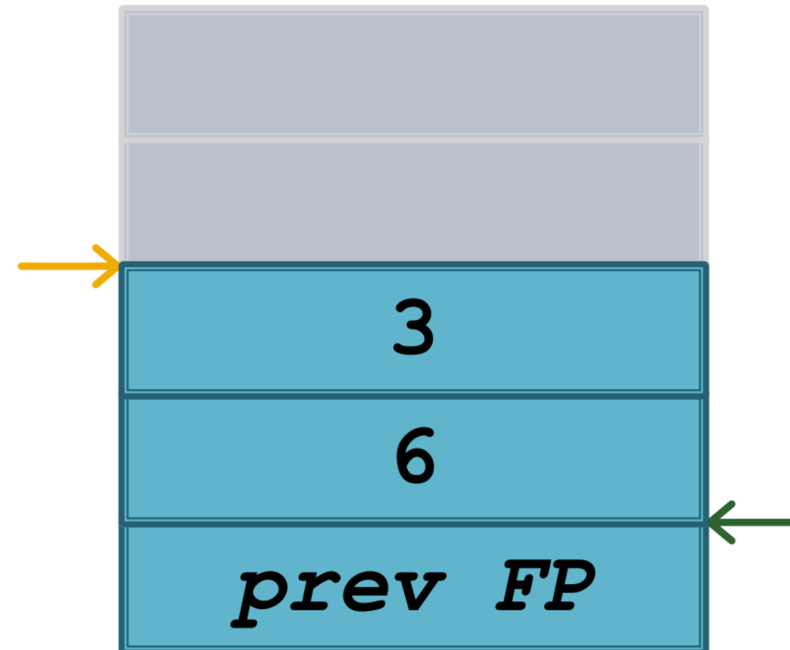
example.s (x86)

```
main:
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    [esp+4], 6
    mov    [esp], 3
    call   func_3
    leave
    ret
```



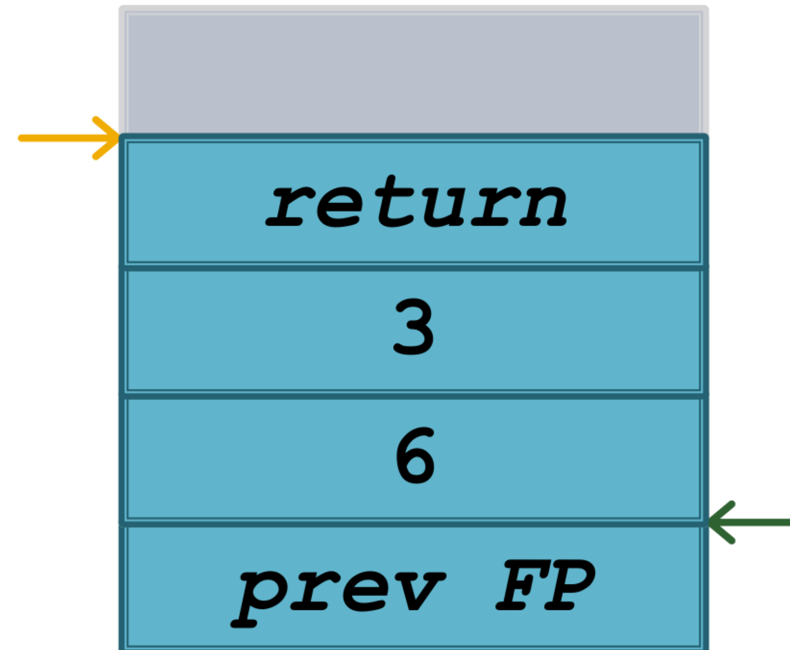
example.s (x86)

```
main:
    push    ebp
    mov     ebp, esp
    sub     esp, 8
    mov     [esp+4], 6
    mov     [esp], 3
    call    func_3
    leave
    ret
```



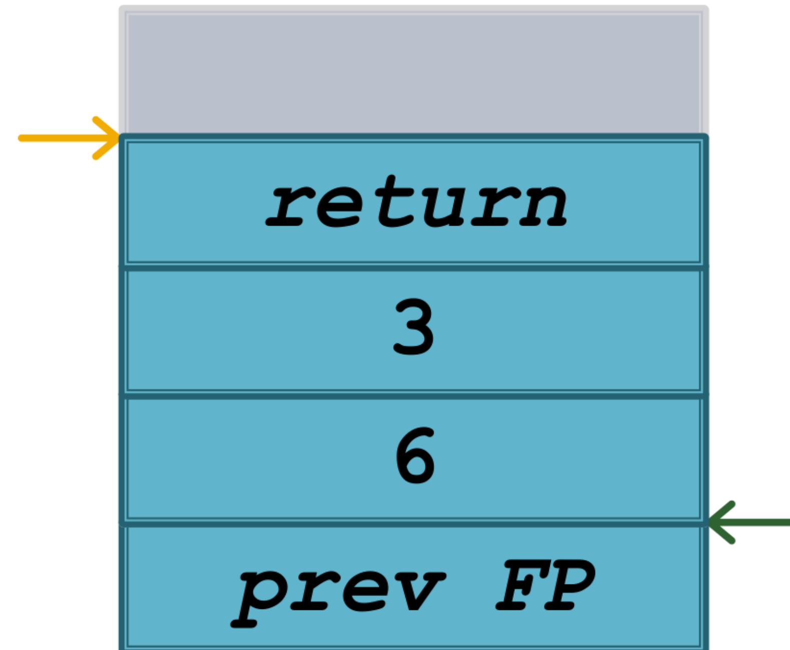
example.s (x86)

```
main:
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    [esp+4], 6
    mov    [esp], 3
    call   func_3
    leave
    ret
```



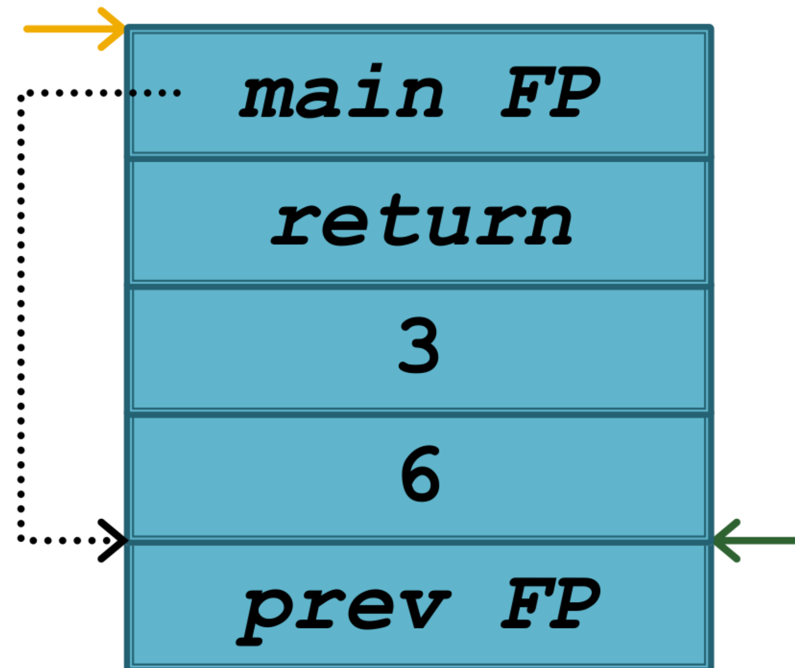
example.s (x86)

```
func_3:
  push   ebp
  mov    ebp, esp
  sub    esp, 0x10
  leave
  ret
```



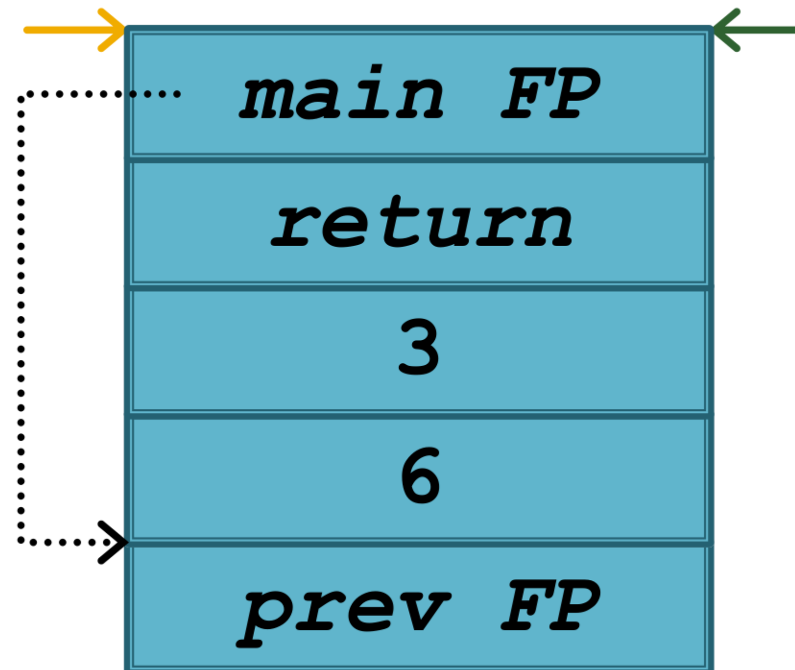
example.s (x86)

```
func_3:
  push   ebp
  mov    ebp, esp
  sub    esp, 0x10
  leave
  ret
```



