Memory management vulnerabilities

Frank Piessens

The concrete attack examples in these slides are based on the paper:
"Low-level Software Security by Example" by Erlingsson, Younan and Piessens

Introduction: the setting of this lecture

- System model:
    - Software in a C-like language compiled to a typical modern processor
- Attack model:
    - Attacker can interact with the software by providing input and reading output
    - Attacker knows the source code, and knows how code is compiled and executed
- Objectives of the lecture are to understand:
    - How software could be attacked in this setting
    - What the vulnerabilities are that enable these attacks
    - What defenses can help remove these vulnerabilities or mitigate these attacks

Example vulnerable C program

```c
#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");
}
```

Source: https://github.com/gerasdf/InsecureProgramming

Example vulnerable C program

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

Overview

- System model
- Attack scenarios
- Mitigating attacks
- Avoiding vulnerabilities
- Conclusions

System model

- The system under attack is a C program that has been compiled to run on a modern processor
- Hence, the details of the system under attack vary with
  - The underlying platform (processor, operating system, …)
  - The compiler used
- But fortunately, the general structure of processors, operating systems and compilers is sufficiently similar
- We will describe the system model abstractly, but for examples we will pick a specific system (e.g. gcc on Linux on x86)

Abstract model of the target platform

- Target platform consists of:
  - A byte-addressable memory (addresses MIN … MAX)
    - Could be either virtual memory (e.g., user-mode process) or physical memory (e.g., embedded code on a micro-processor)
  - A CPU with
    - Registers of word size (word = typically 4 or 8 bytes), including both general-purpose registers, as well as a program counter (PC), stack pointer (SP) and so forth
    - An instruction set with typical machine code instructions like
      - Arithmetic instructions
      - Memory access instructions
      - Branch and Call/return instructions
    - Instructions are stored in memory as sequence of bytes
      - Some processors have fixed instruction size, for others the size is variable

Mapping words in registers to memory

- We assume the processor is *little-endian*

0x107	0x07
0x106	0x06
0x105	0x05
0x104	0x04
0x103	0x03
0x102	0x02
0x101	0x01
0x100	0x00

Little-endian

0x104	0x07060504
0x100	0x03020100

Big-endian

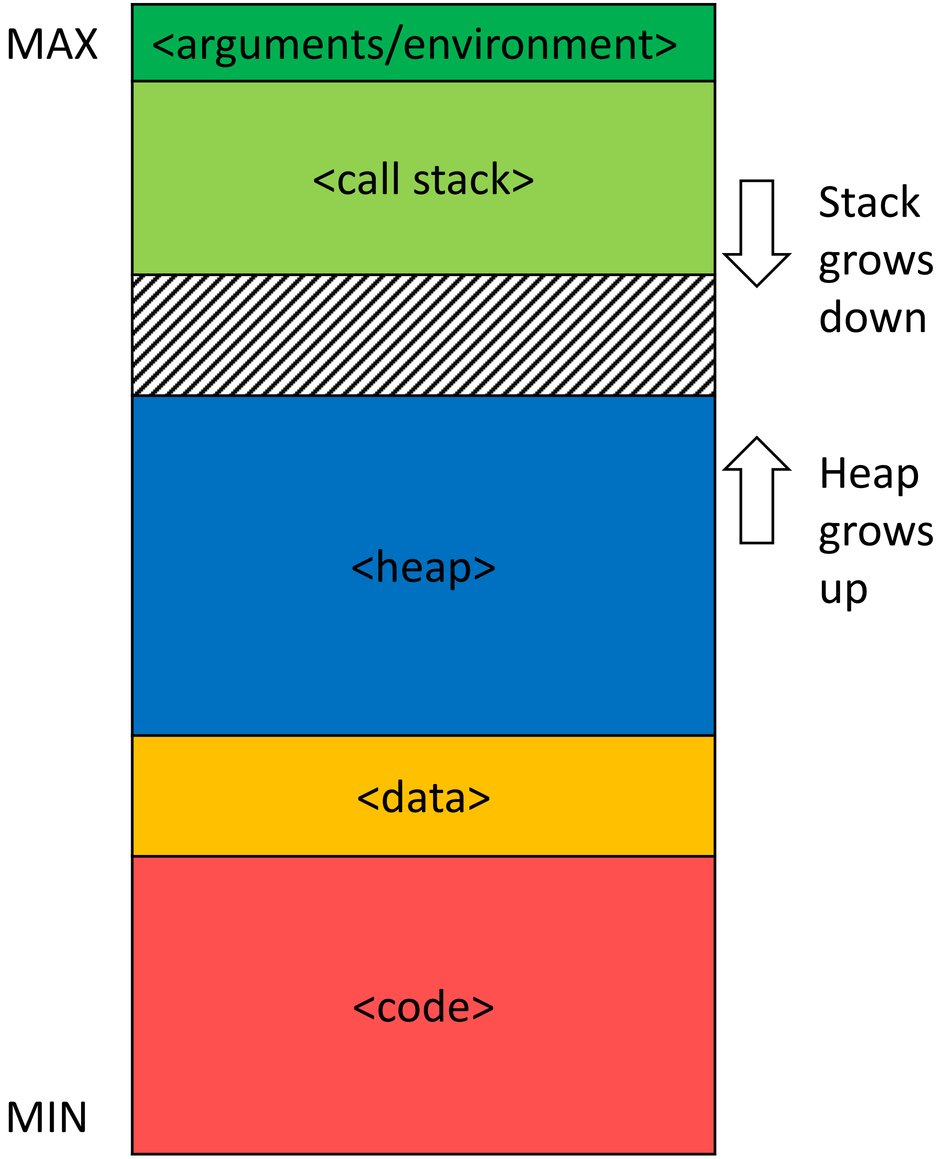
0x104	0x04050607
0x100	0x00010203

Source code model

- Simple C-like language
  - Types: char, int, void, pointers (e.g. char*, int**, …), arrays (e.g. char[10])
  - Local and global variables
    - Array variable is a pointer to the first element of the array
  - Statements and expressions:
    - Constants, variables, logical and arithmetic expressions, array indexing
    - If / while / sequencing / blocks / assignment / function calls
    - Library functions for I/O and memory management:
    - getchar(), putchar(),gets(), printf() + other typical C functions for I/O
    - malloc() and free()

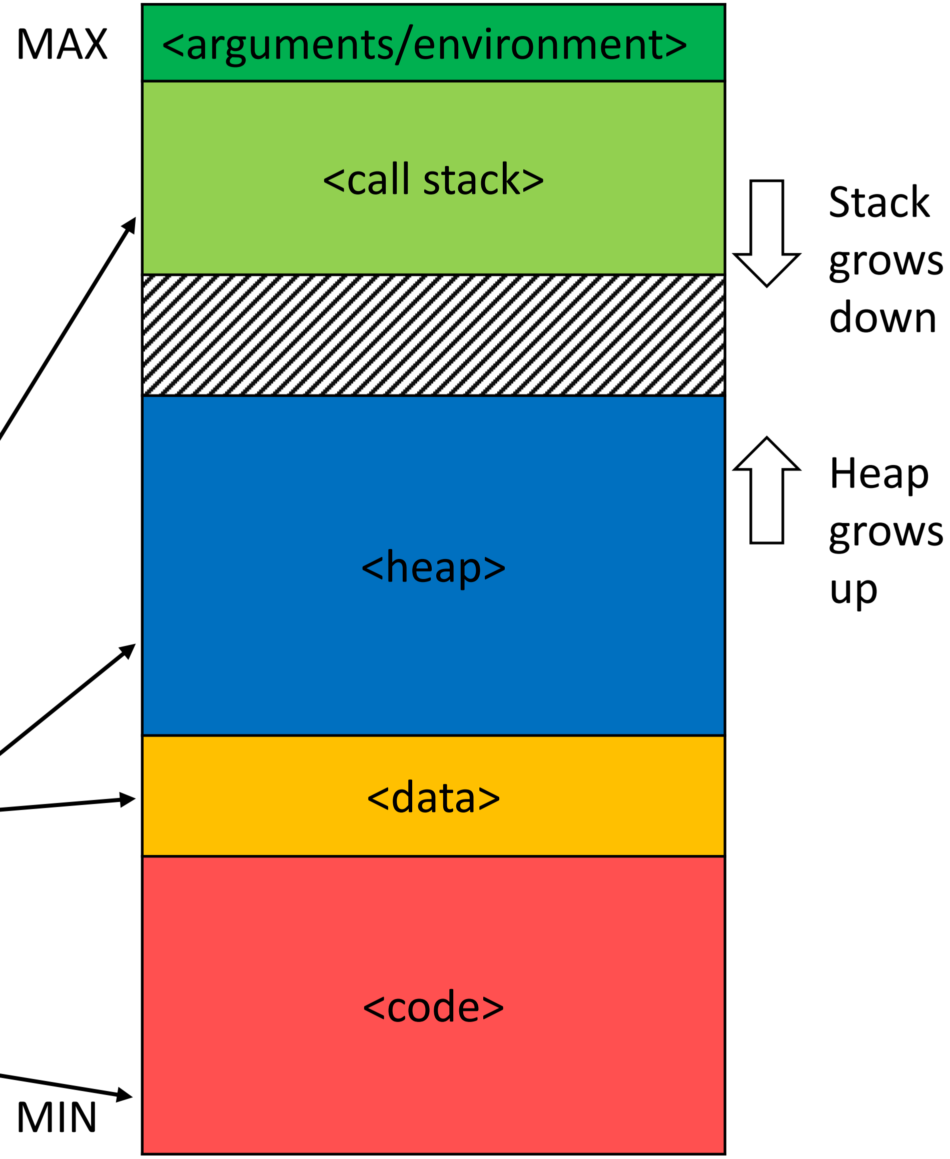
Compilation: overview

- Each function is compiled separately, and the resulting machine code is stored in a **code** section in memory
- Control-flow through the program is tracked by means of a **call stack**
- Variables used in the program are allocated in a number of ways:
  - Local variables are allocated on the call stack
  - Global variables are allocated in a dedicated **data** section in memory
  - Dynamic allocation is handled by a memory management library that manages a **heap**
    - malloc(), free(), …
- Pointers are represented as integer addresses, supporting pointer arithmetic
- Arrays are represented as pointers, indexing is similar to pointer arithmetic



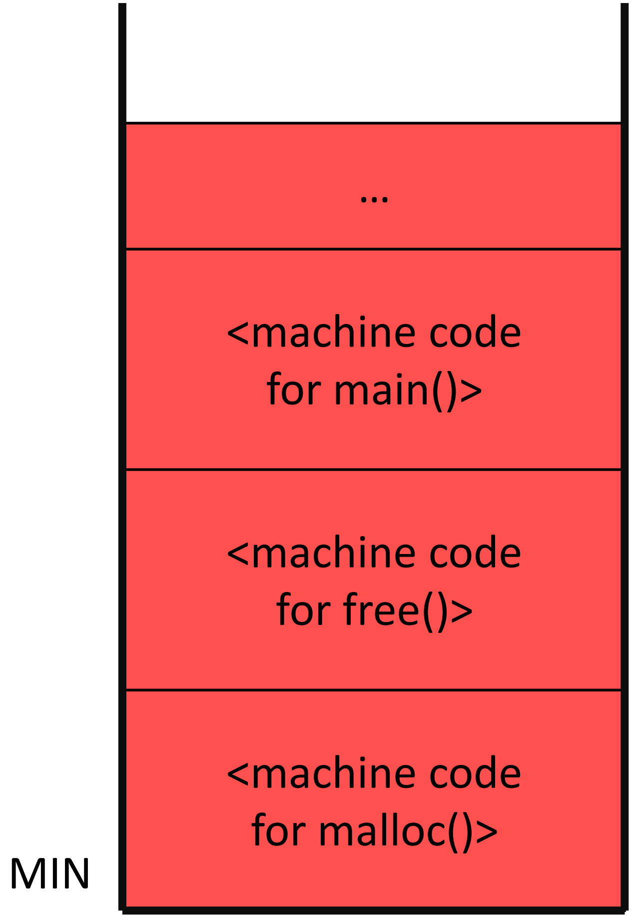
Example

```
#include <stdio.h>
#include <stdlib.h>

int g = 12;

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



Compilation: the code section

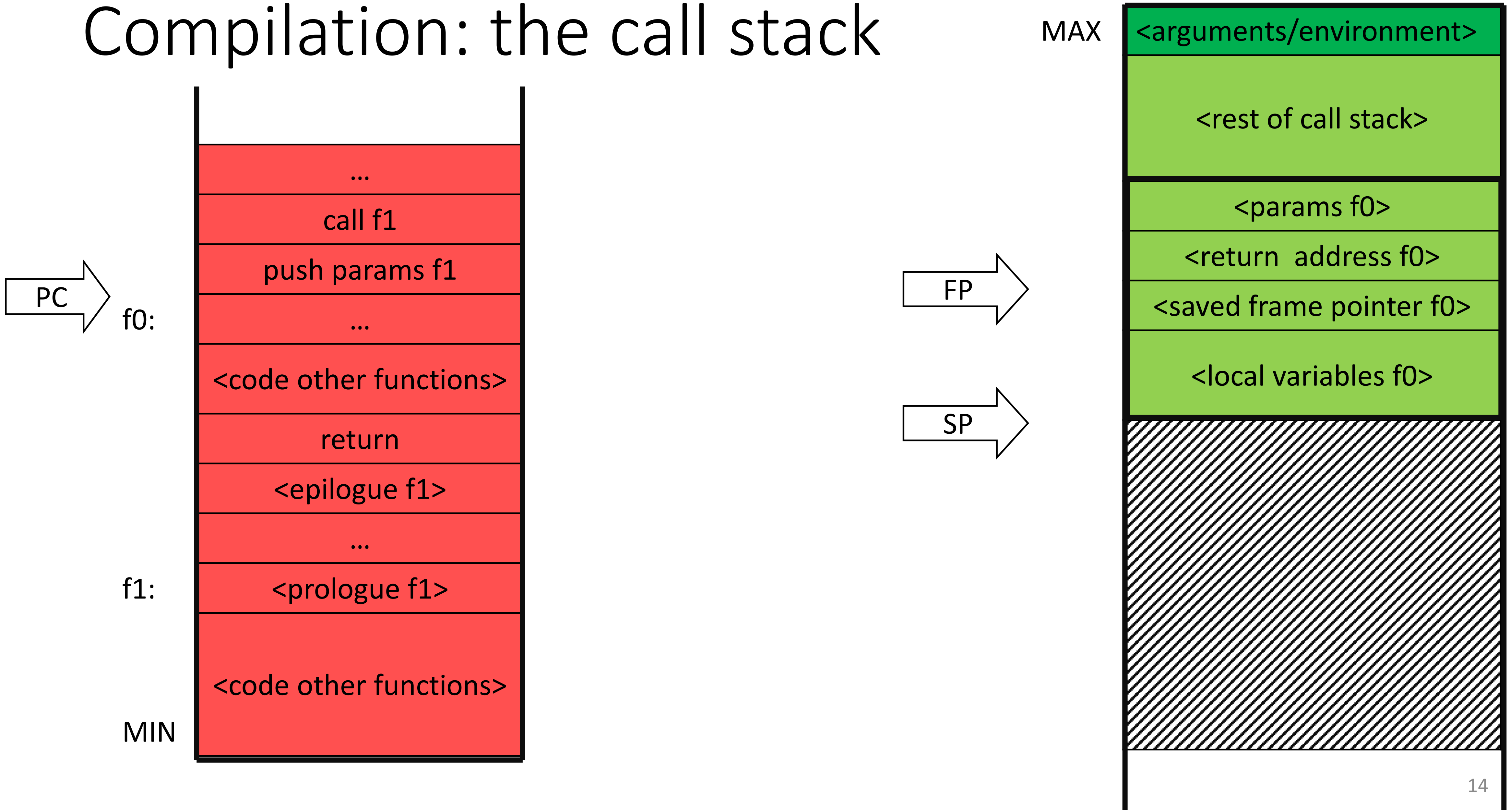
- Code for every function is compiled separately
  - Prologue: allocates space for activation record
  - Code for the body
  - Epilogue: put result in designated register, clear space for activation record
- We make abstraction of shared dynamically loaded libraries



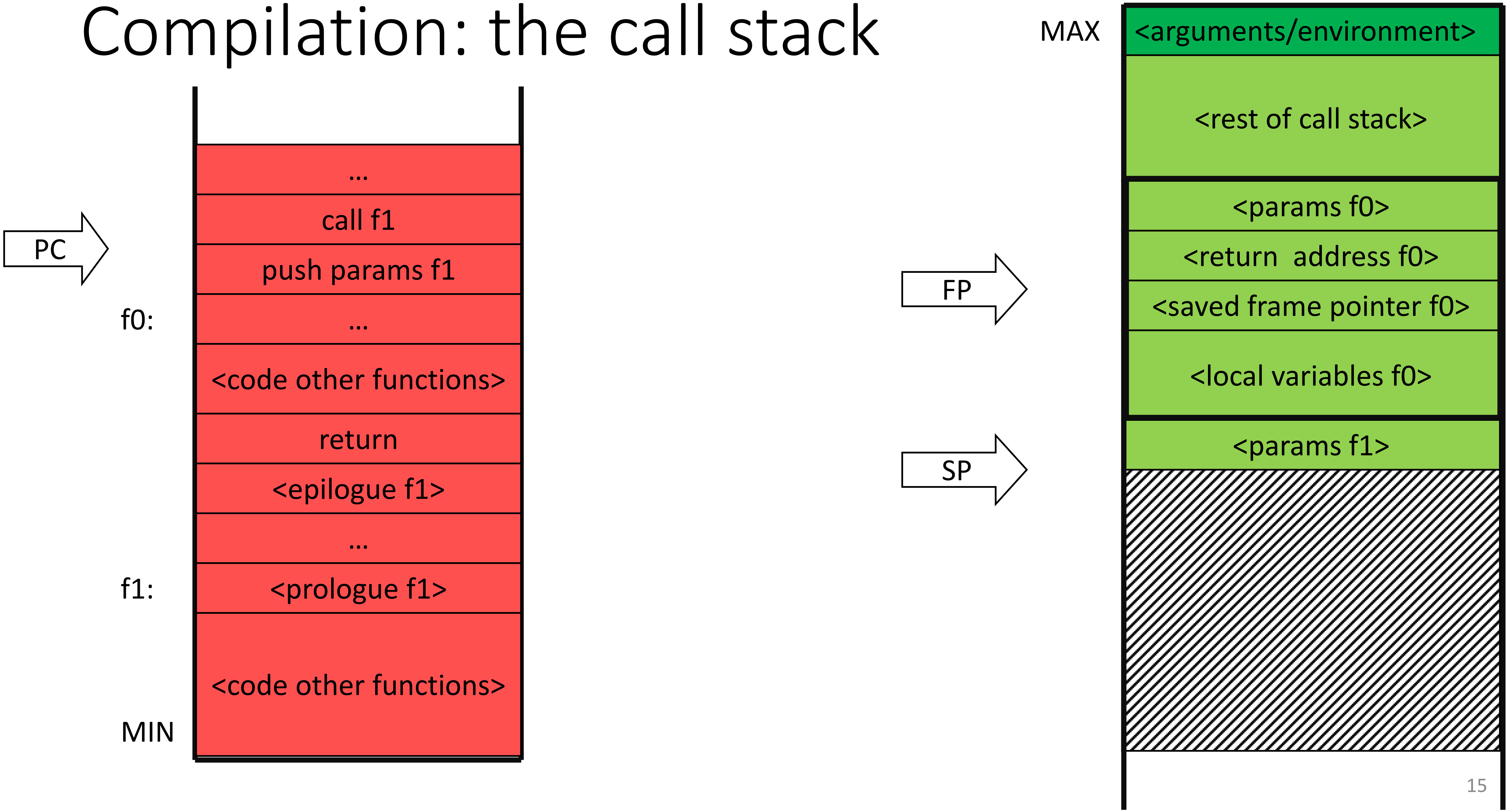
Compilation: the call stack

- The stack 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, …
  - On return of a call, the corresponding stack frame is popped from the stack

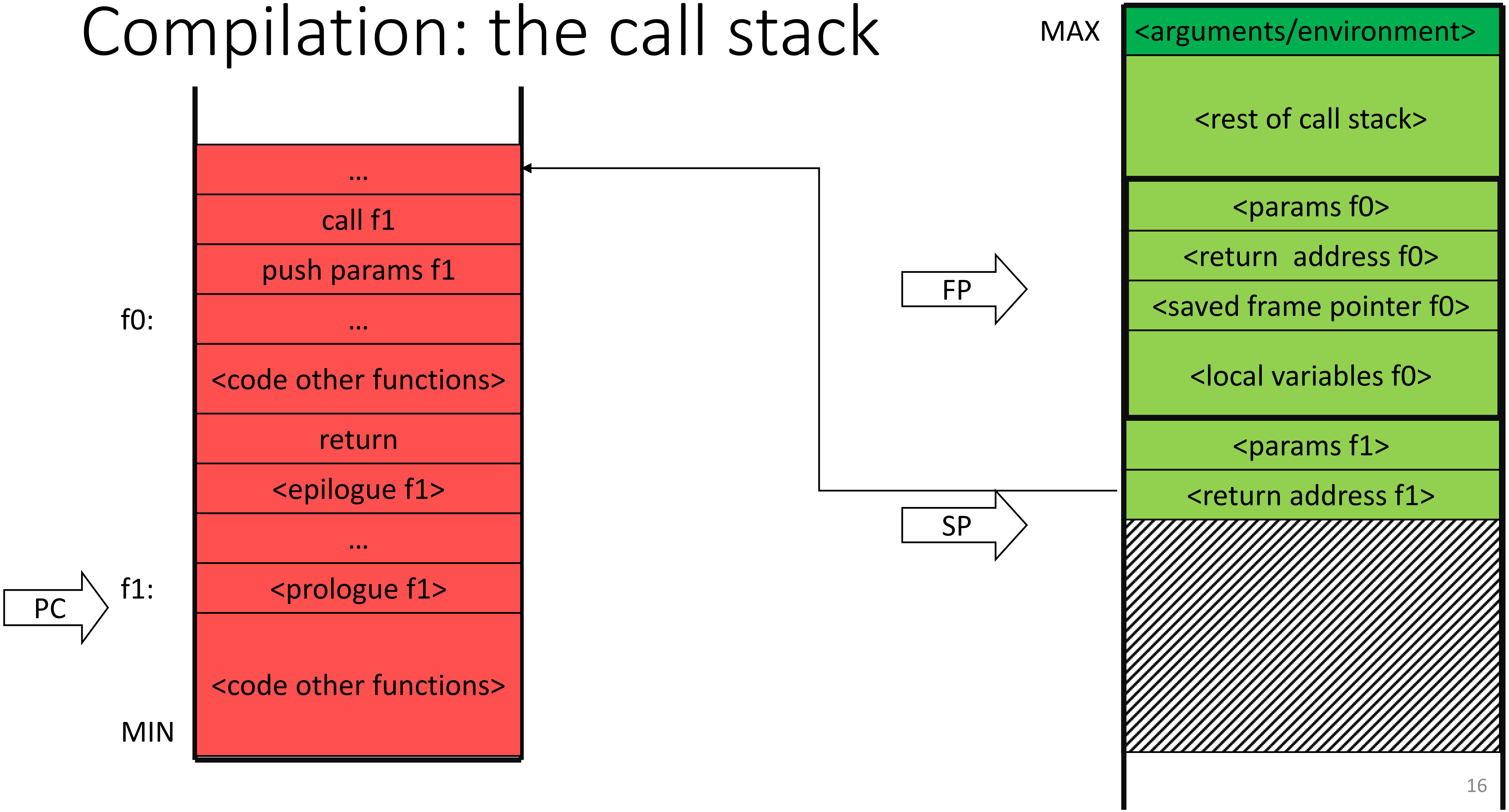
Compilation: the call stack



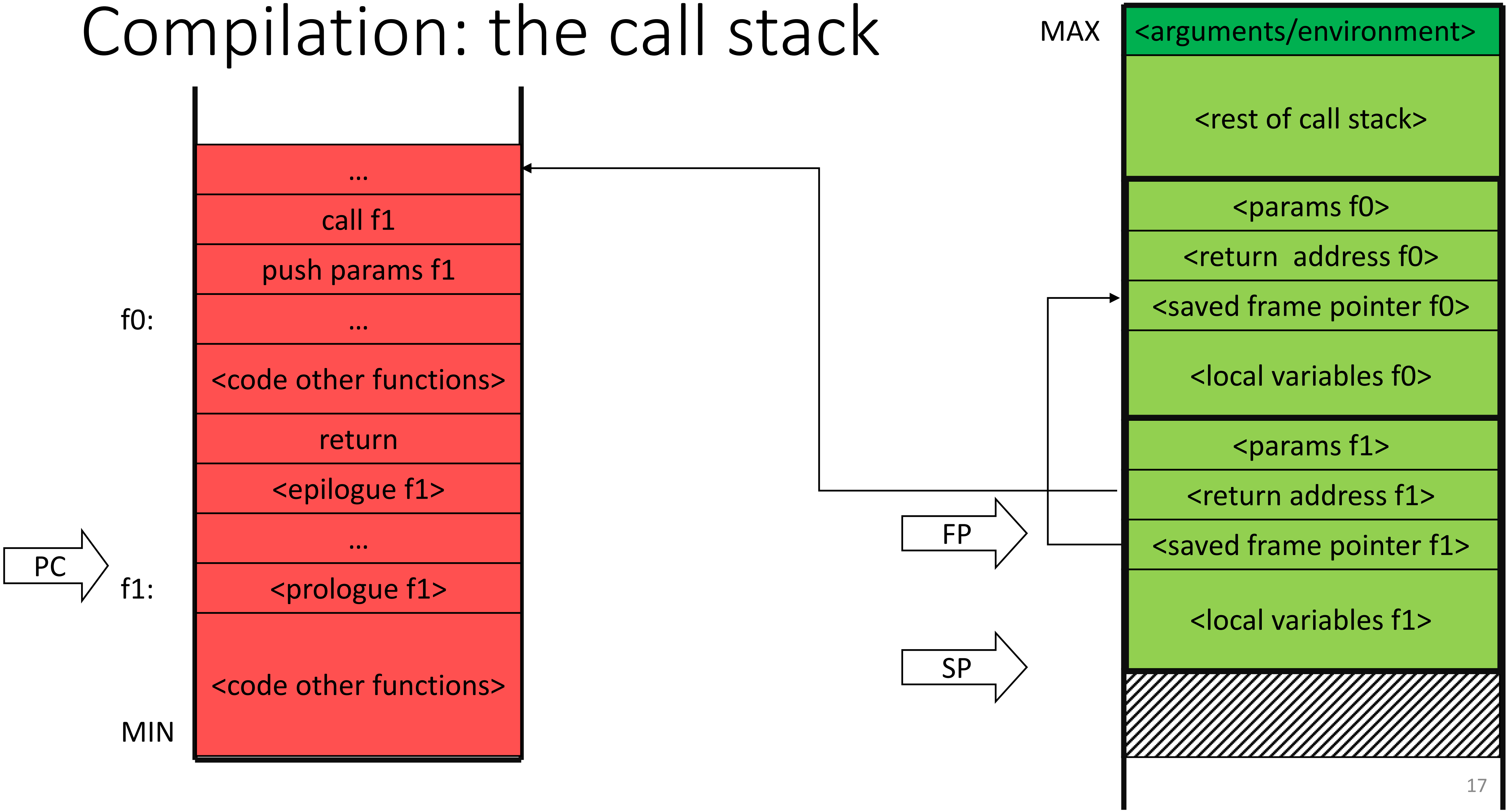
Compilation: the call stack



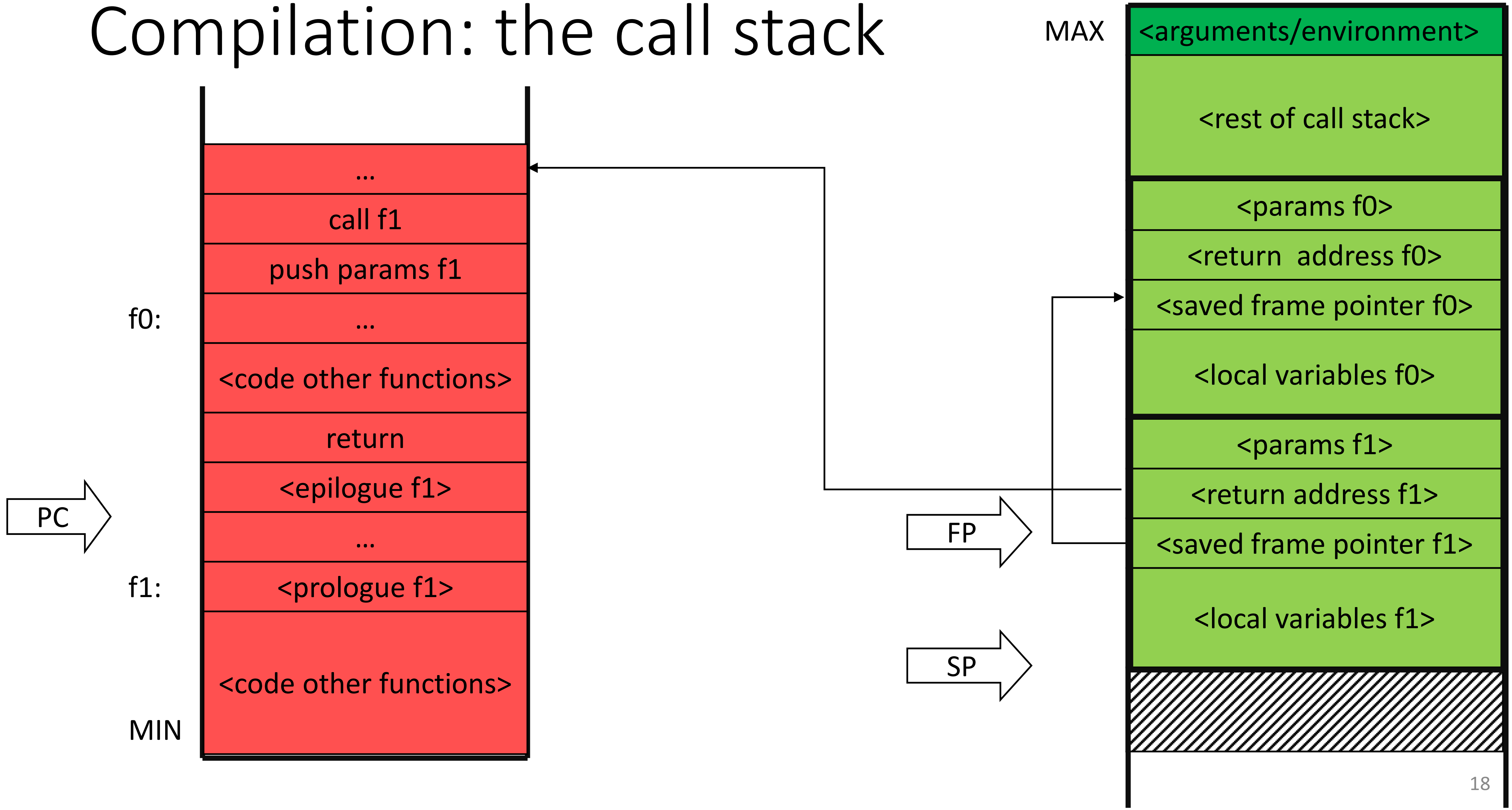
Compilation: the call stack



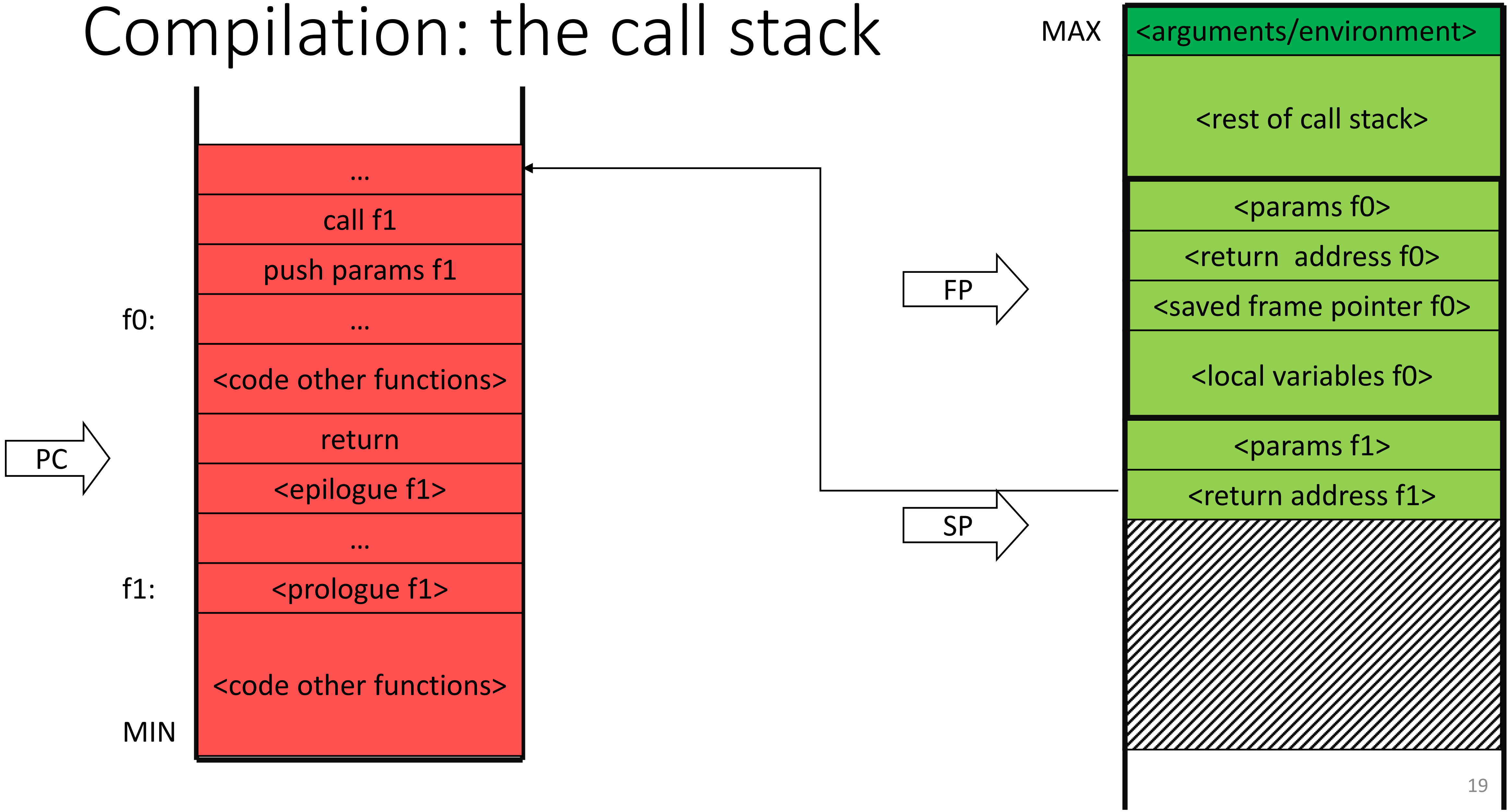
Compilation: the call stack



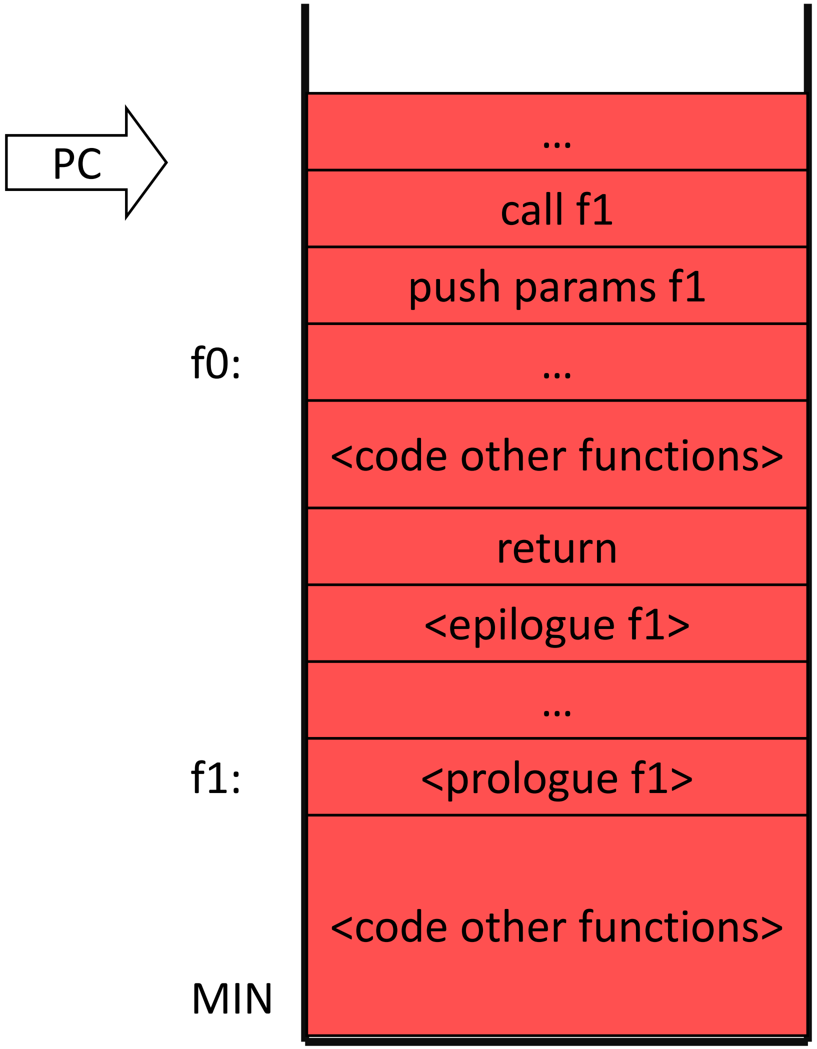
Compilation: the call stack

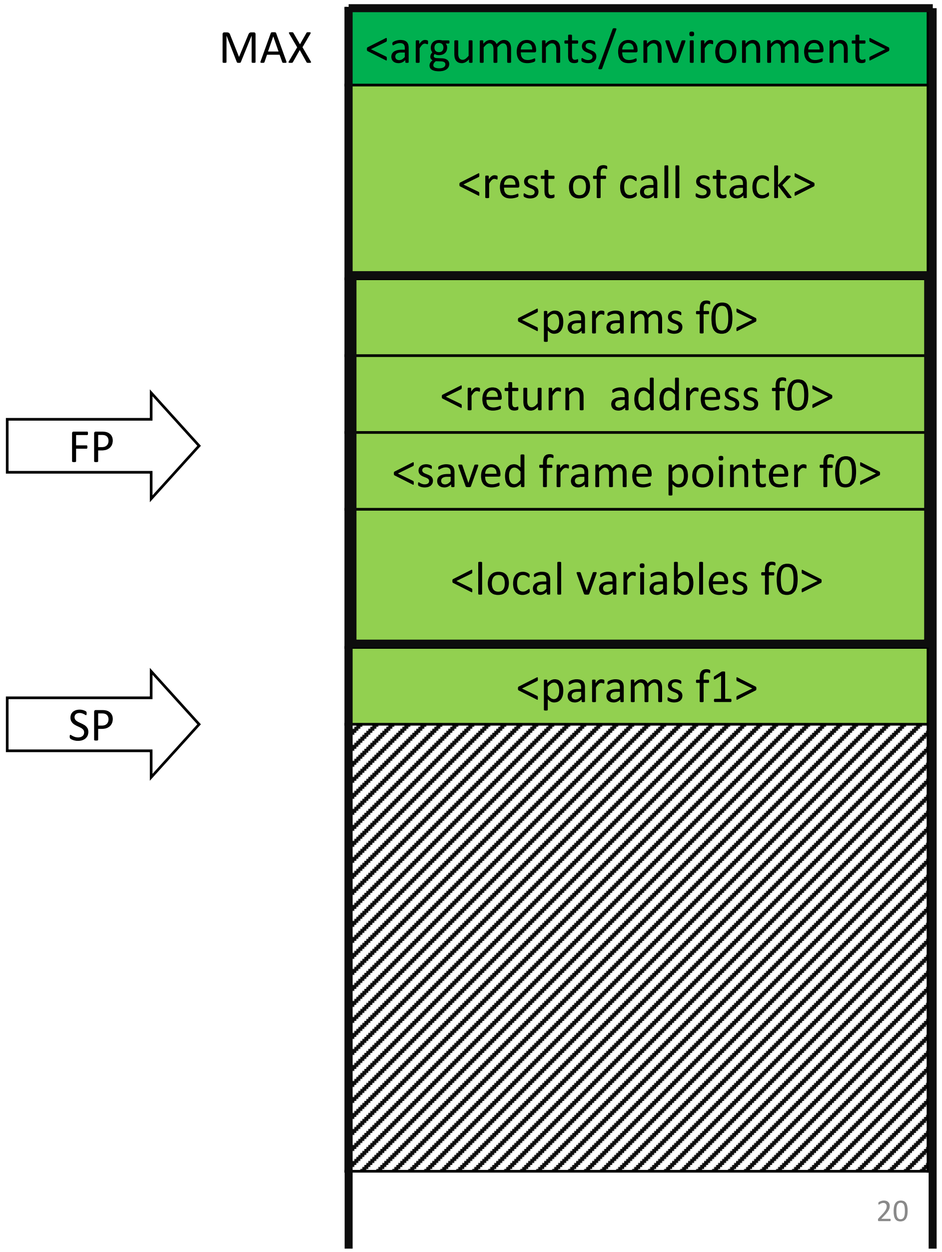


Compilation: the call stack



Compilation: the call stack





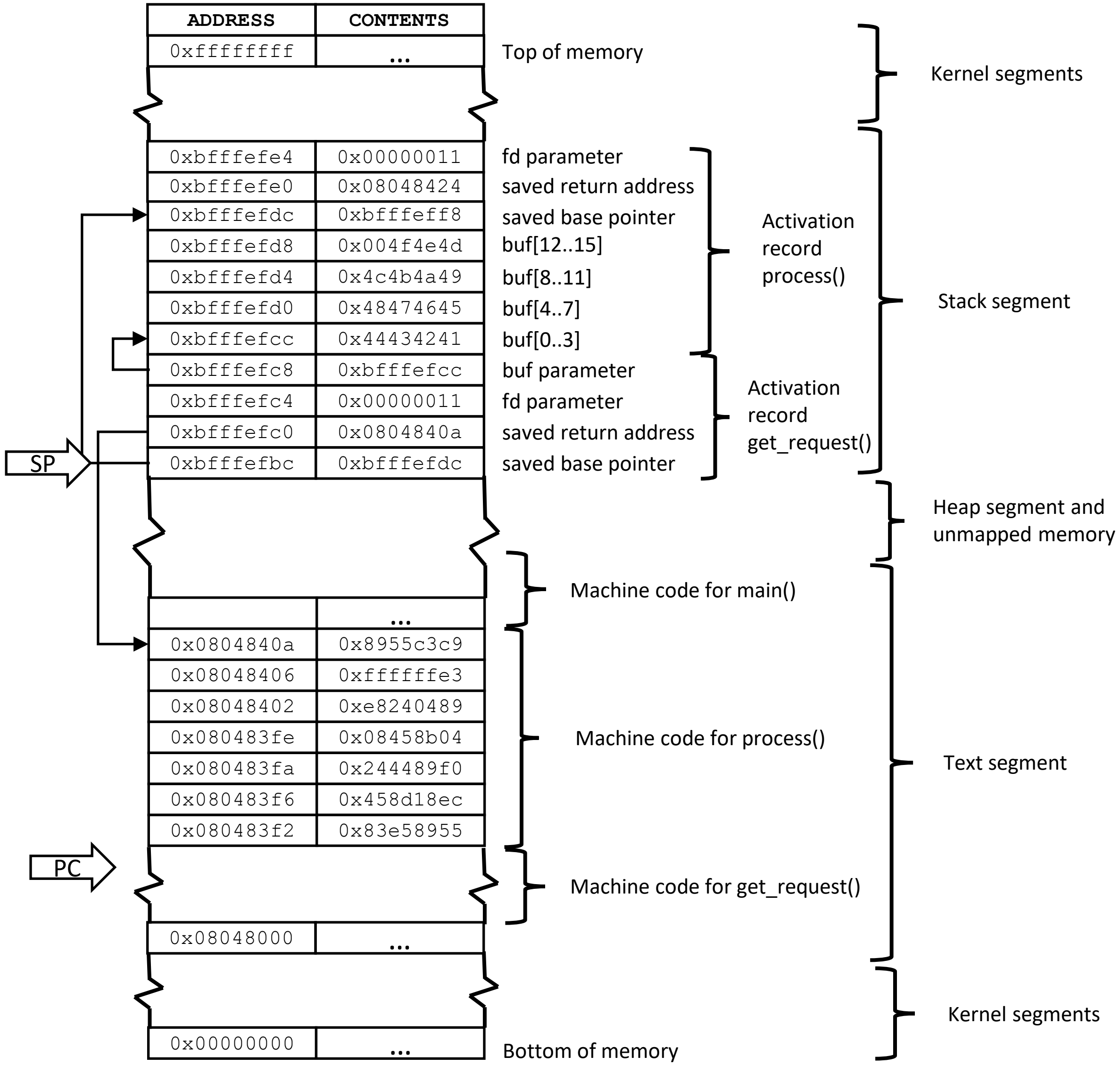
Putting it all together for gcc on 32-bit x86 linux …