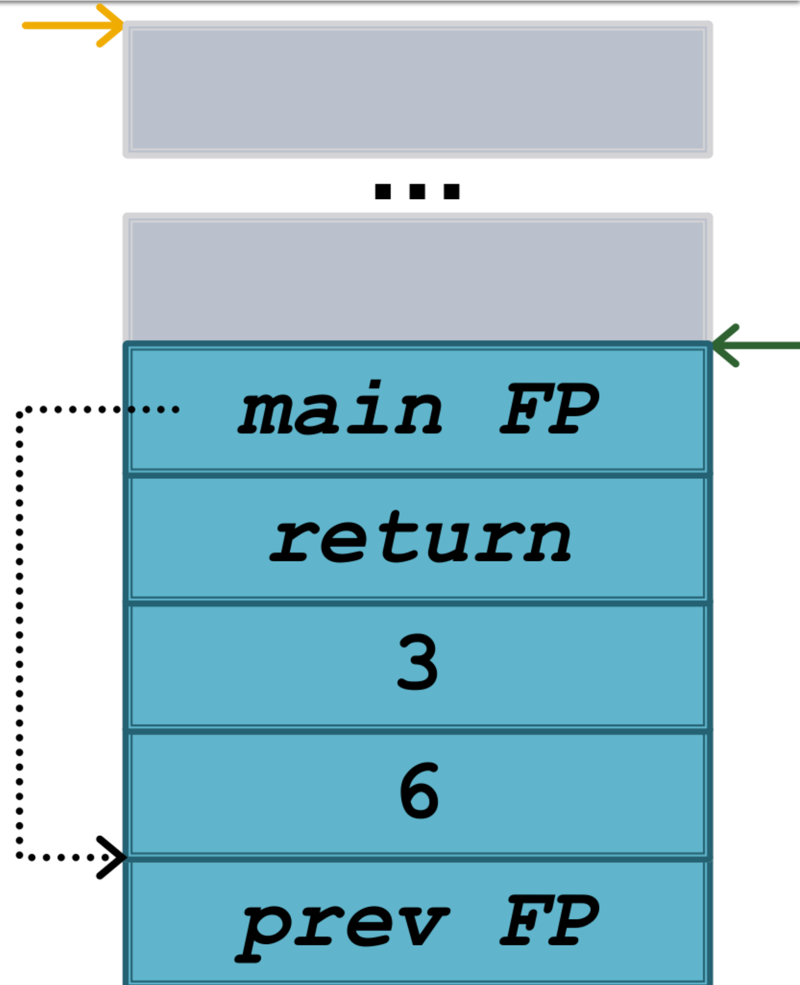
example.s (x86)

```
func_3:
  push   ebp
  mov    ebp, esp
  sub    esp, 0x10
  leave
  ret
```



example.s (x86)

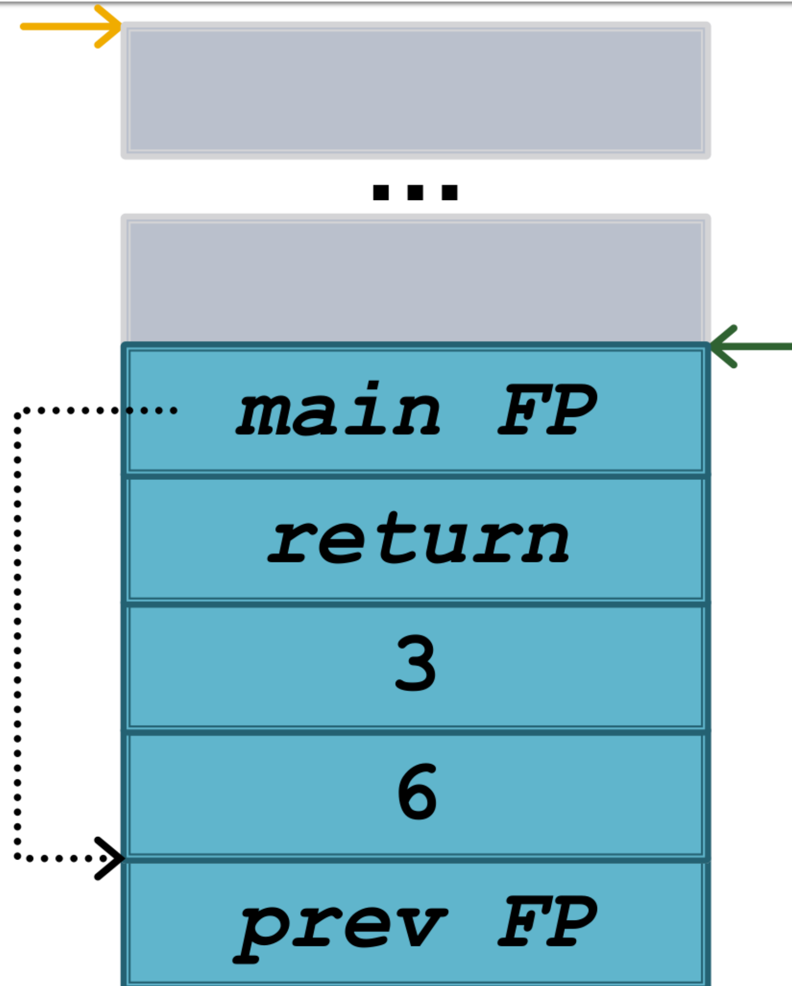
```
func_3:
  push   ebp
  mov    ebp, esp
  sub    esp, 0x10
  leave
  ret
```



example.s (x86)

```
func_3:
  push   ebp
  mov    ebp, esp
  sub    esp, 0x10
  leave
  ret
```

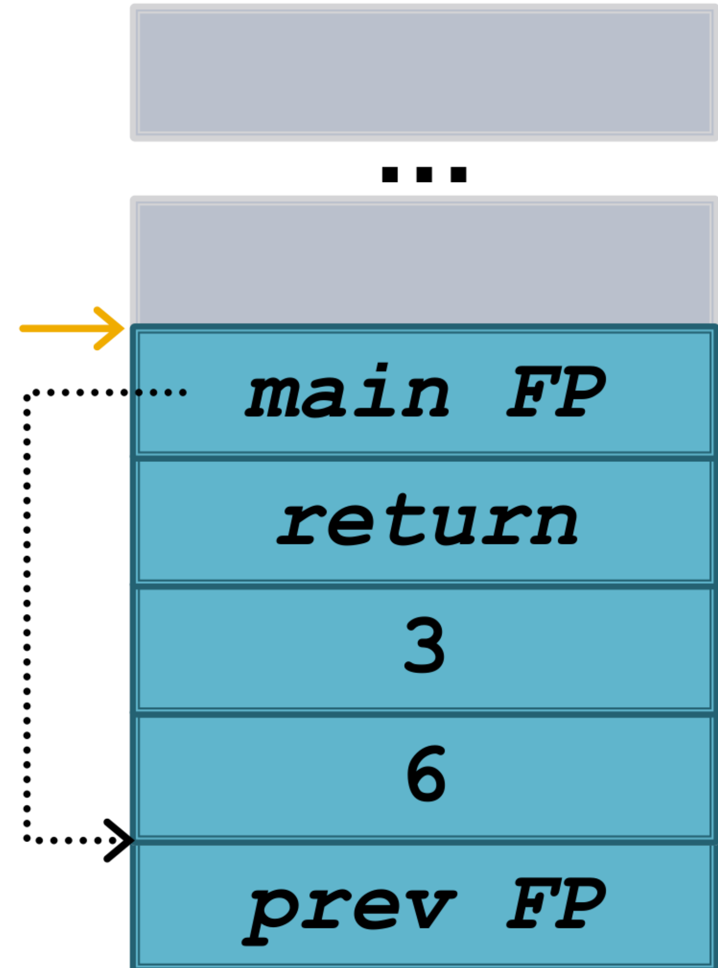
```
mov   esp, ebp
pop   ebp
```



example.s (x86)

```
func_3:
   push   ebp
   mov    ebp, esp
   sub    esp, 0x10
   leave
   ret
```

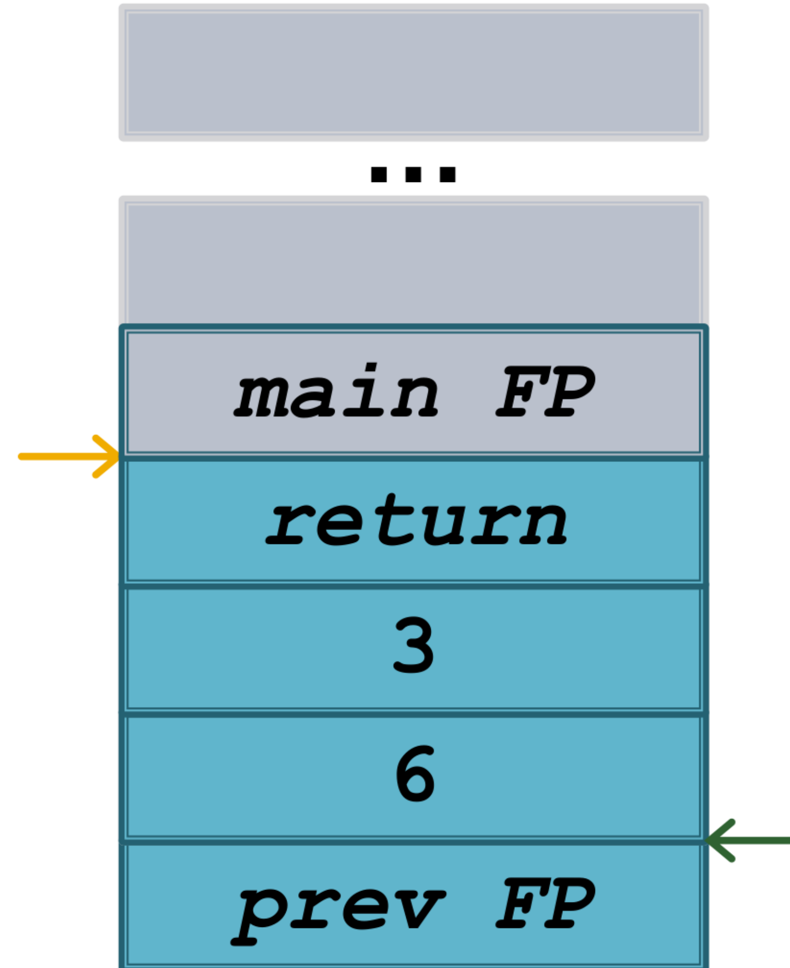
```
mov   esp, ebp
pop   ebp
```



example.s (x86)