```c
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



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

Memory management vulnerabilities

- C-like languages offer mutable variables that can be allocated, deallocated and accessed in a number of ways:
  - Automatic, static or dynamic allocation and deallocation
  - Access through pointers and array indexing
- These memory management and access operations should be used correctly, e.g.:
  - Access arrays within bounds
  - Do not access memory after it has been deallocated
- For performance, compilers do not detect invalid memory accesses
  - Instead, behavior of the program becomes **undefined**
  - A program that can perform such an invalid access has a **memory management vulnerability**

Overview

- System model
- Attack scenarios
- Mitigating attacks
- Avoiding vulnerabilities
- Conclusions

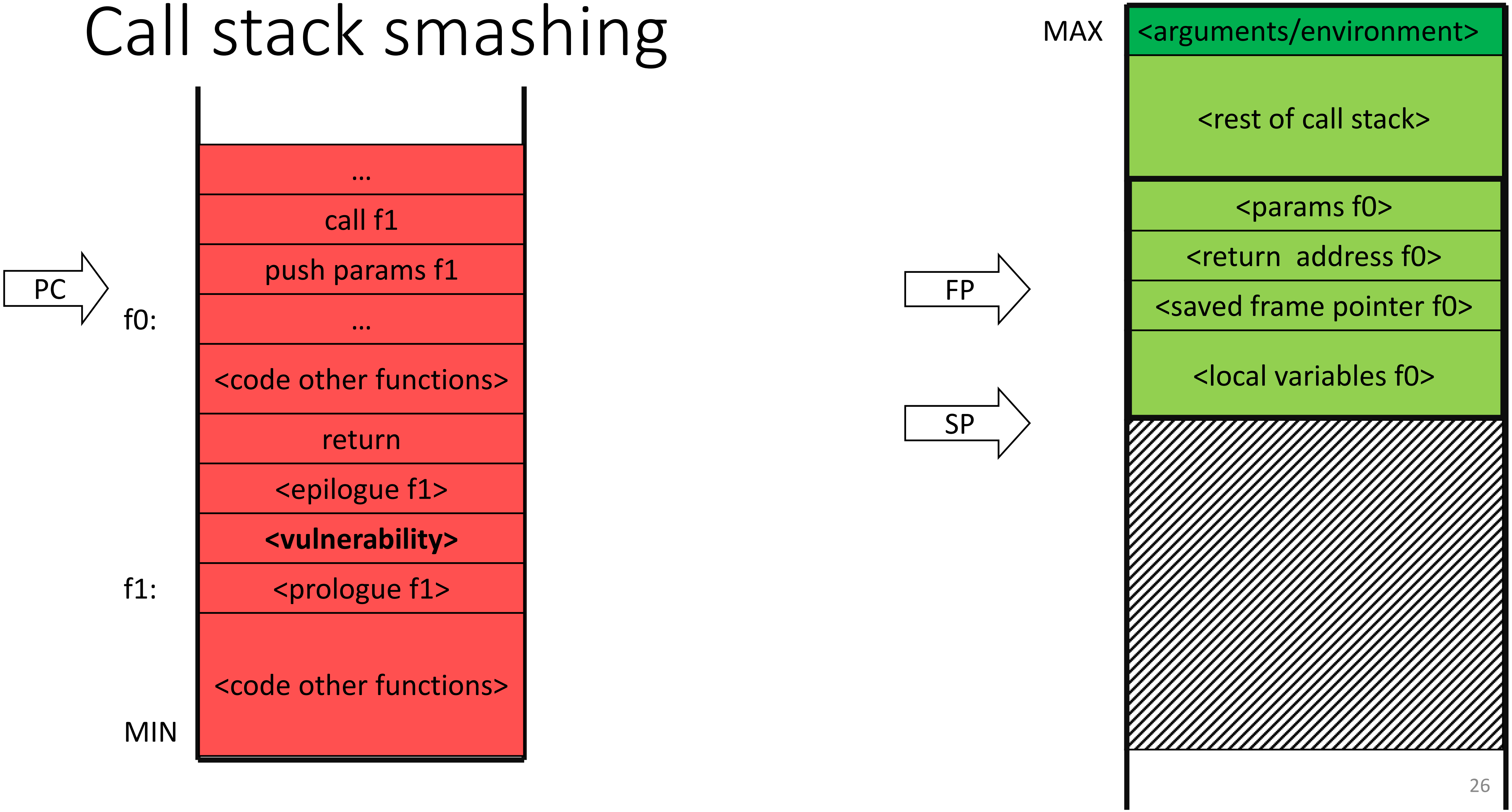
Introduction: attack scenarios

- The key idea underlying exploitation of programs with memory management vulnerabilities is:
  - Feed the program input that triggers the vulnerability, i.e., causes an invalid memory access
    - Hence, further behavior is undefined according to the language specification
  - Use knowledge and understanding of this particular implementation of the language to make sure that what happens is useful to the attacker
    - Leak data
    - Tamper with data or the execution of the program