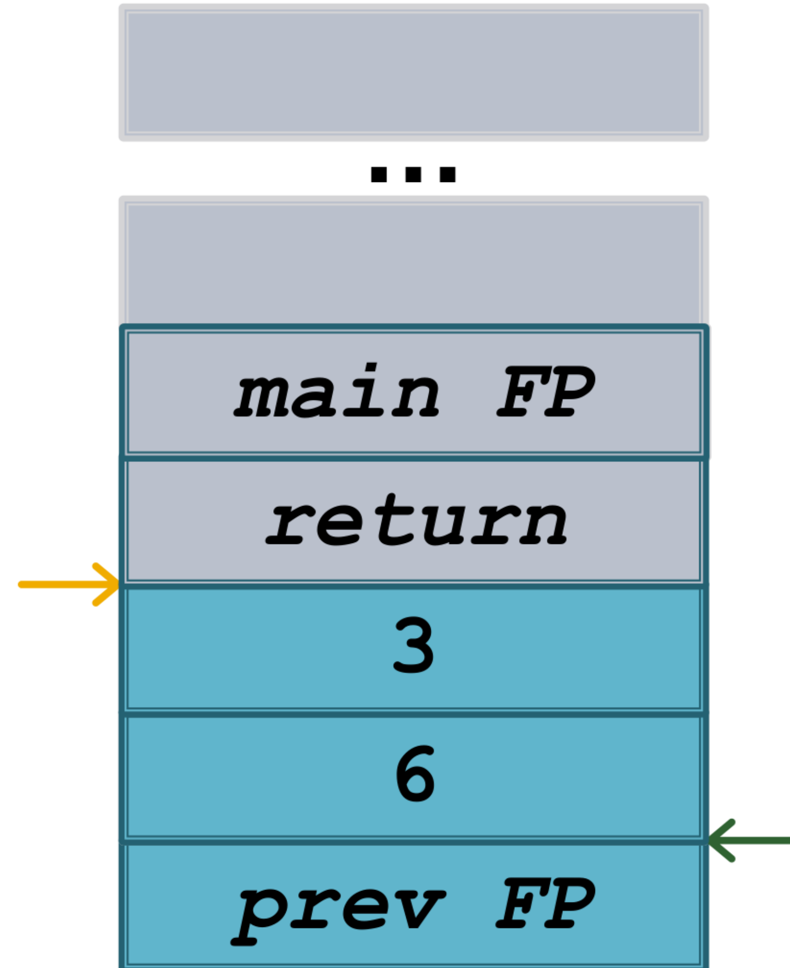
```
func_3:
  push   ebp
  mov    ebp, esp
  sub    esp, 0x10
  leave
  ret
```

```
mov   esp, ebp
pop   ebp
```



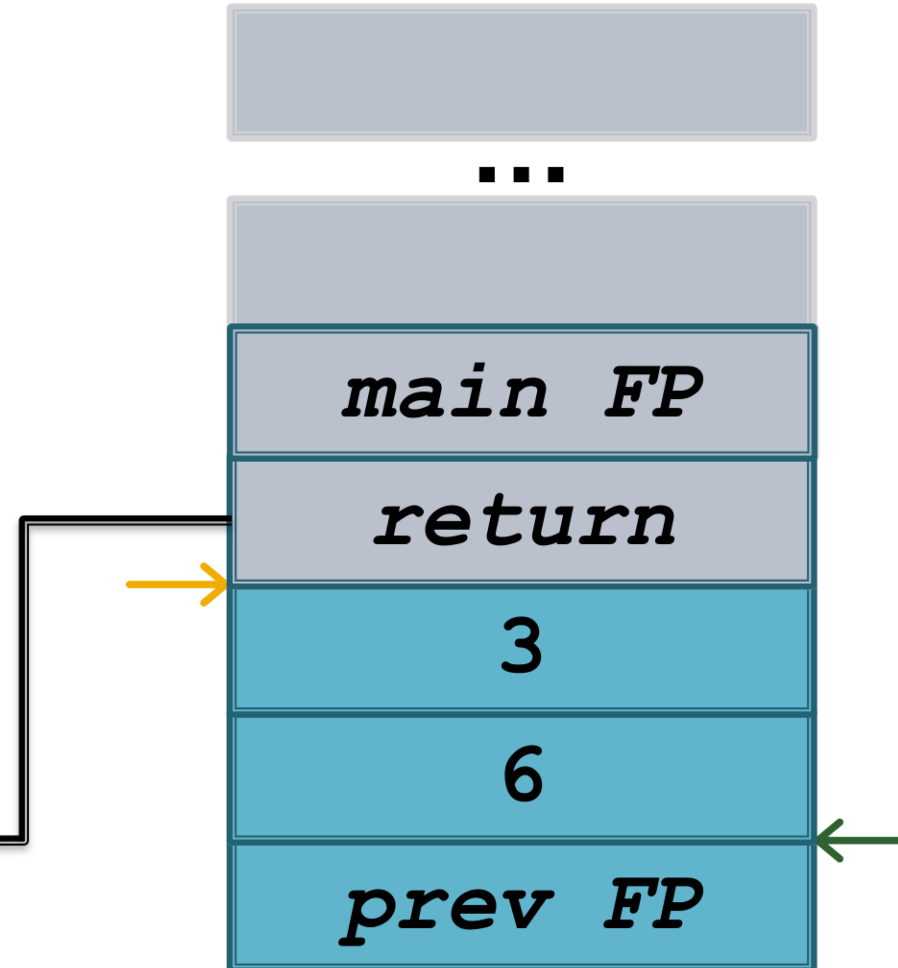
example.s (x86)

```
func_3:
  push   ebp
  mov    ebp, esp
  sub    esp, 0x10
  leave
  ret
```



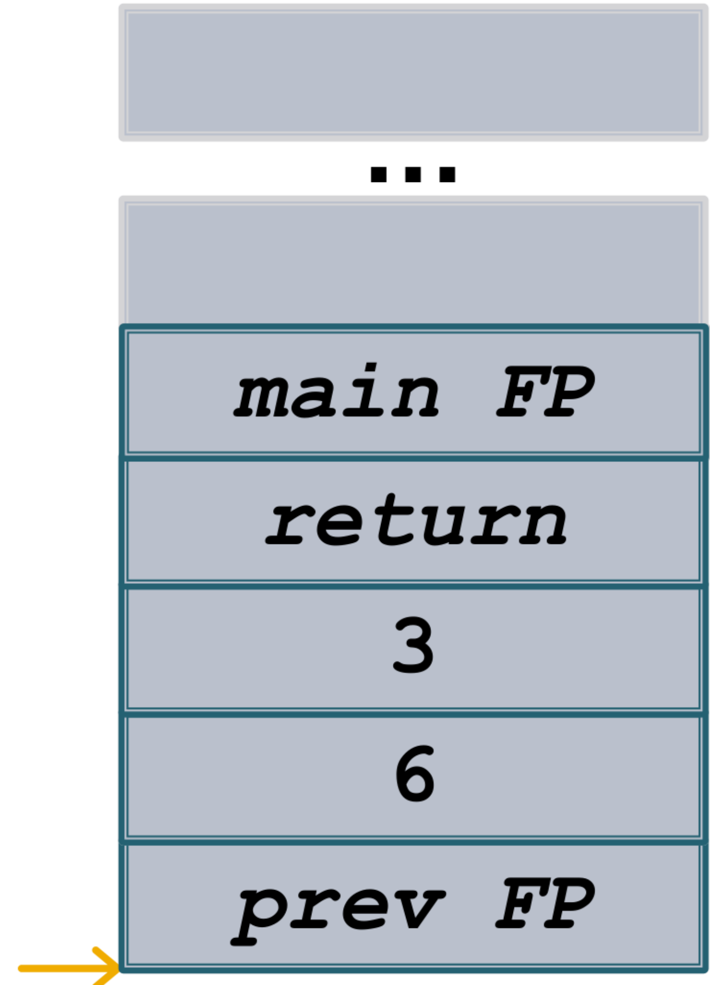
example.s (x86)

```
main:
    push  ebp
    mov   ebp, esp
    sub   esp, 8
    mov   [esp+4], 6
    mov   [esp], 3
    call  func_3
    leave
    ret
```



example.s (x86)

```
main:
  push  ebp
  mov   ebp, esp
  sub   esp, 8
  mov   [esp+4], 6
  mov   [esp], 3
  call  func_3
  leave
  ret
```