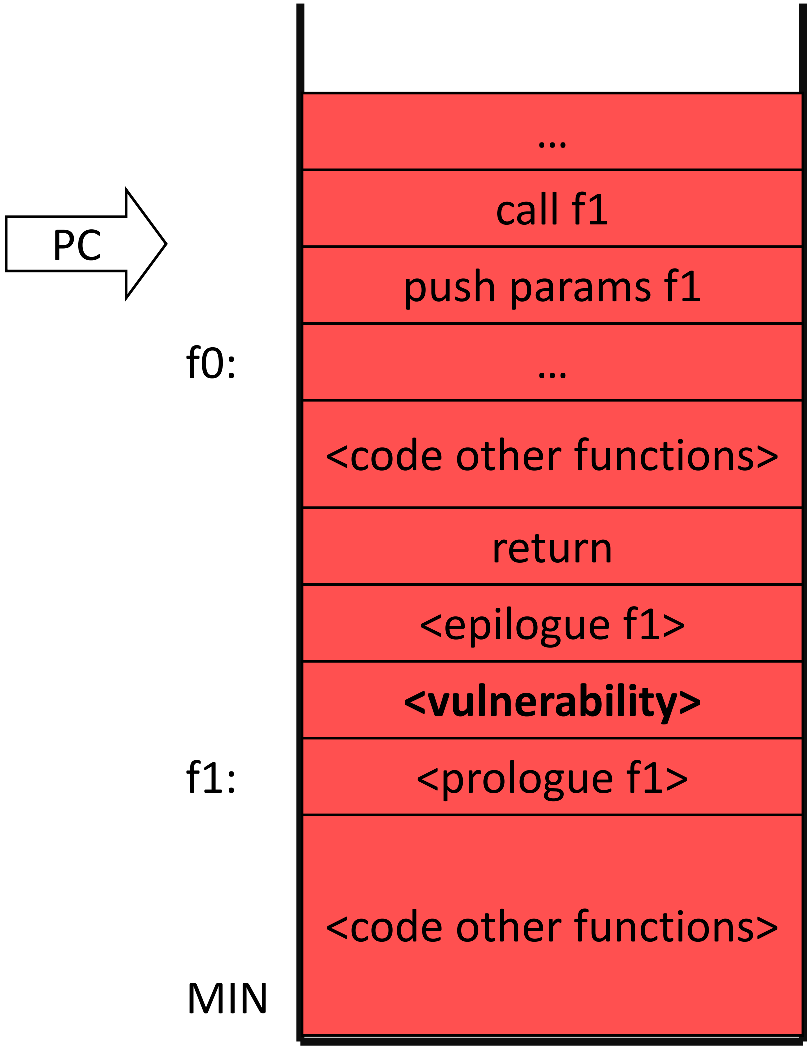
Attack scenario 1: call stack smashing

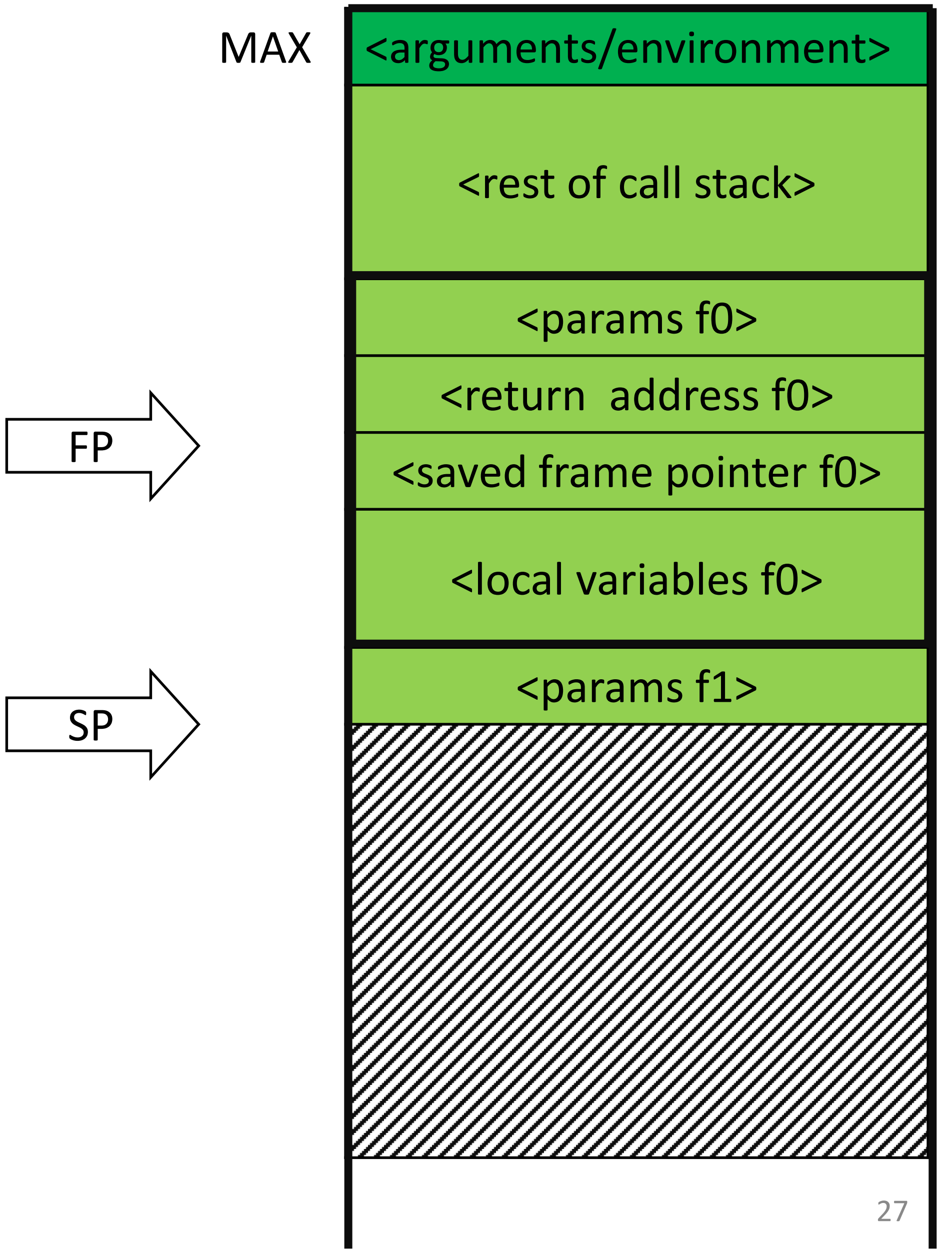
- The stack used at run time to track function calls and returns
  - Per call, *an activation record* or *stack frame* is pushed on the stack, containing:
    - Actual parameters, return address, automatically allocated local variables, …
  - On return of a call, the corresponding stack frame is popped from the stack
- As a consequence, if a local array (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**)