Buffer overflow example

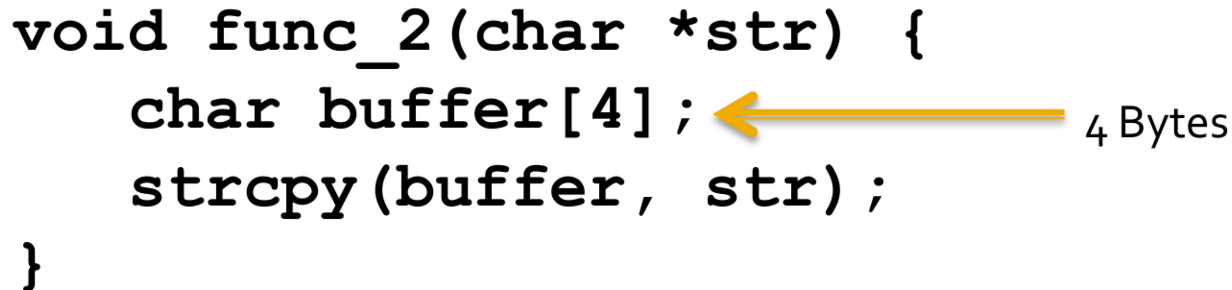
```
void func_2(char *str) {
    char buffer[4];
    strcpy(buffer, str);
}

int main() {
   char str = “1234567890AB”;
   func_2(str);
}
```

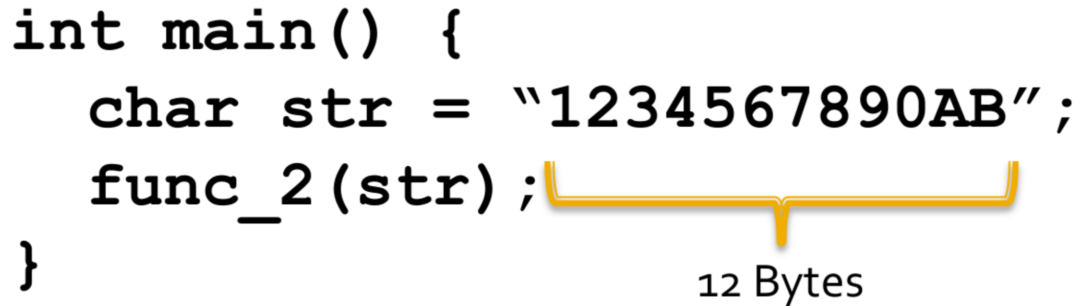
Buffer overflow example

```
void func_2(char *str) {
    char buffer[4];
    strcpy(buffer, str);
}

int main() {
   char str = "1234567890AB";
   func_2(str);
}
```

12 Bytes

Buffer overflow example





example2.s (x86)

```
int main() {
   char str = "1234567890AB";
   func_2(str);}
```

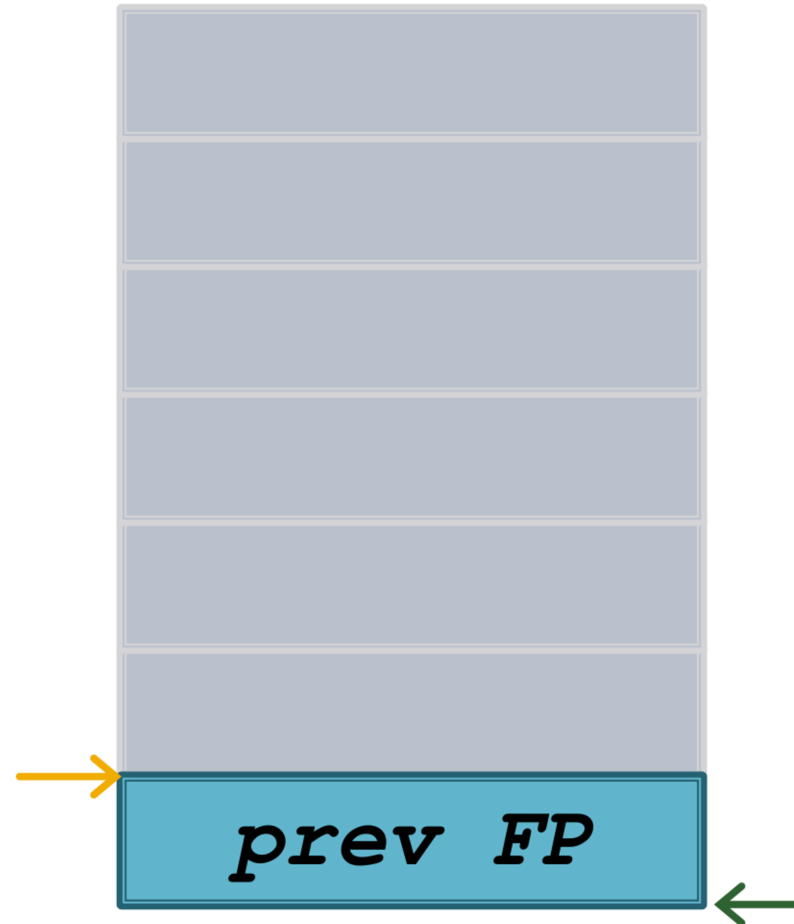
```
void func_2(char *str) {
   char buffer[4];
   strcpy(buffer, str);}
```

```
main:
   push   ebp
   mov    ebp, esp
   push   str_ptr
   call   func_2
   leave
   ret
RODATA:
str_ptr:"1234567890AB"
```

```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   ebp - 4
   call   strcpy
   leave
   ret
```

example2.s (x86)

```
main:
   push   ebp
   mov    ebp, esp
   push   str_ptr
   call   func_2
   leave
   ret
```




```
str_ptr: "1234567890AB"
```

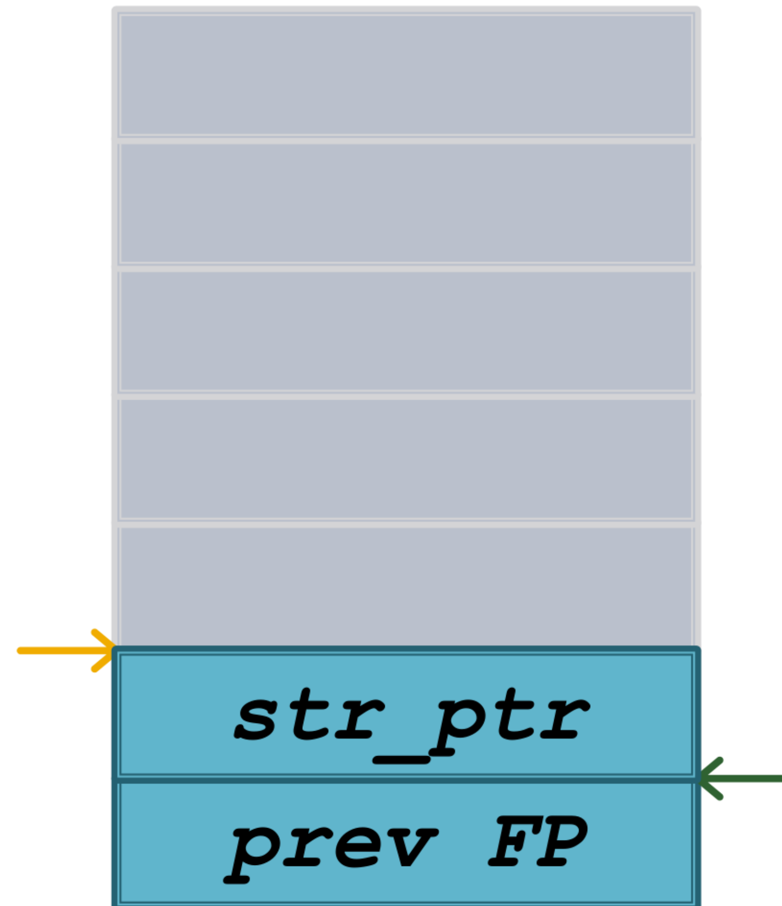
example2.s (x86)

```
main:
   push  ebp
   mov   ebp, esp
   push  str_ptr
   call  func_2
   leave
   ret
```

```
str_ptr: "1234567890AB"
```

example2.s (x86)

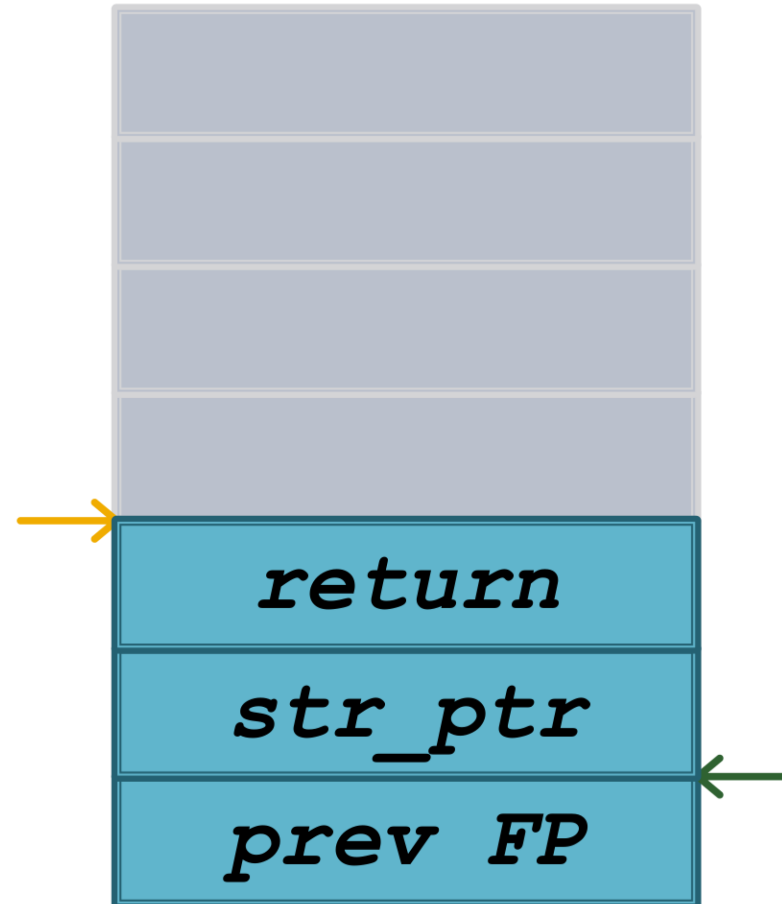
```
main:
    push   ebp
    mov    ebp, esp
    push   str_ptr
    call   func_2
    leave
    ret
```




```
str_ptr: "1234567890AB"
```

example2.s (x86)

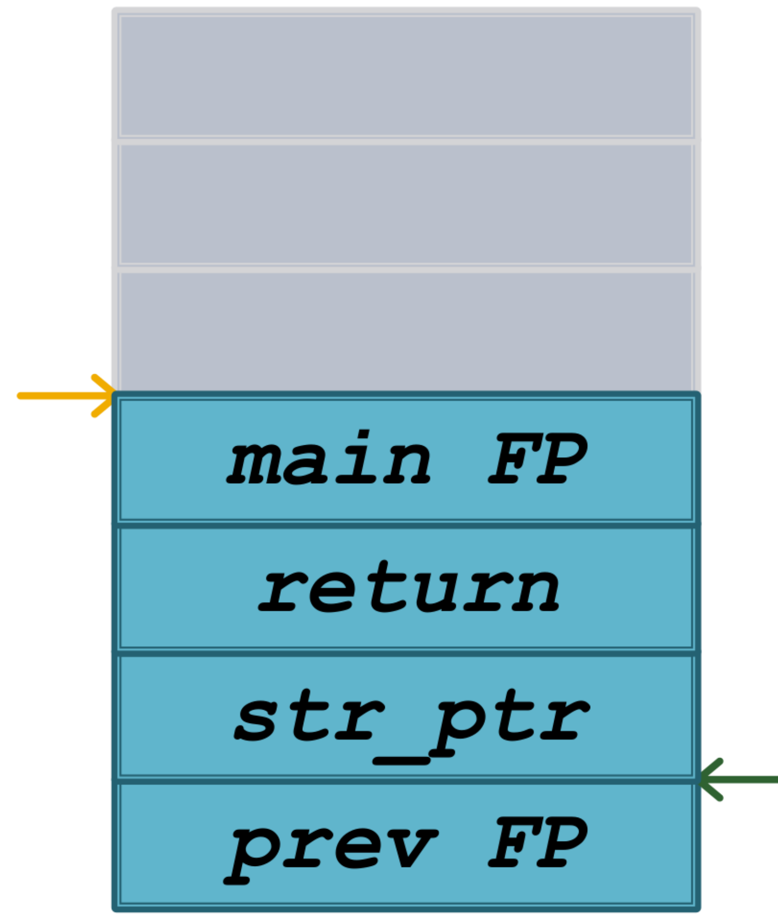
```
main:
    push   ebp
    mov    ebp, esp
    push   str_ptr
    call   func_2
    leave
    ret
```




```
str_ptr: "1234567890AB"
```

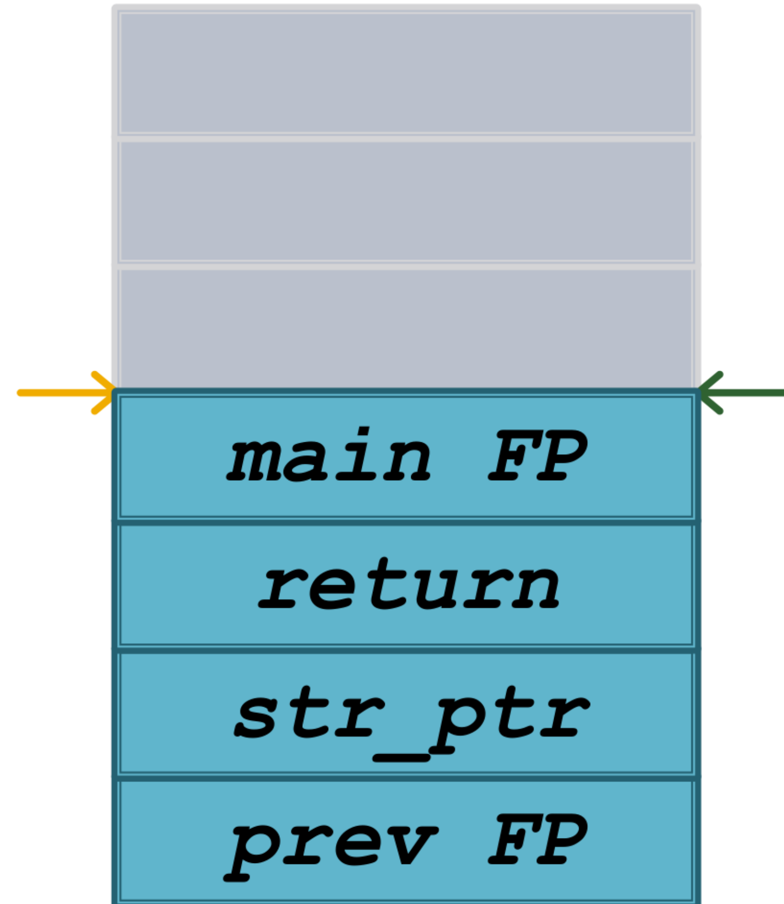
example2.s (x86)

```
func_2:
  push  ebp
  mov   ebp, esp
  sub   esp, 4
  push  [ebp + 8]
  push  ebp - 4
  call  strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



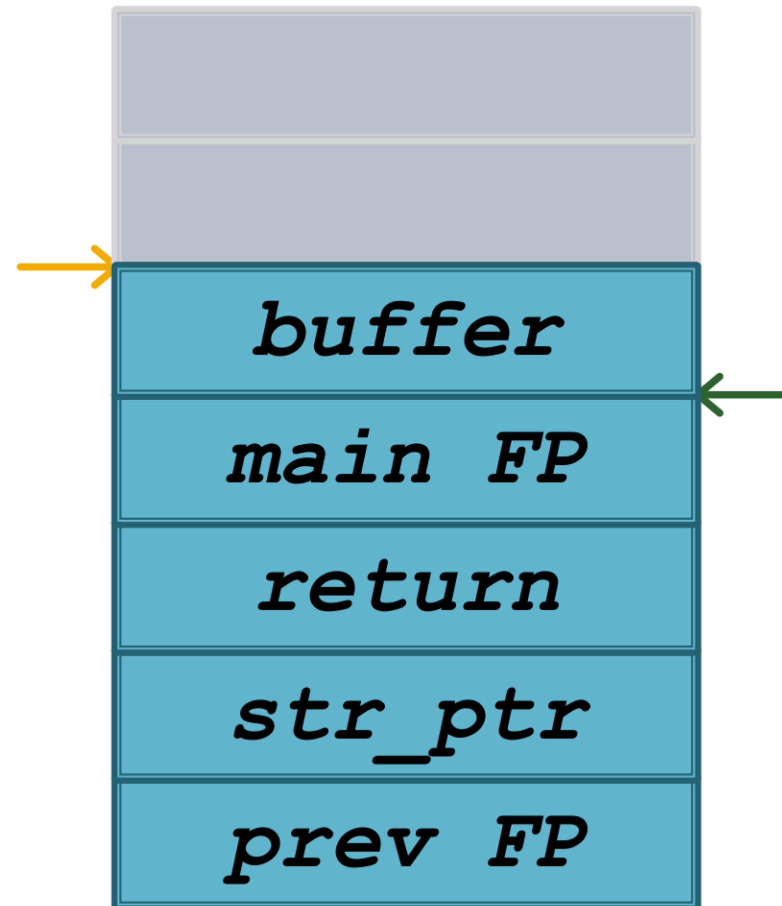
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



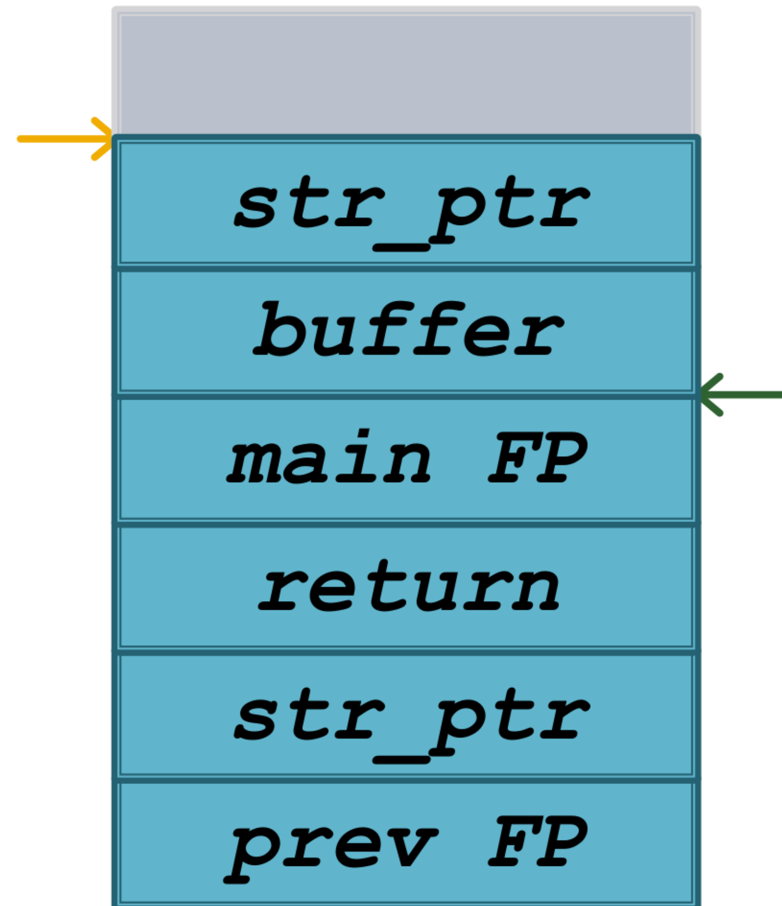
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



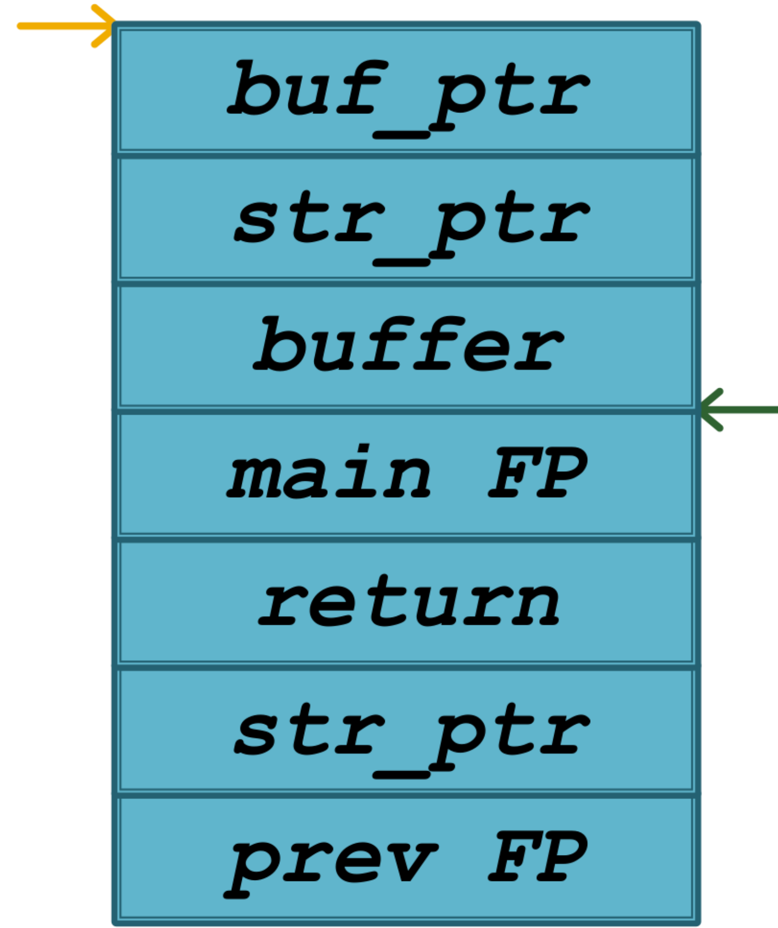
example2.s (x86)