Call stack smashing

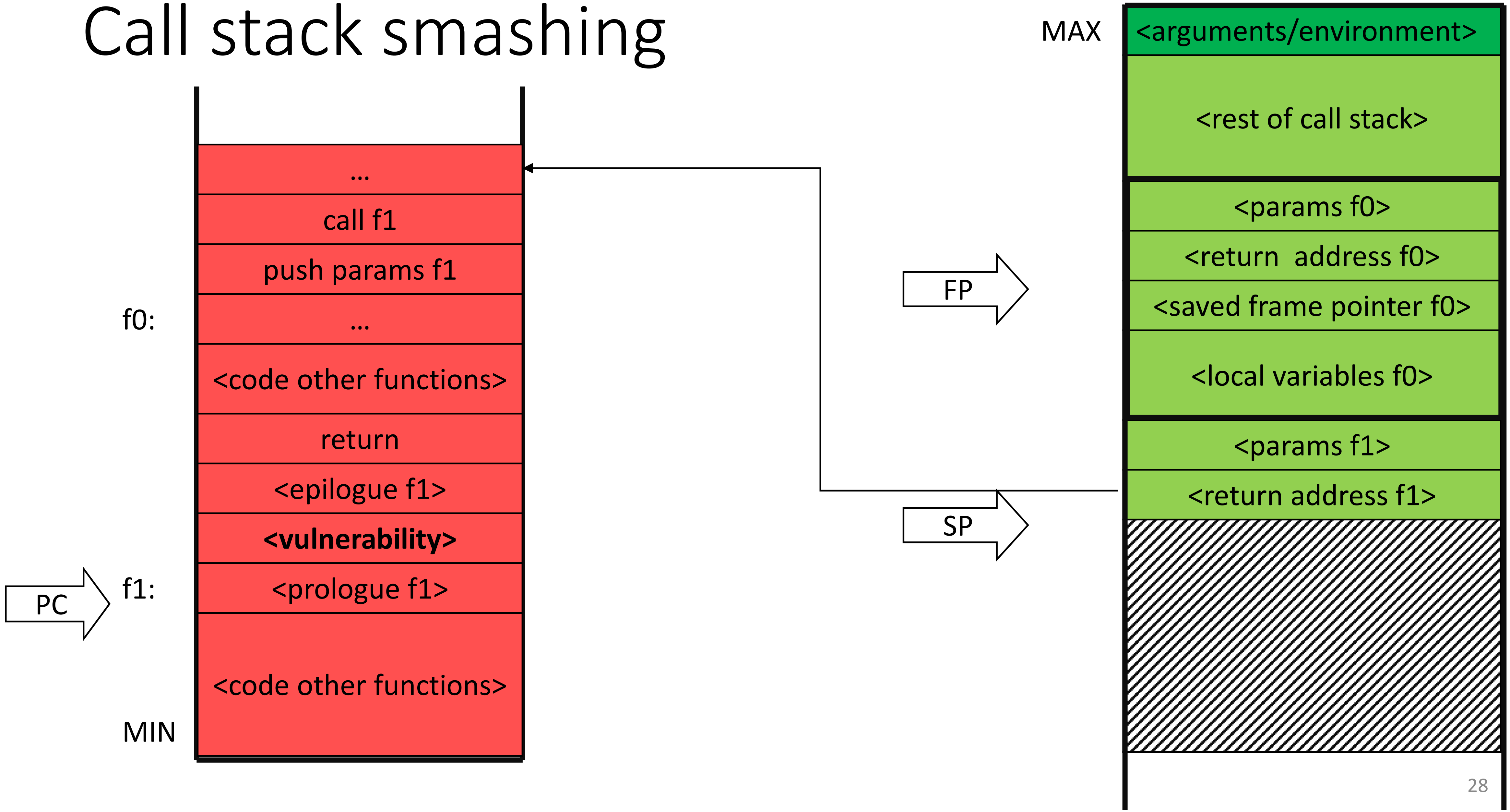


Call stack smashing

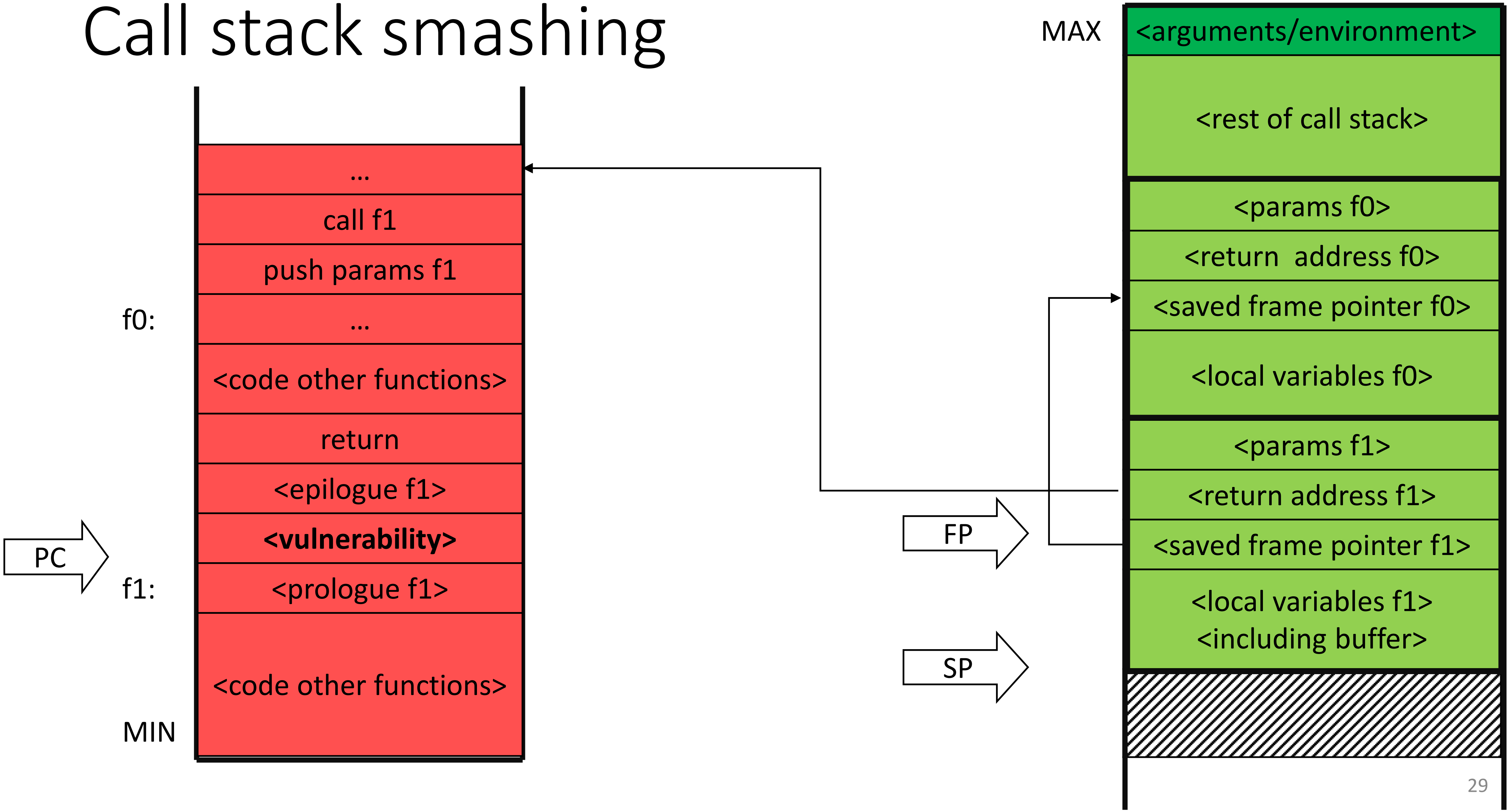




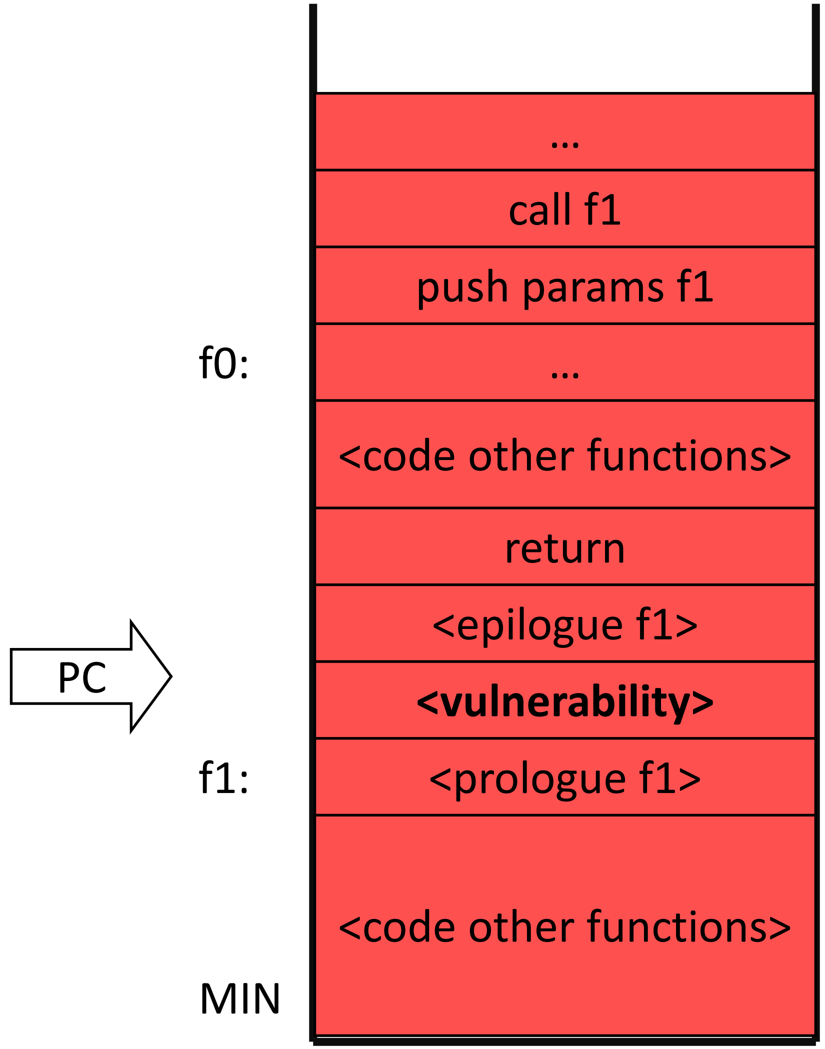
Call stack smashing

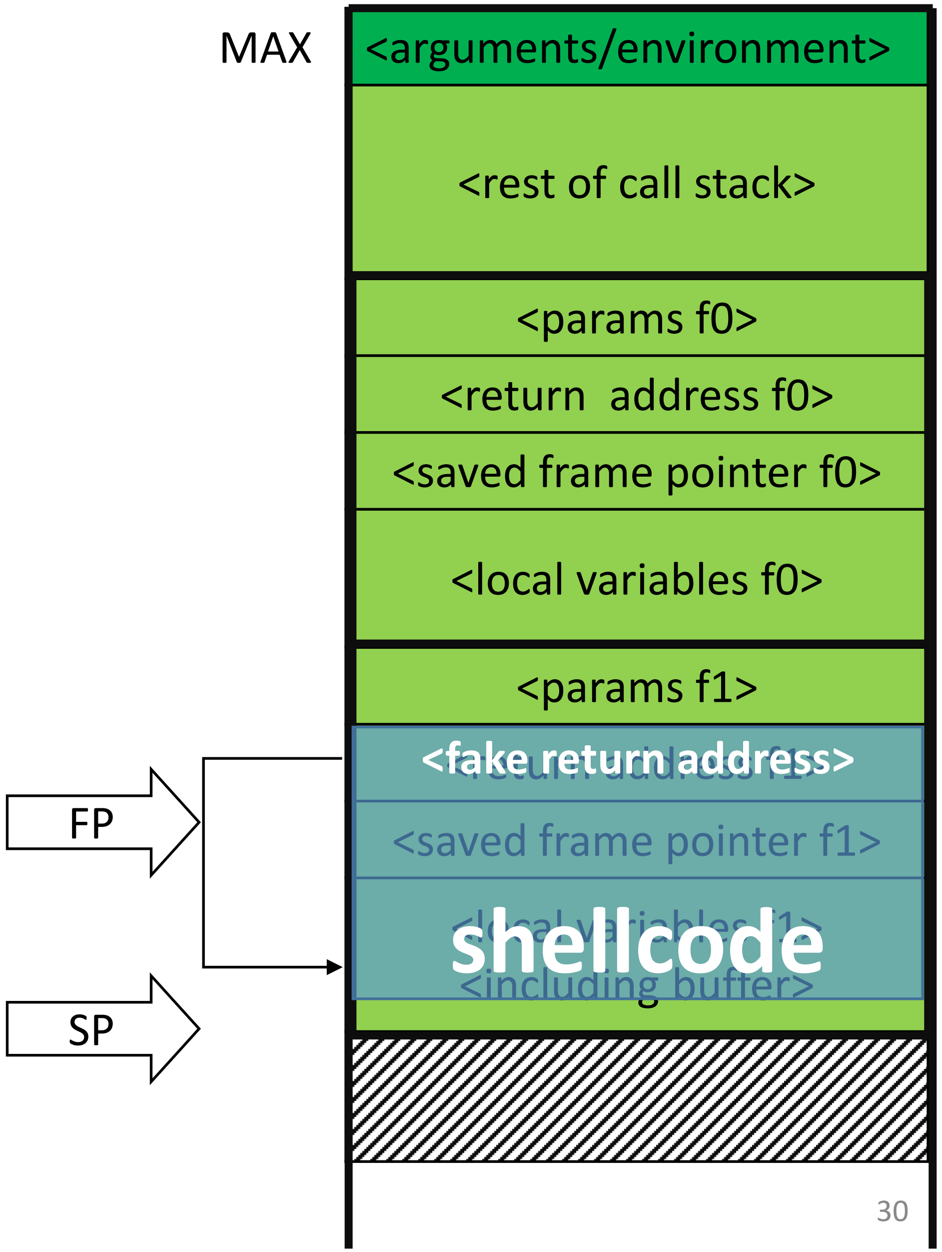


Call stack smashing

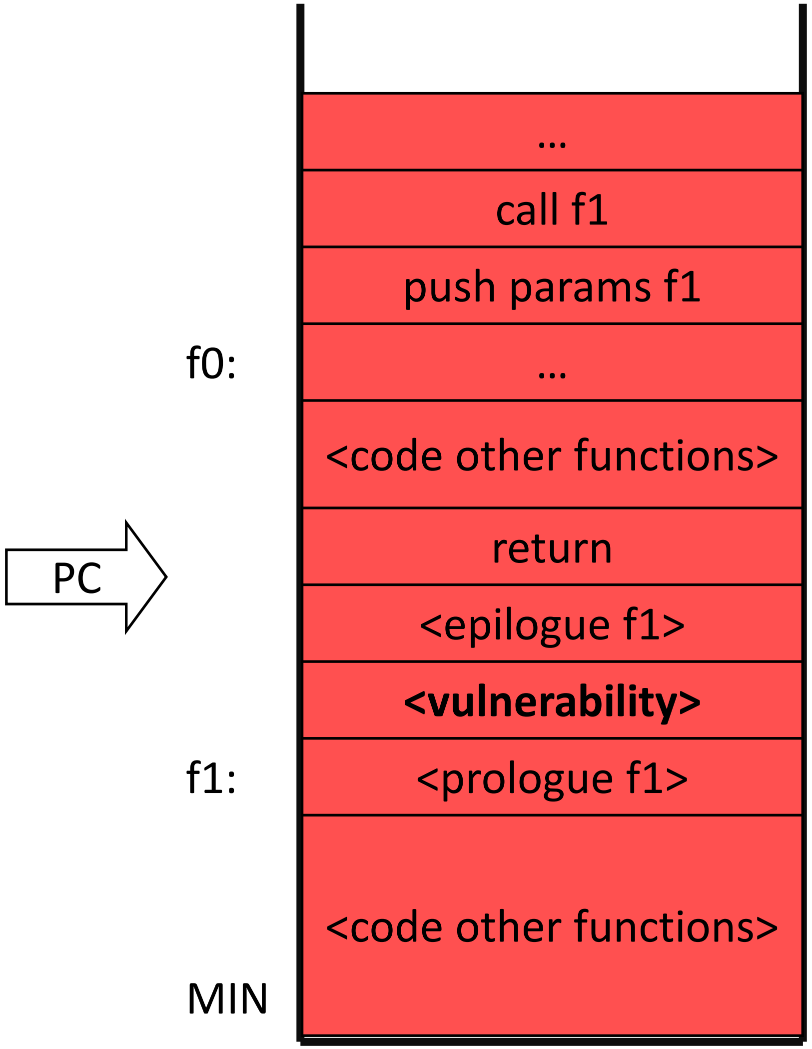


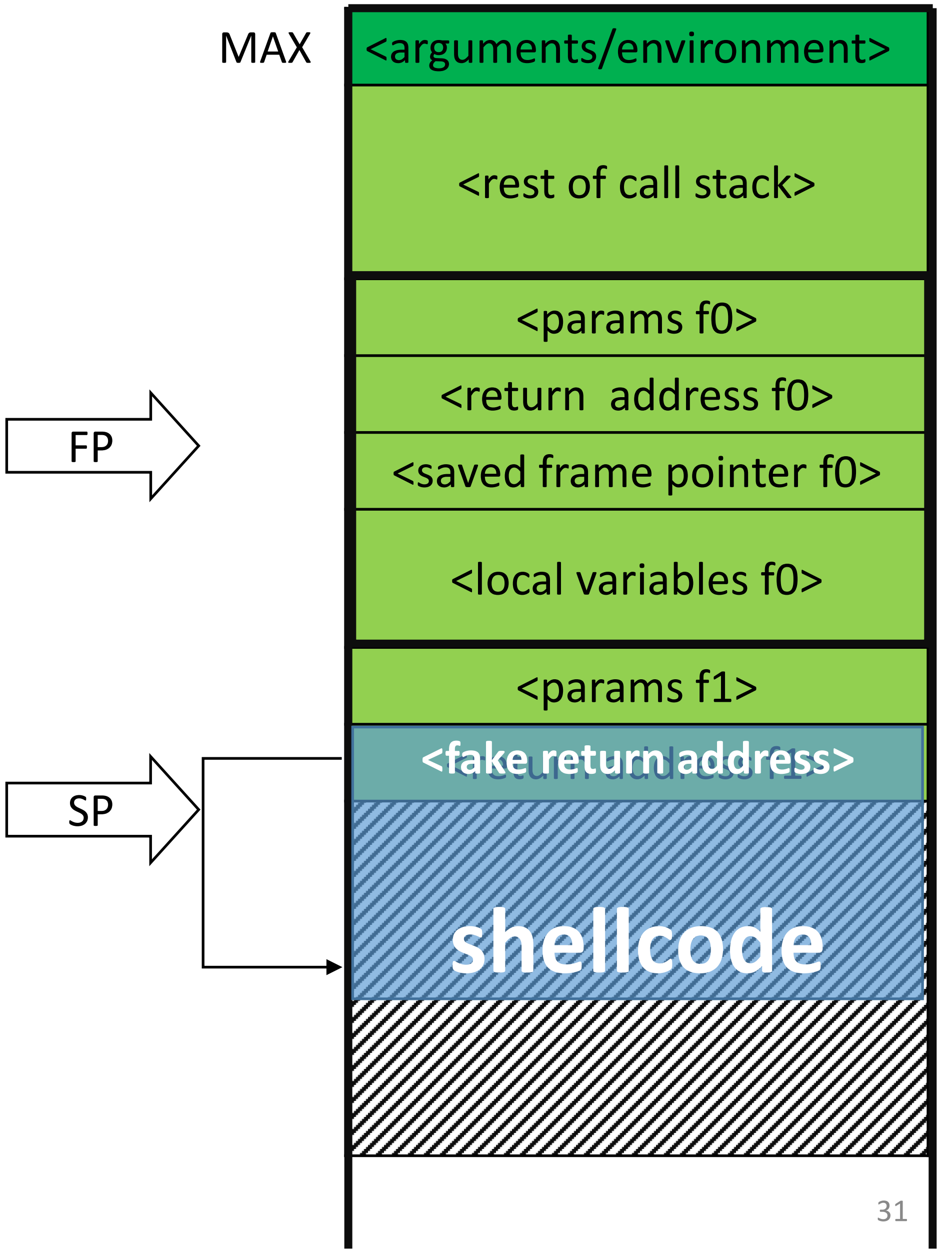
Call stack smashing



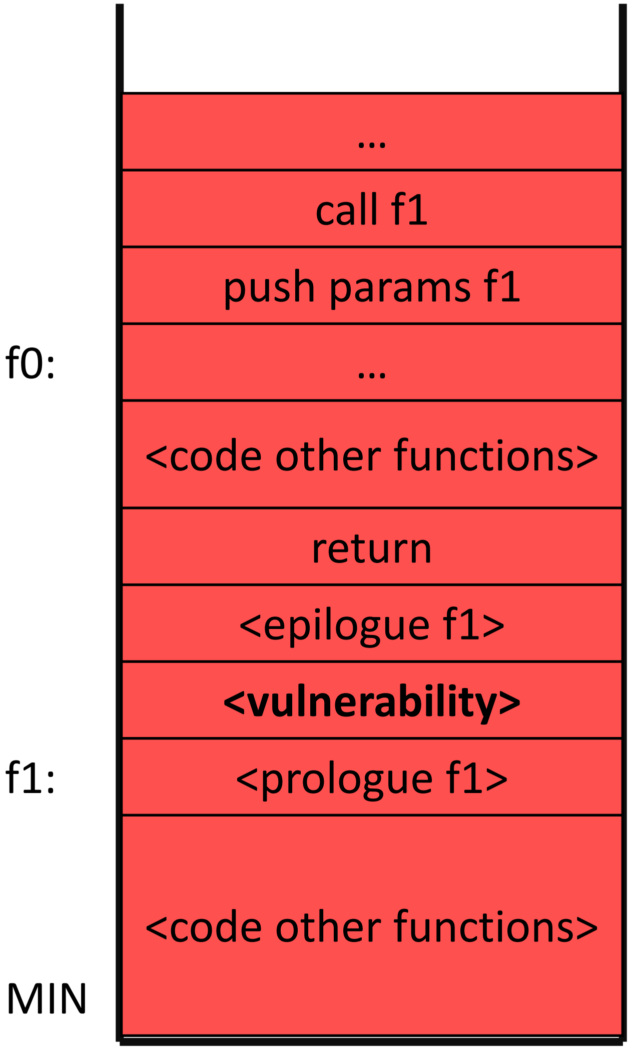


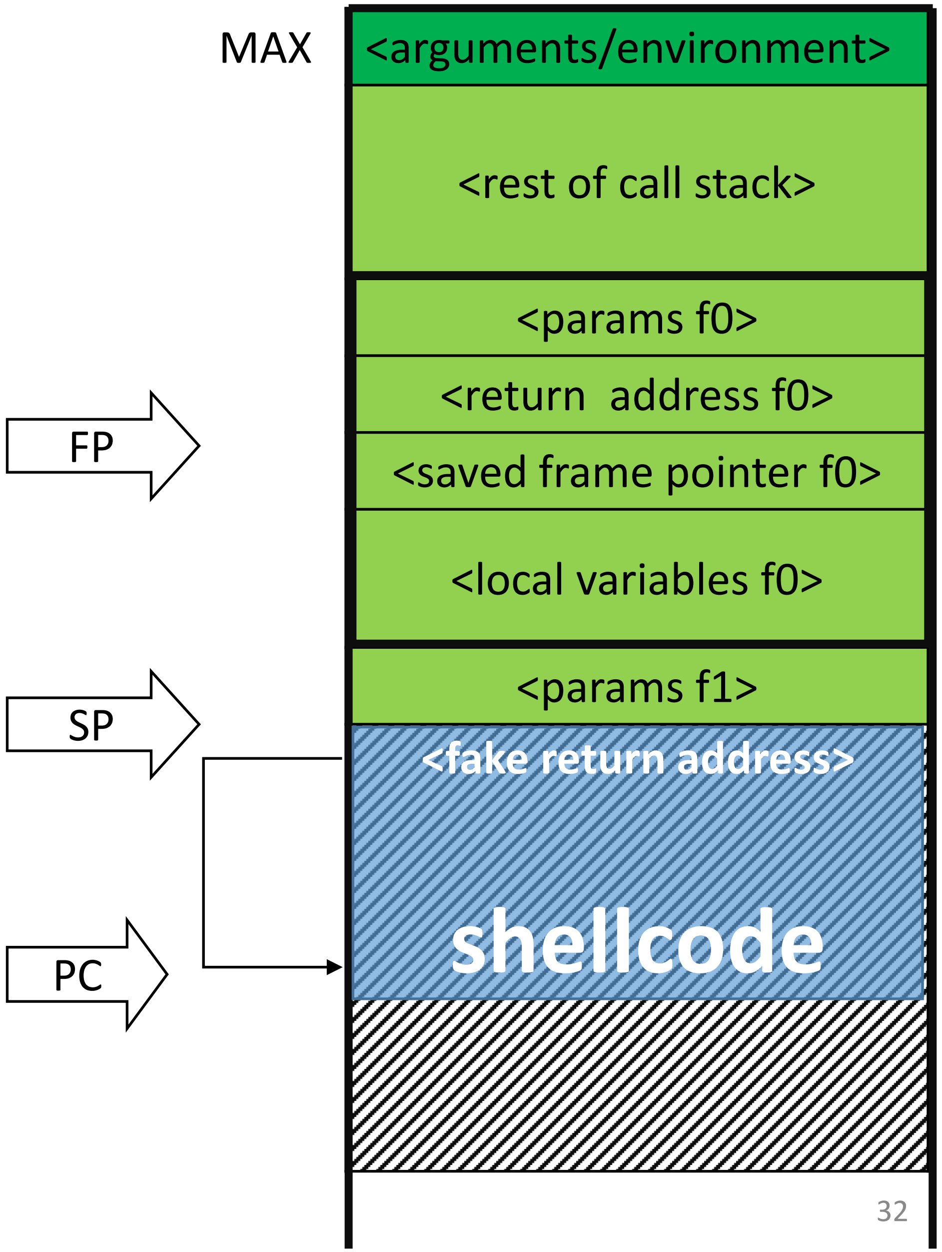
Call stack smashing





Call stack smashing





A concrete attack

- Very simple shell code:

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

- Keeping in mind little-endianness, this shell code appears as the 4-byte word `0xfeeb2ecd`

A concrete attack

- Vulnerable code:

```
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" );
}
```

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'
```

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'

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	0x66 2f2f3a	; tmp continues	':' '/' '/' 'f'
0x0012ff40	0x656c6966	; tmp array:	'f' 'i' 'l' 'e'

Call stack smashing

- 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

Attack scenario 2: overwriting a function pointer 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
- We look at an example where we overwrite a function pointer

Concrete attack: 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" );
}
```

This defines the type: vulnerable

This struct can be allocated anywhere, most likely on the heap

Concrete attack: overwriting a function pointer

	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".