```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   ebp - 4
   call   strcpy
   leave
   ret
str_ptr: "1234567890AB"
```



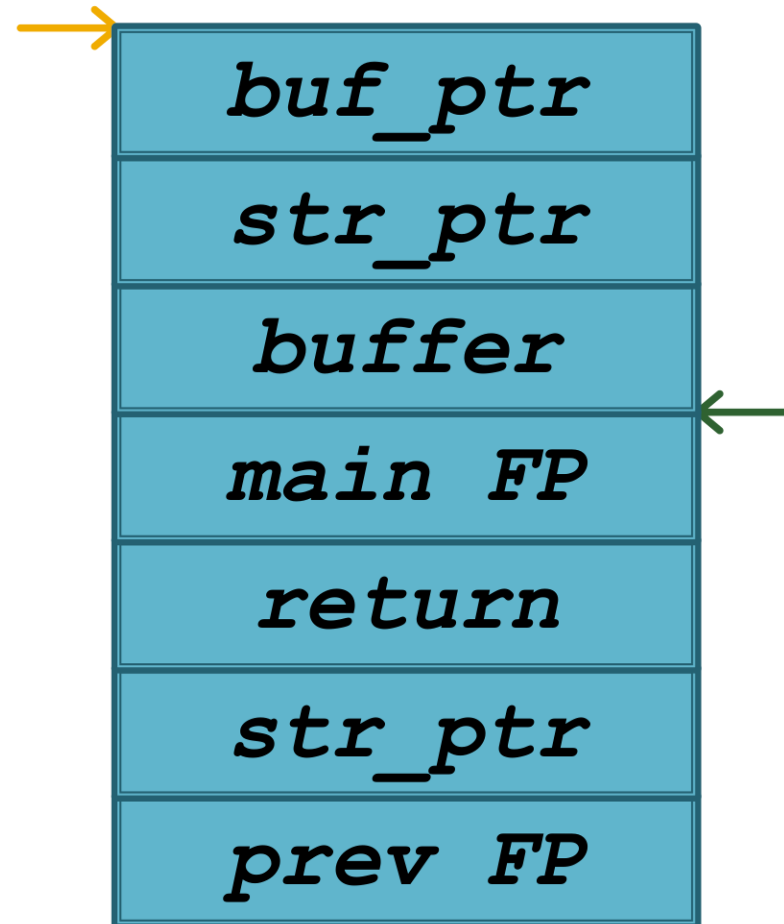
example2.s (x86)

```
func_2:
  push  ebp
  mov   ebp, esp
  sub   esp, 4
  push  [ebp + 8]
  push  ebp - 4
  call  strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



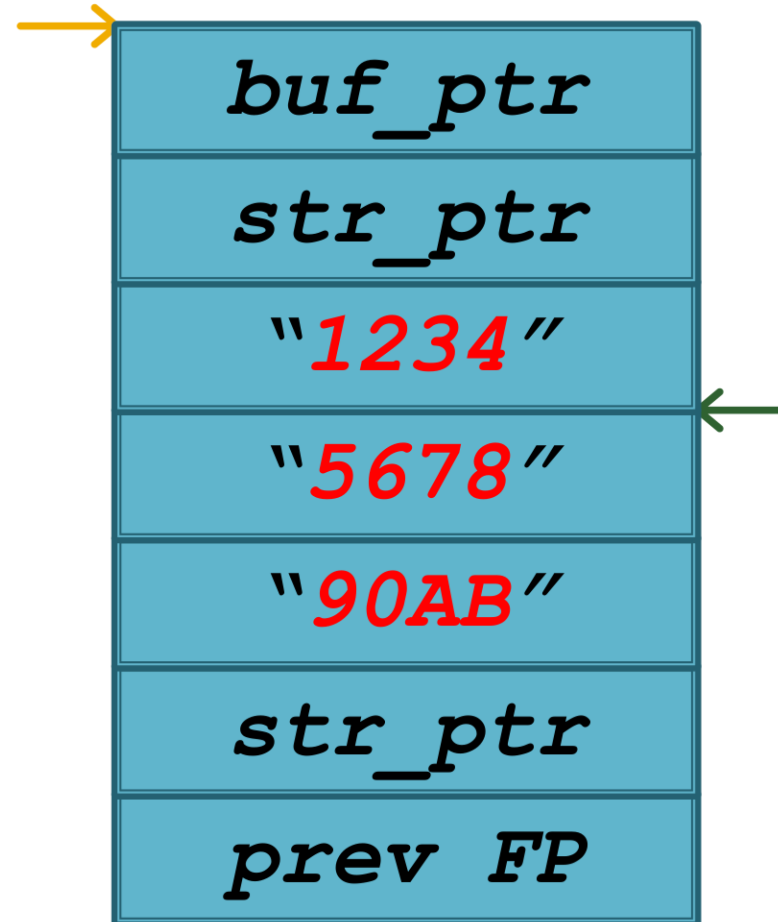
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



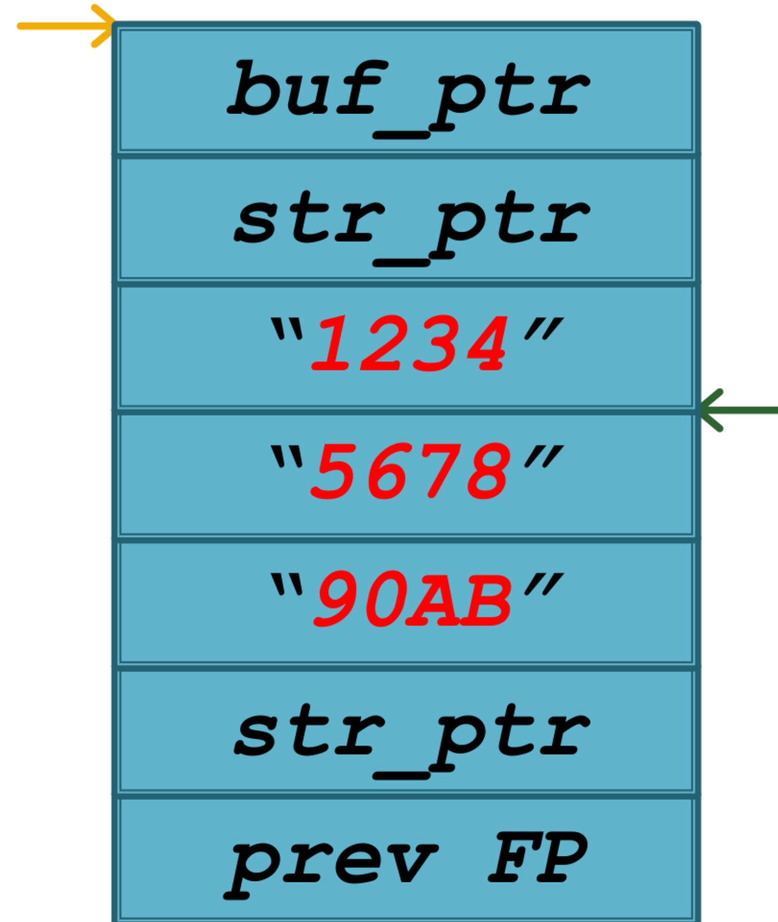
example2.s (x86)

```
func_2:
  push  ebp
  mov   ebp, esp
  sub   esp, 4
  push  [ebp + 8]
  push  ebp - 4
  call  strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



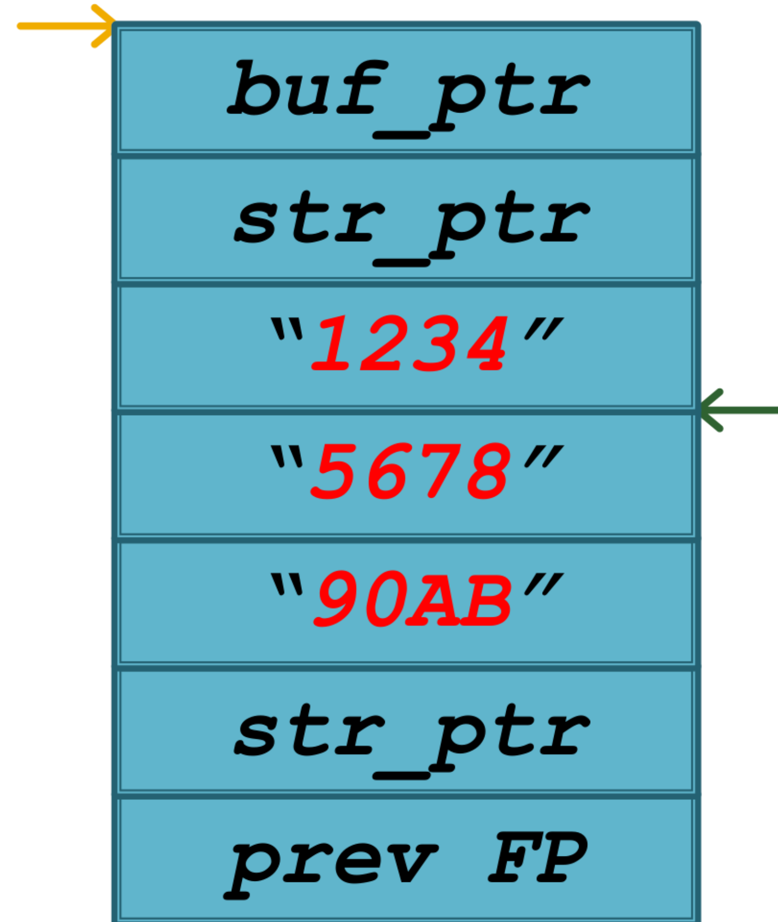
example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: “1234567890AB”
```



example2.s (x86)

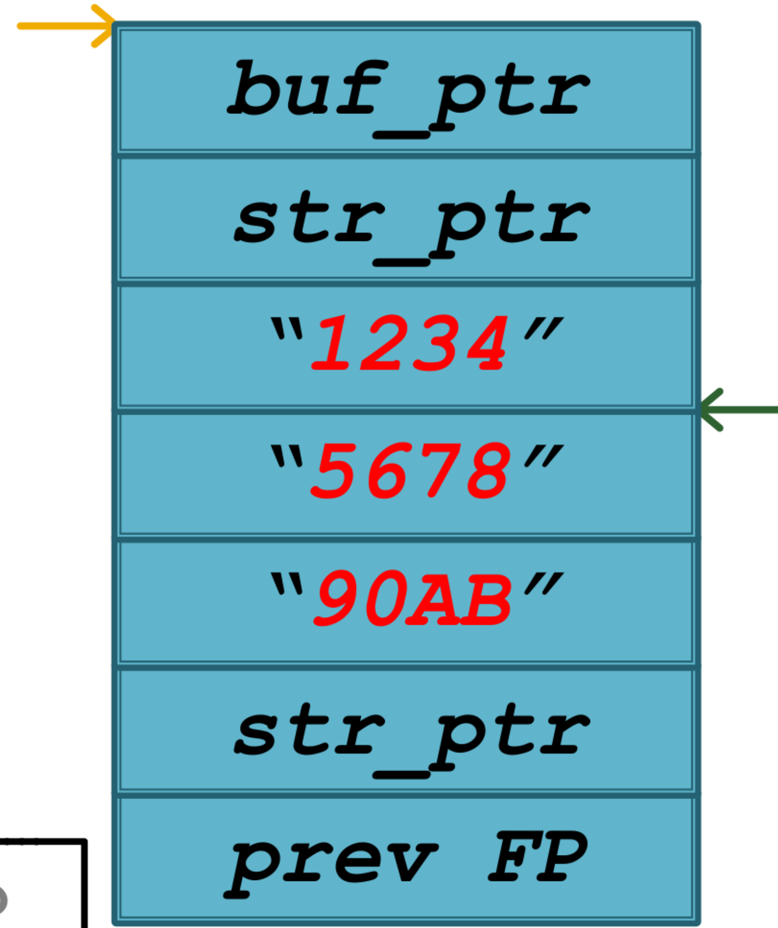
```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr: "1234567890AB"
```



example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr:
```

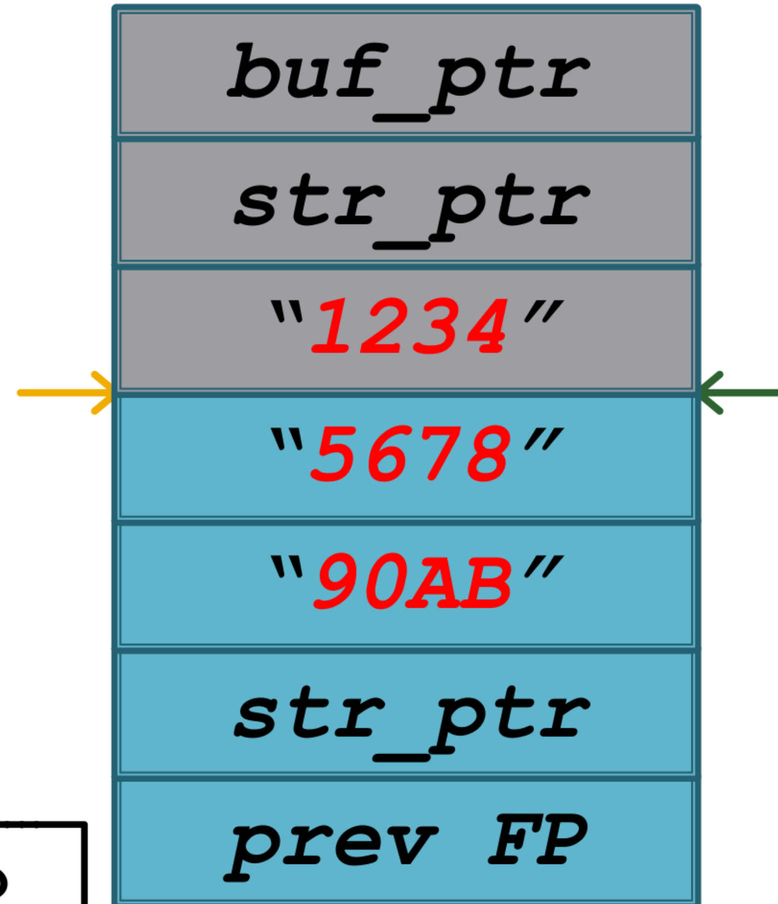
```
mov   esp, ebp
pop   ebp
```



example2.s (x86)

```
func_2:
  push   ebp
  mov    ebp, esp
  sub    esp, 4
  push   [ebp + 8]
  push   ebp - 4
  call   strcpy
  leave
  ret
str_ptr:
```

```
mov   esp, ebp
pop   ebp
```

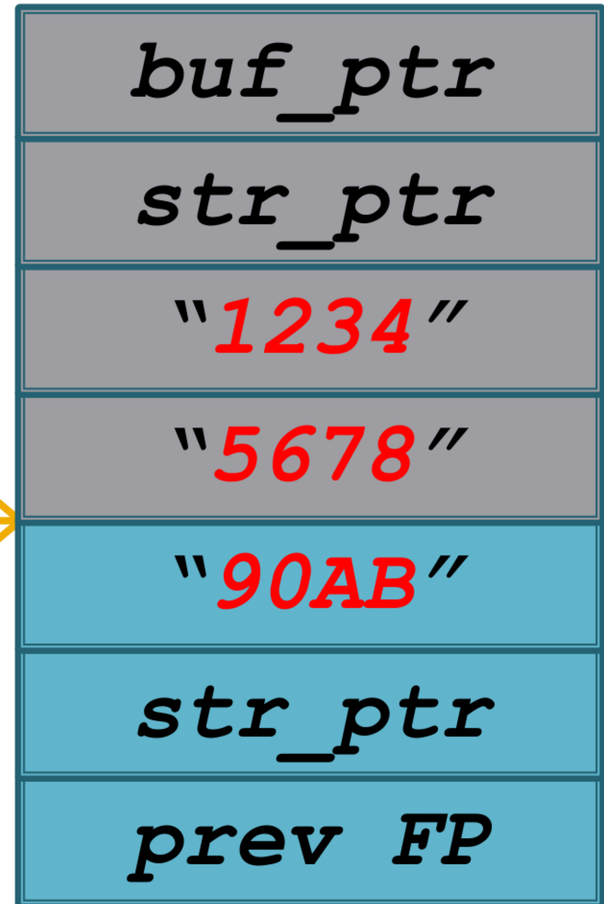


example2.s (x86)

?? FP ← ?? == 0x35363738