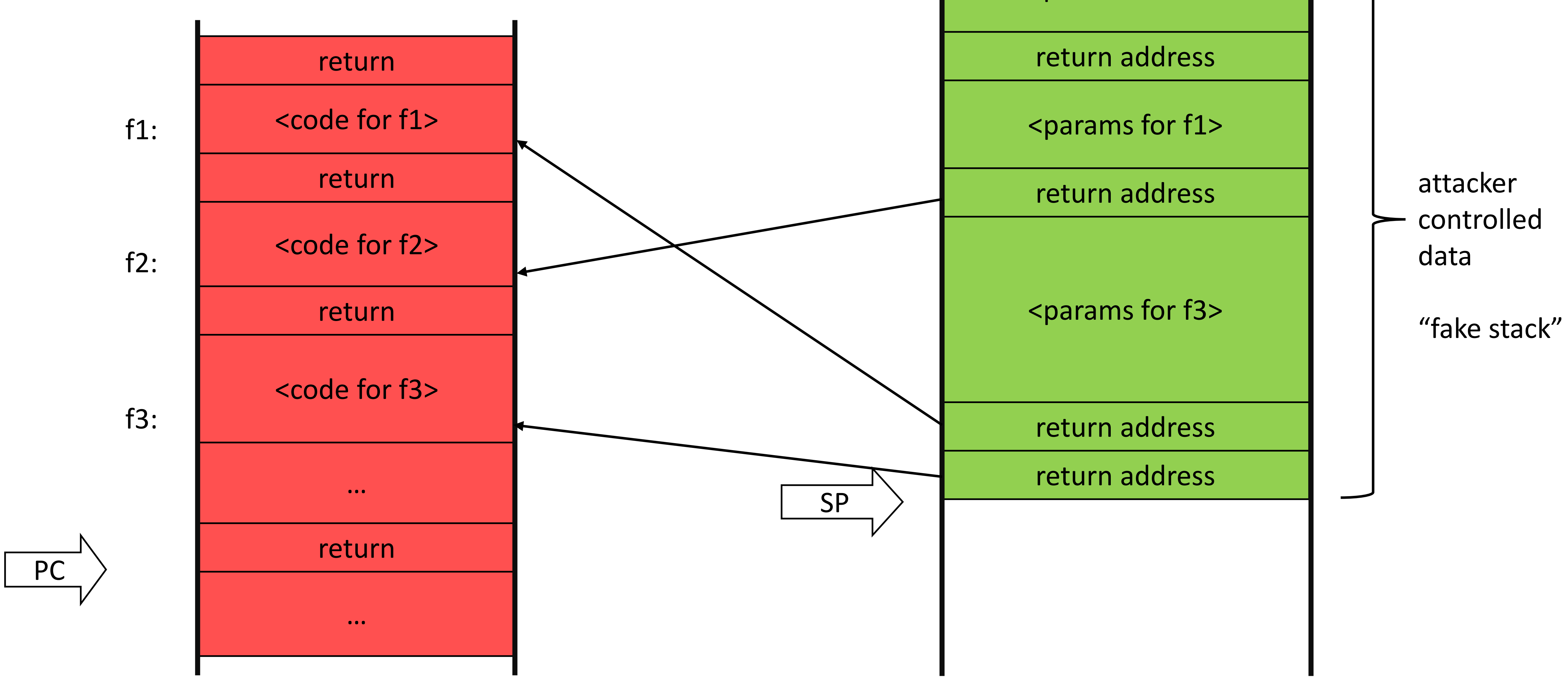
	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0xfeeb2ecd	0x11111111	0x11111111	0x11111111	0x00353068

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

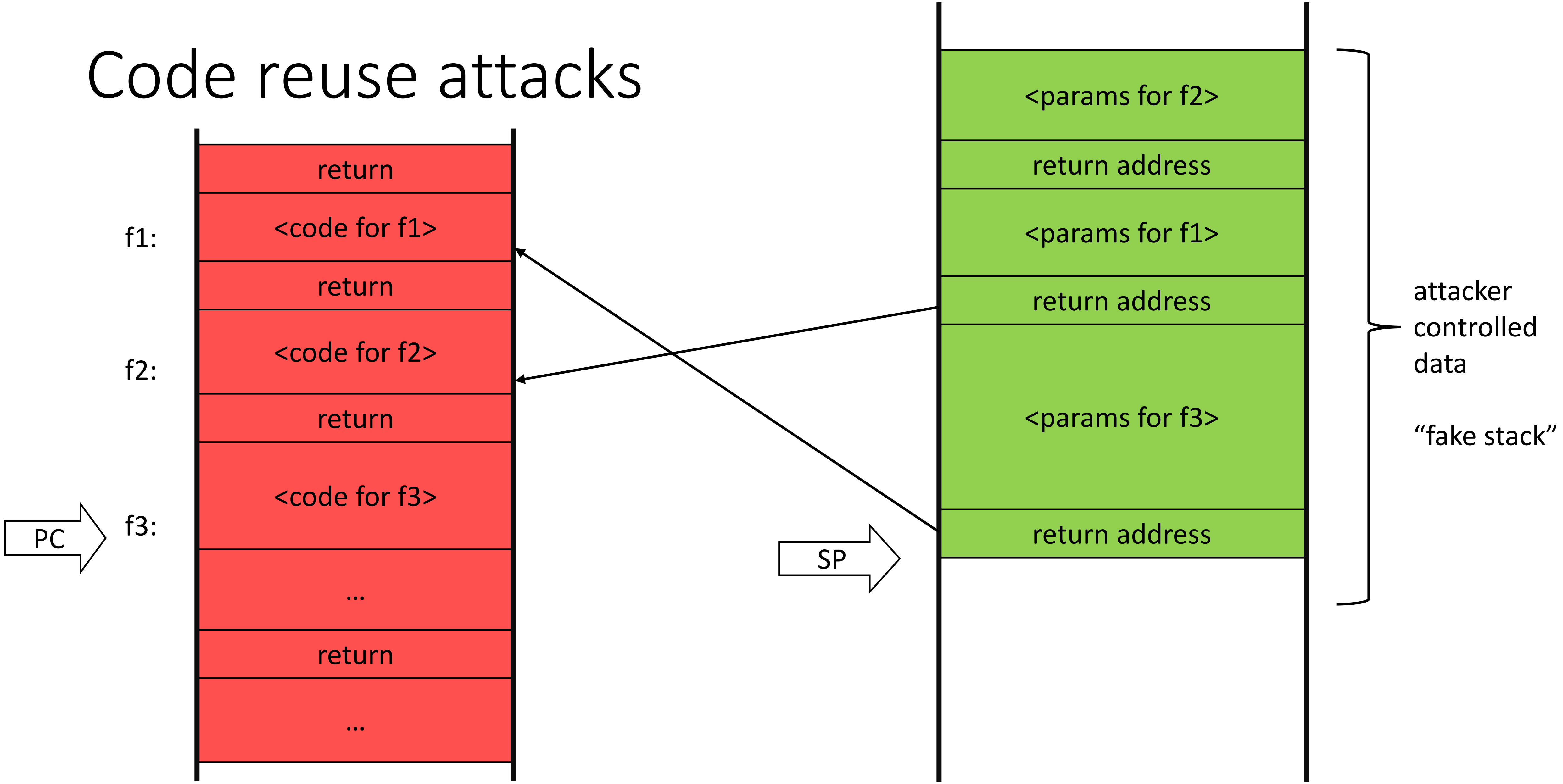
Attack scenario 3: code reuse attacks

- *Direct code injection*, where an attacker injects code as data is not always feasible
  - E.g. When certain countermeasures are active
- *Indirect code injection* or *code reuse* attacks will control execution of the program by reusing fractions of the existing code
- The crux of the attack is to find a way to execute (a chain of) code fractions under the control of the attacker
  - One way of achieving this is by controlling the stack pointer

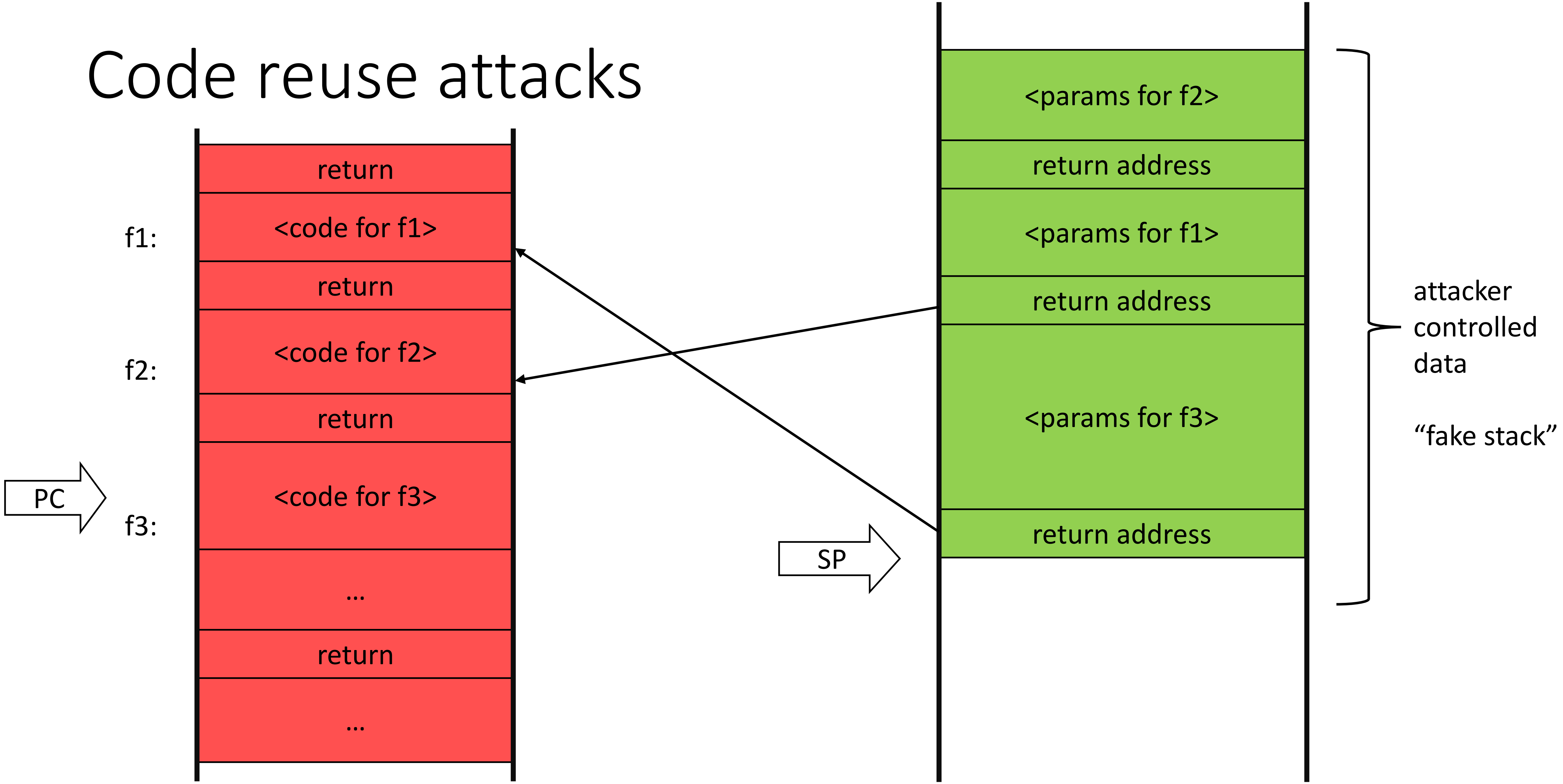
Code reuse attacks



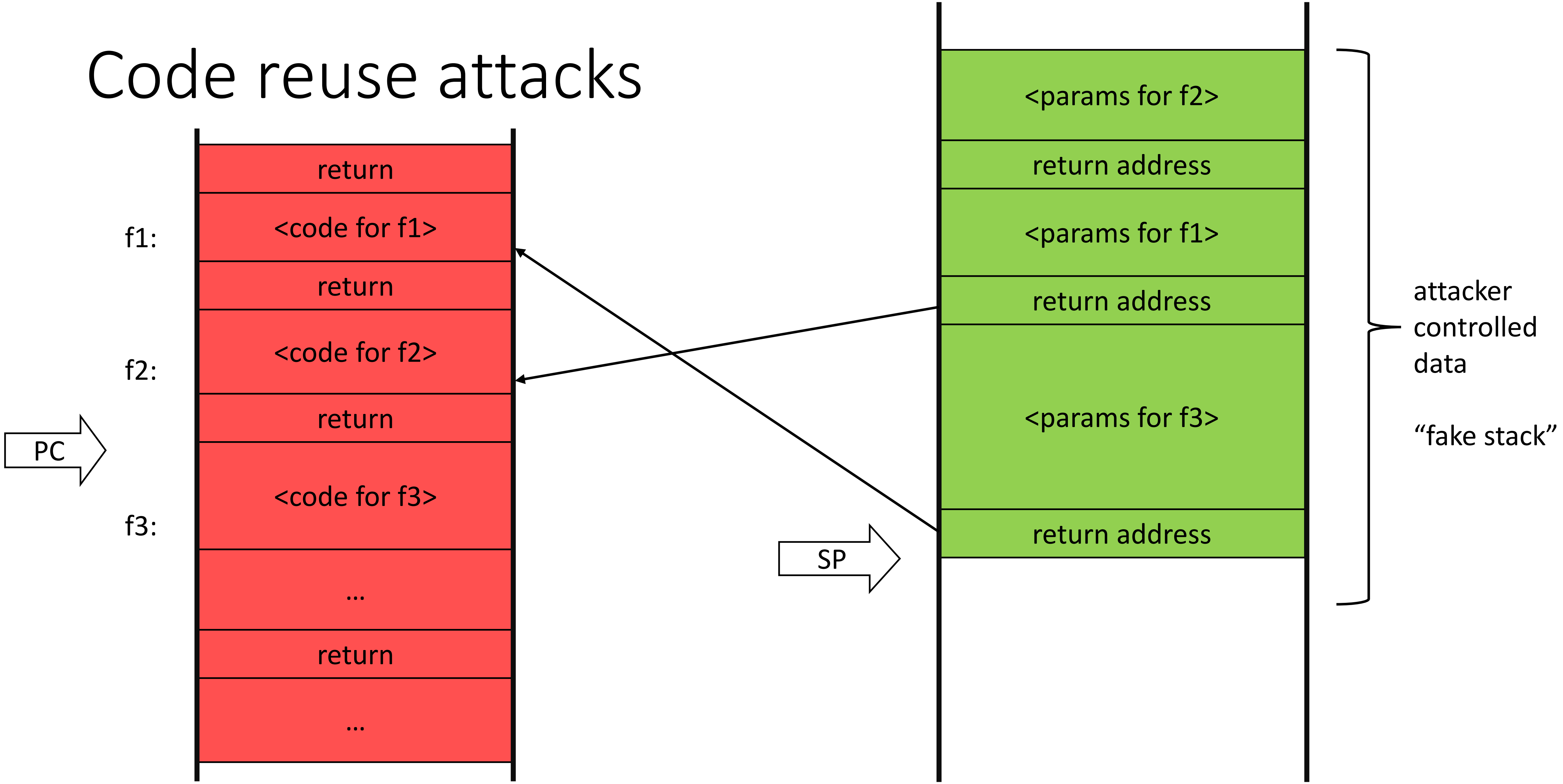
Code reuse attacks



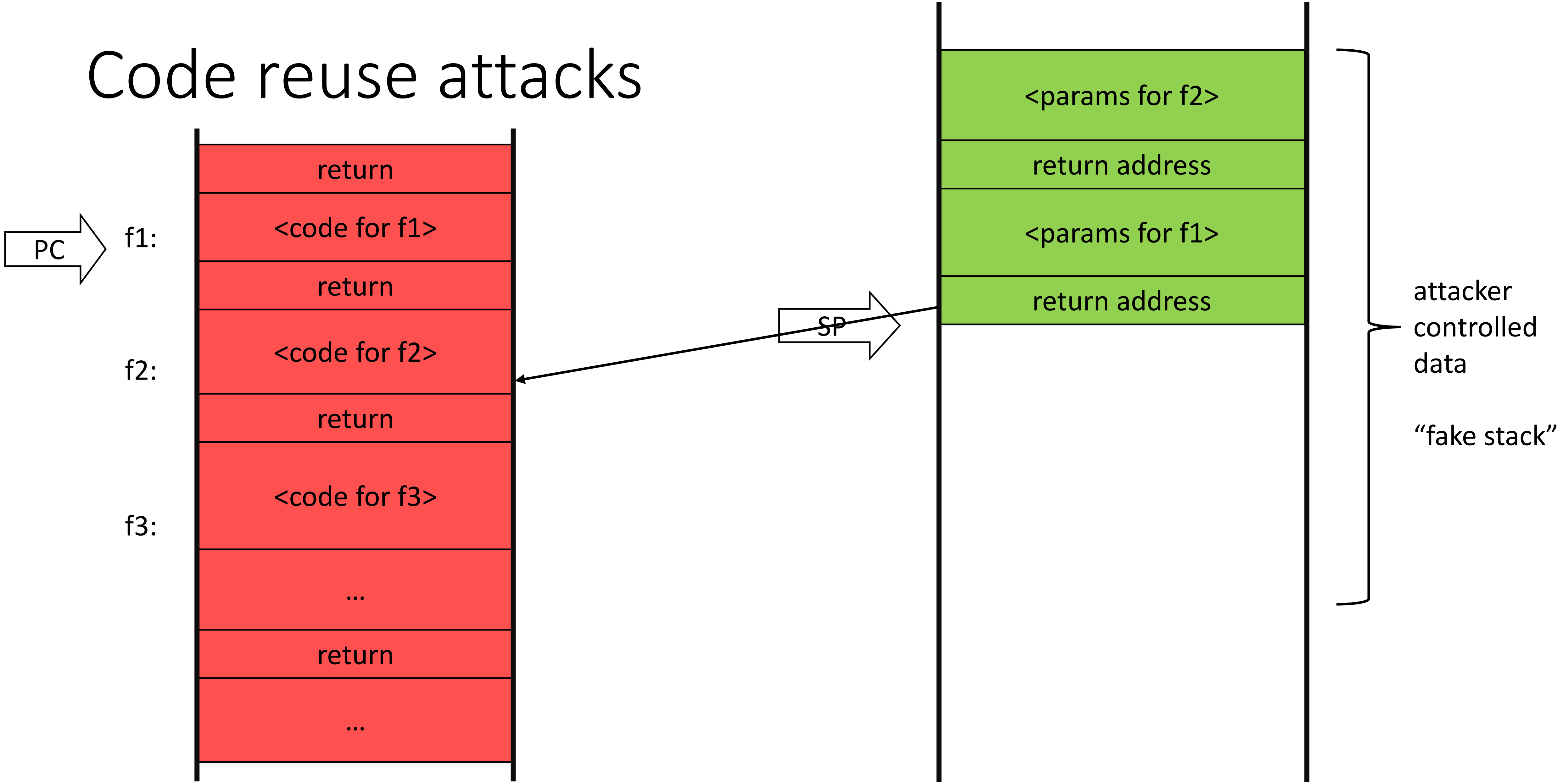
Code reuse attacks



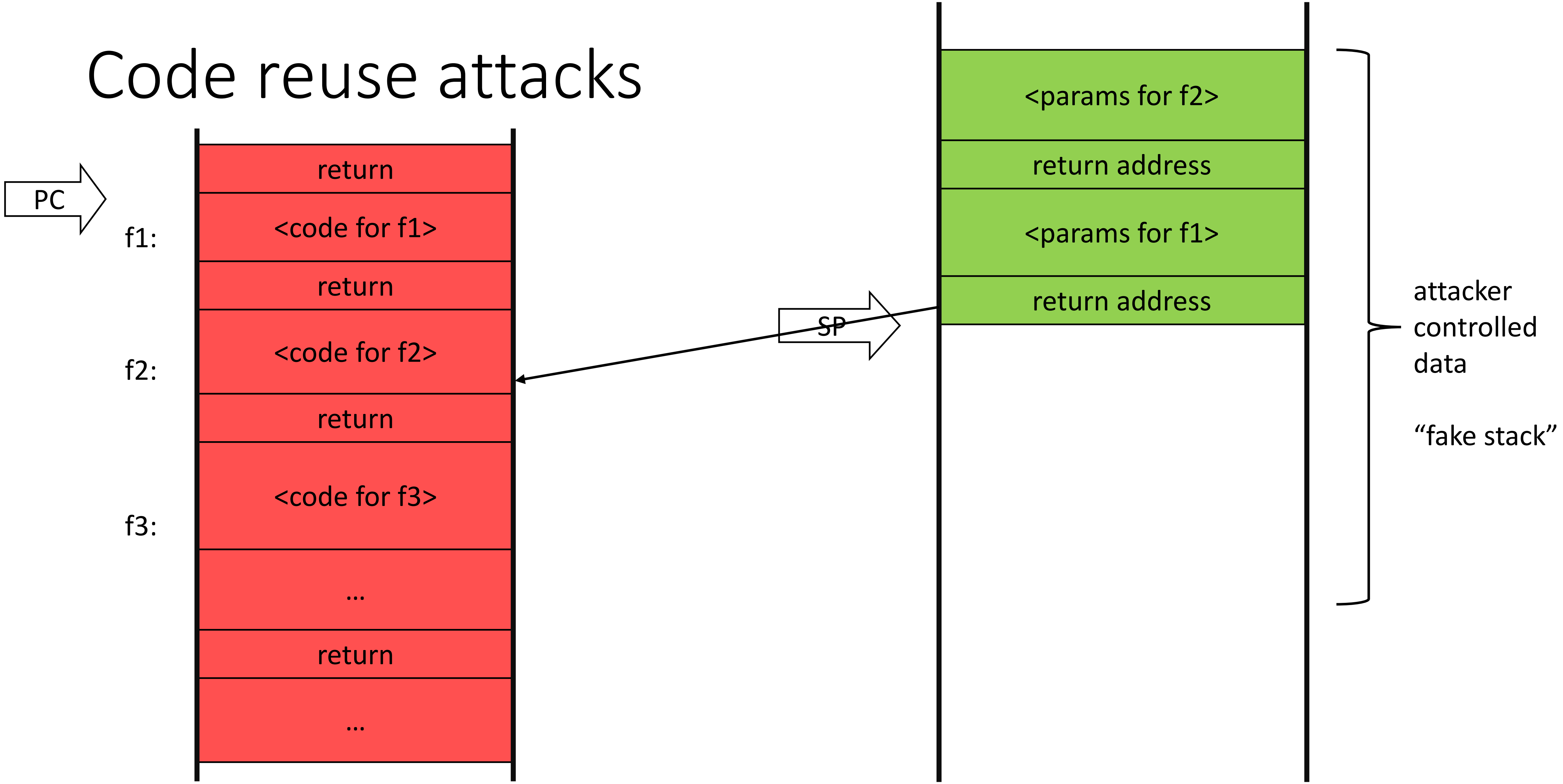
Code reuse attacks



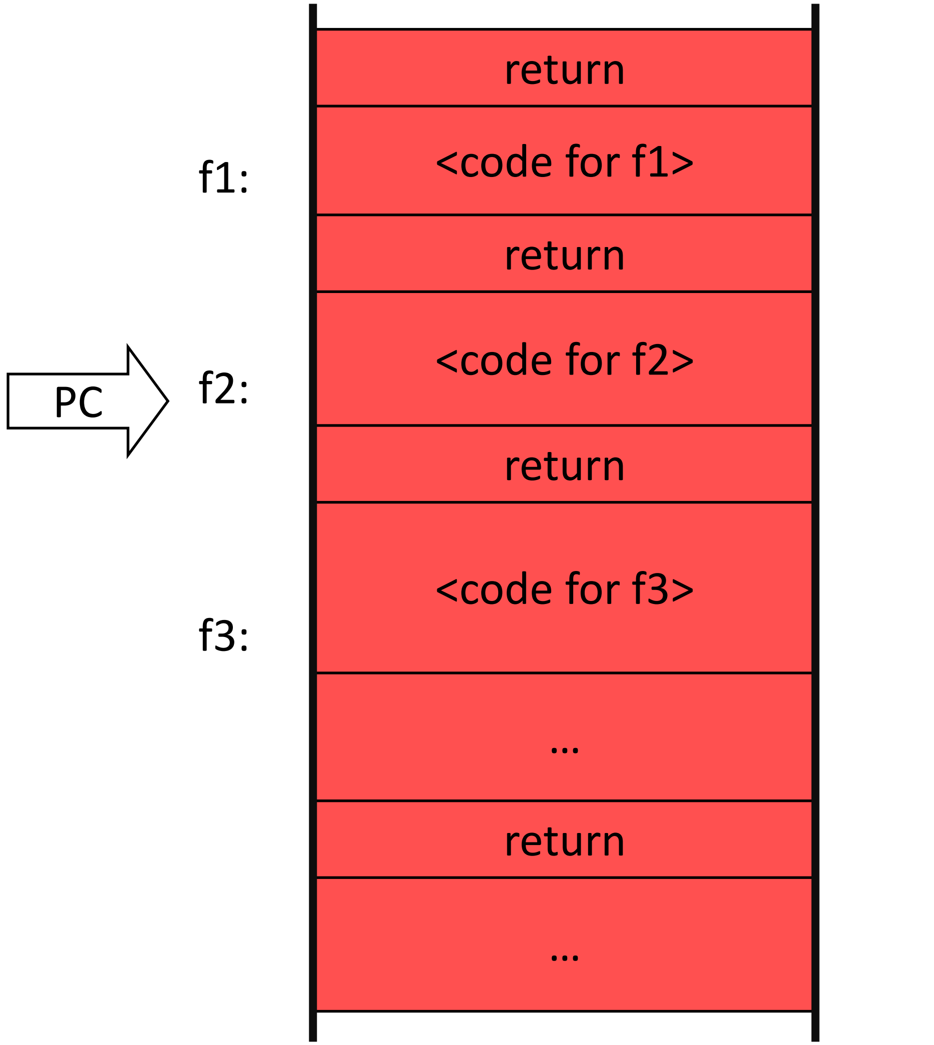
Code reuse attacks

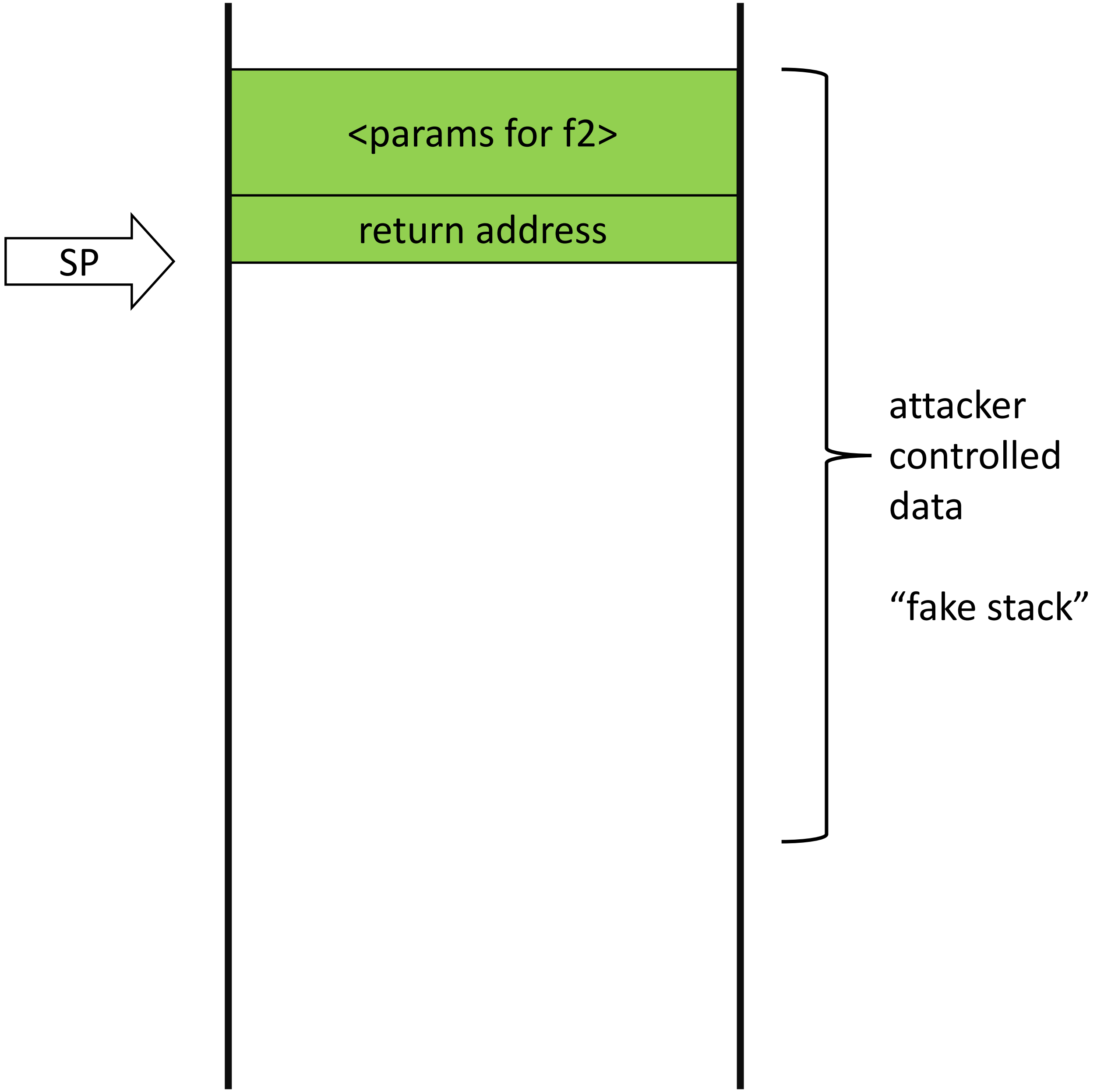


Code reuse attacks

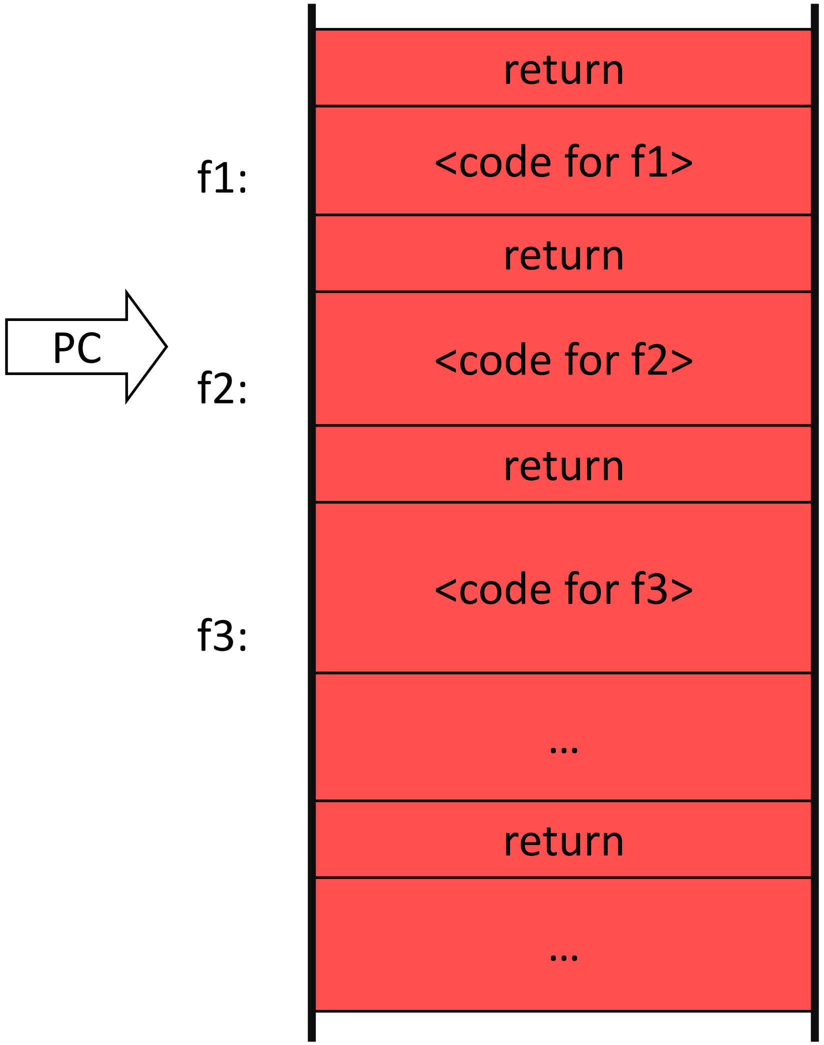


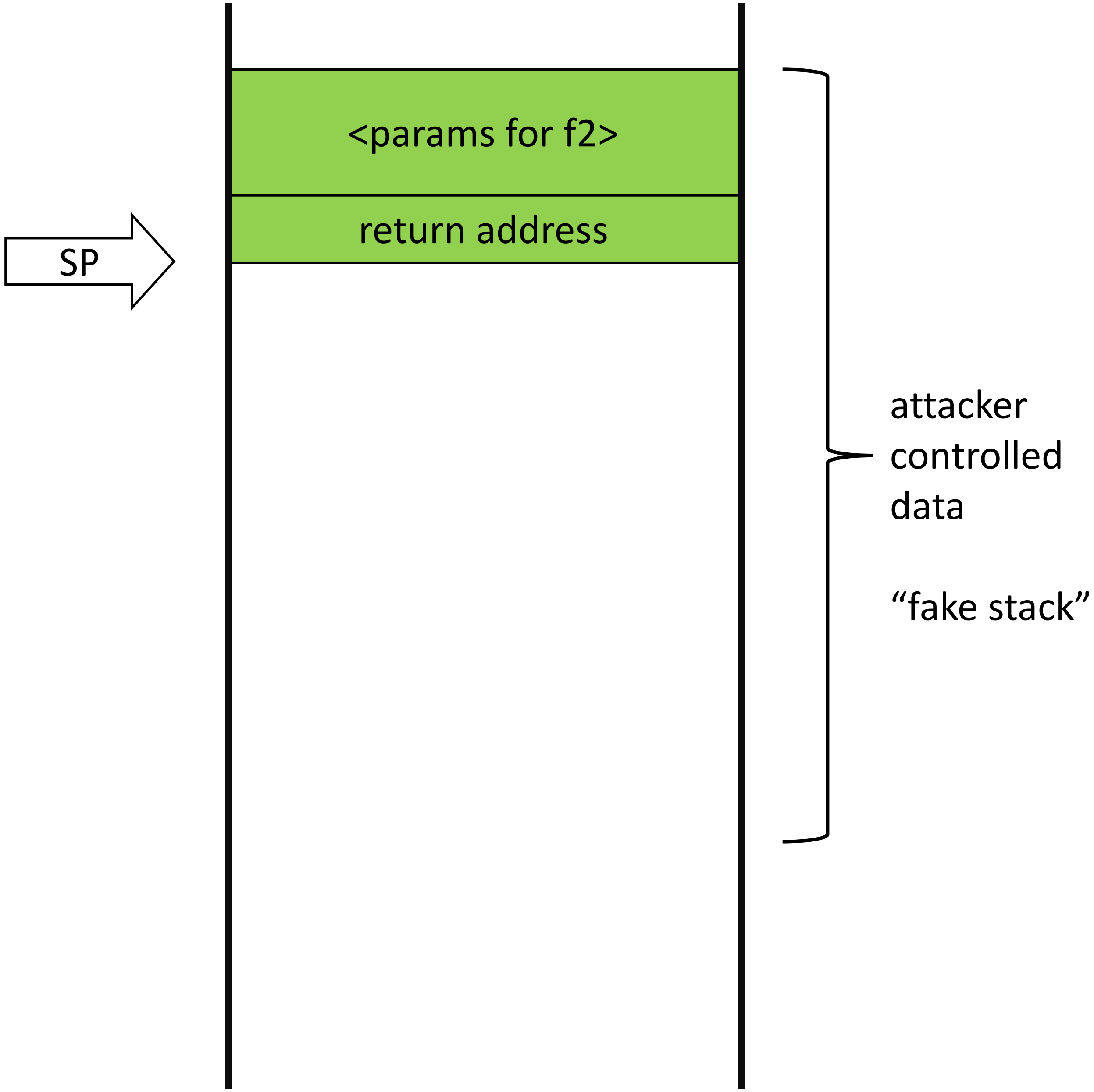
Code reuse attacks





Code reuse attacks





A concrete attack

- 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

```c
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

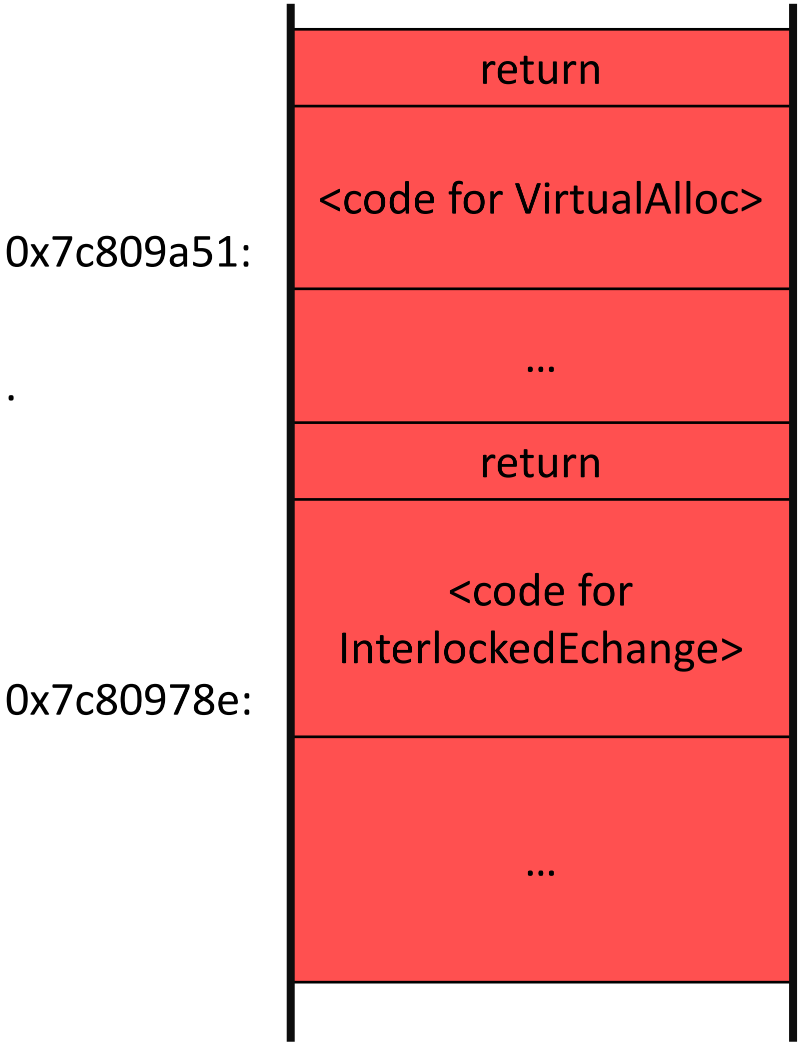
Assembly code of qsort:

```
...
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
...
```

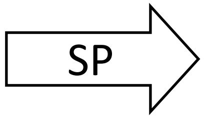