```
func_2:
   push  ebp
   mov   ebp, esp
   sub   esp, 4
   push  [ebp + 8]
   push  ebp - 4
   call  strcpy
   leave
   ret
str_ptr:
```

```
mov   esp, ebp
pop   ebp
```

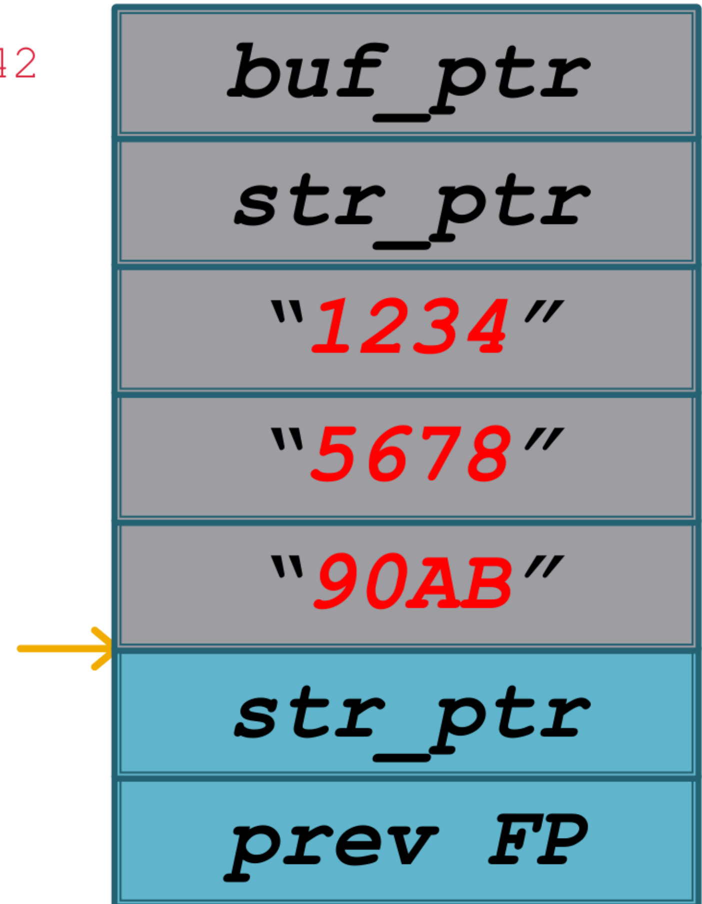


example2.s (x86)

?? FP ← ?? == 0x35363738

?? EIP ?? == 0x39304142

```
func_2:
   push   ebp
   mov    ebp, esp
   sub    esp, 4
   push   [ebp + 8]
   push   ebp - 4
   call   strcpy
   leave
   ret
str_ptr: "1234567890AB"
```



example.s (x86)

example.s (x86)

This program has performed an illegal operation and will be shut down.

If the problem persists, contact the program vendor.

Close

Details>>

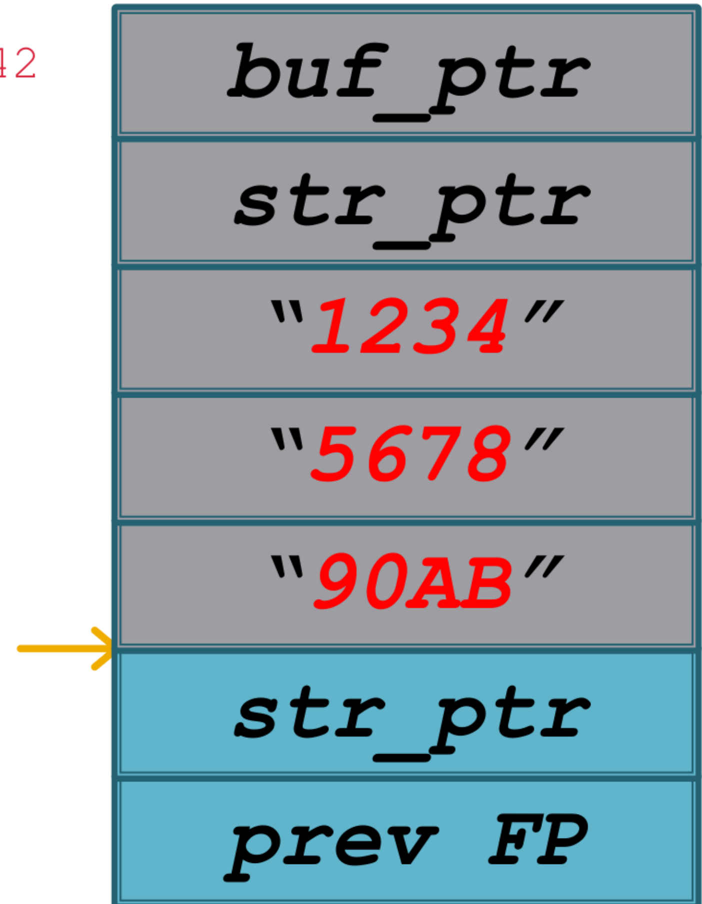
```
OPERA caused an invalid page fault in
module <unknown> at 0000:79e82379.
Registers:
EAX=79e82379 CS=015f EIP=79e82379 EFLGS=00000202
EBX=679e0000 SS=0167 ESP=0065f878 EBP=0065f8ac
ECX=67f879a8 DS=0167 ESI=67f330ec FS=0eaf
EDX=00000003 ES=0167 EDI=00000000 GS=0000
Bytes at CS:EIP:
```

example2.s (x86)

?? FP ← ?? == 0x35363738

?? EIP ?? == 0x39304142

```
func_2:
   push  ebp
   mov   ebp, esp
   sub   esp, 4
   push  [ebp + 8]
   push  ebp + 4
   call  strcpy
   leave
   ret
str_ptr: "123456789AB"
```



Buffer Overflows

Buffer overflows are class of memory corruption bugs where a program attempts to put **too-much data** into **too-small** of a memory allocation.

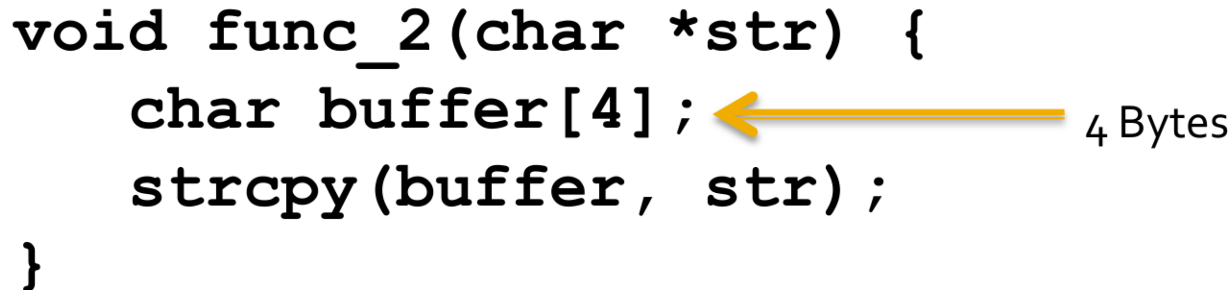
```
void print_name(char** argv) {
    char buf[10];
    strcpy(buf, argv[0]);
    printf("Running: %s", buf);
}
```

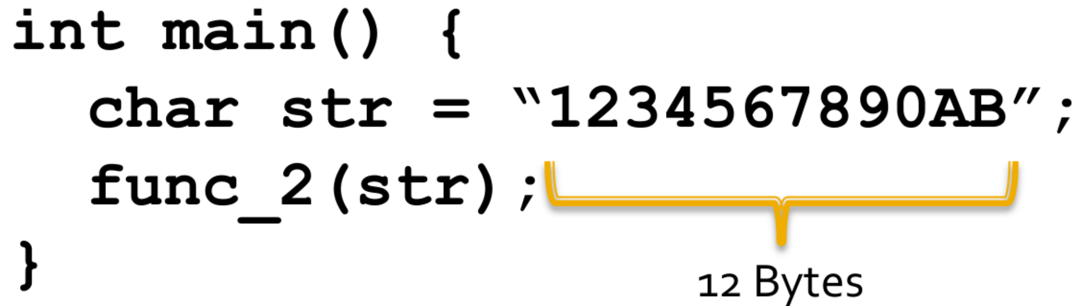
Binary Exploitation

Binary exploitation is the general name for techniques used intentionally trigger bugs in a way meaningful to the attacker.

- Not all buffer overflows are controllable
- Even if controllable, may not be exploitable
- Even if exploitable, may not be predictable
- Even if predictable, may not be useful

Buffer Overflow Example





Computer and Network Security

Lecture 10:
Buffer Overflows & Binary Exploitation

COMP-5370/6370
Fall2024