Trampoline code

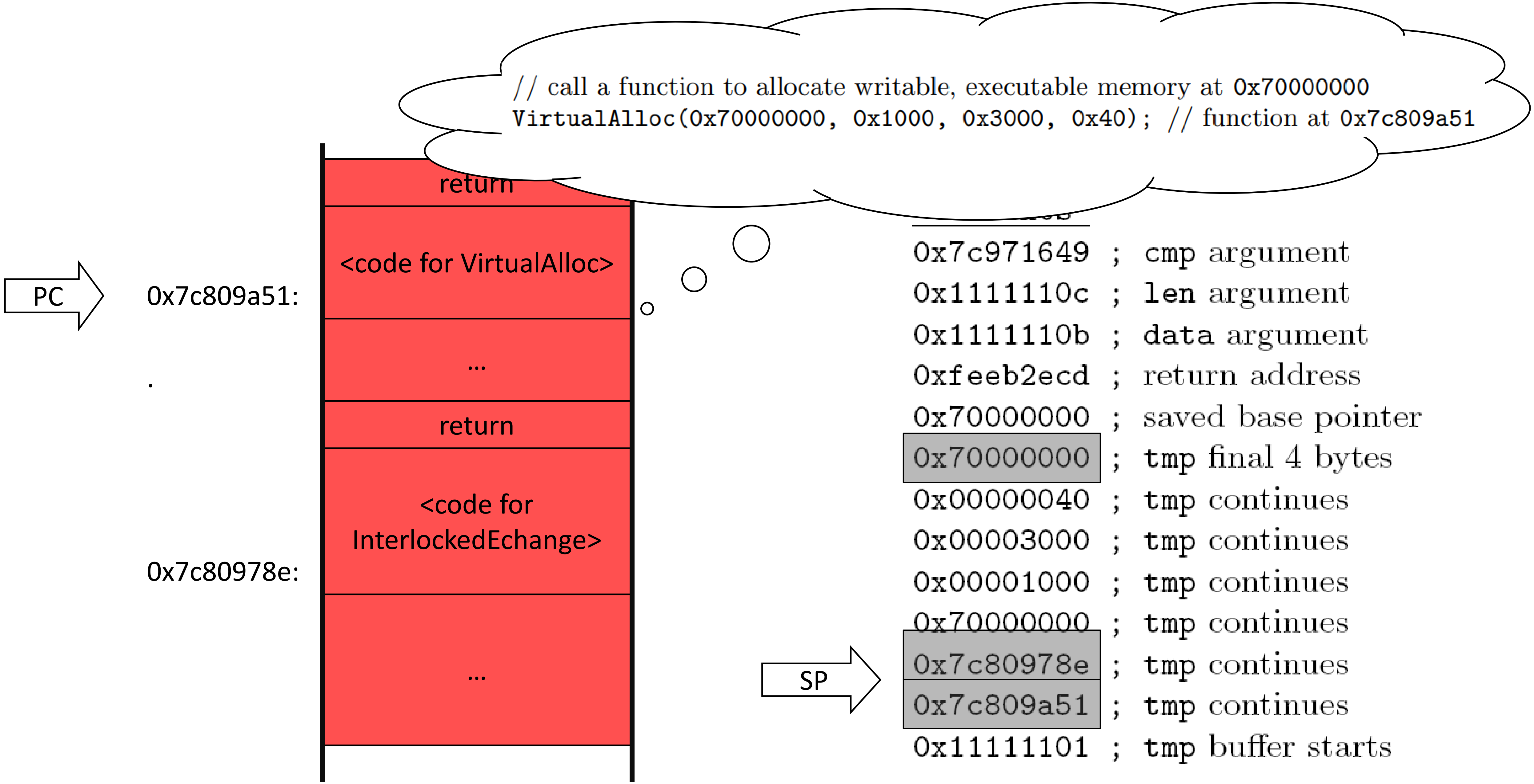
address	machine code opcode bytes	assembly-language version of the machine code
0x7c971649	0x8b 0xe3	mov esp, ebx ; change the stack location to ebx
0x7c97164b	0x5b	pop ebx ; pop ebx from the new stack
0x7c97164c	0xc3	ret ; return based on the new stack

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.

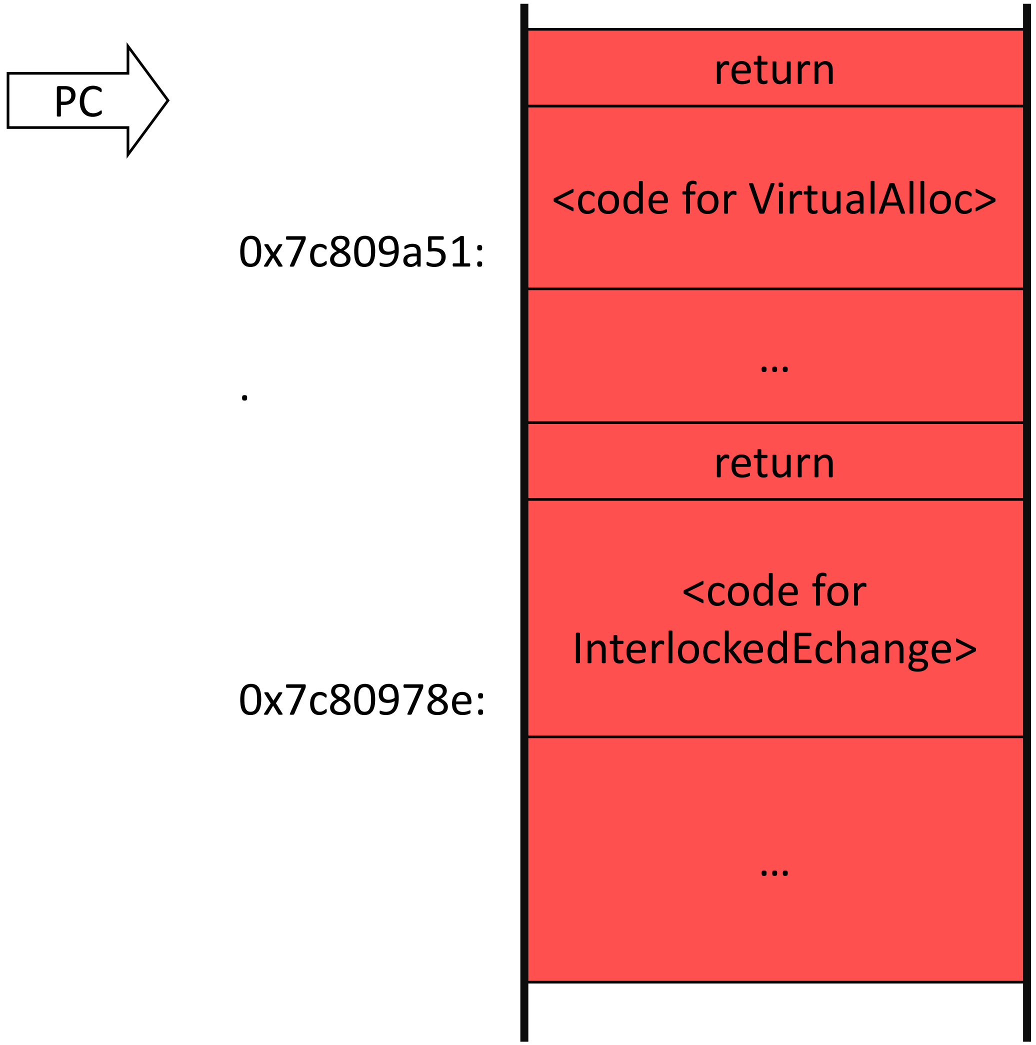


malicious overflow contents		
0x7c971649	;	cmp argument
0x1111110c	;	len argument
0x1111110b	;	data argument
0xfeeb2ecd	;	return address
0x70000000	;	saved base pointer
0x70000000	;	tmp final 4 bytes
0x00000040	;	tmp continues
0x00003000	;	tmp continues
0x00001000	;	tmp continues
0x70000000	;	tmp continues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

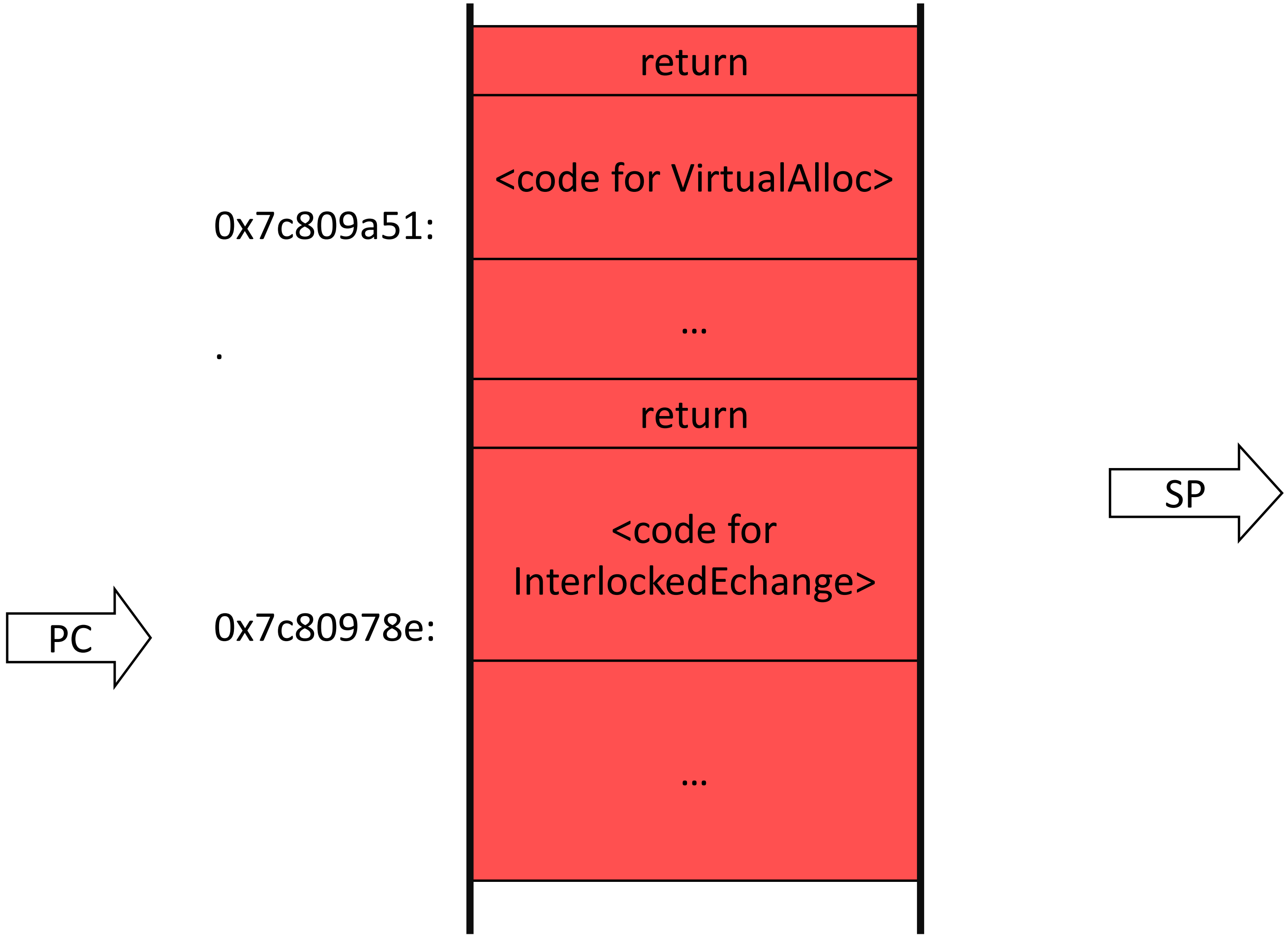




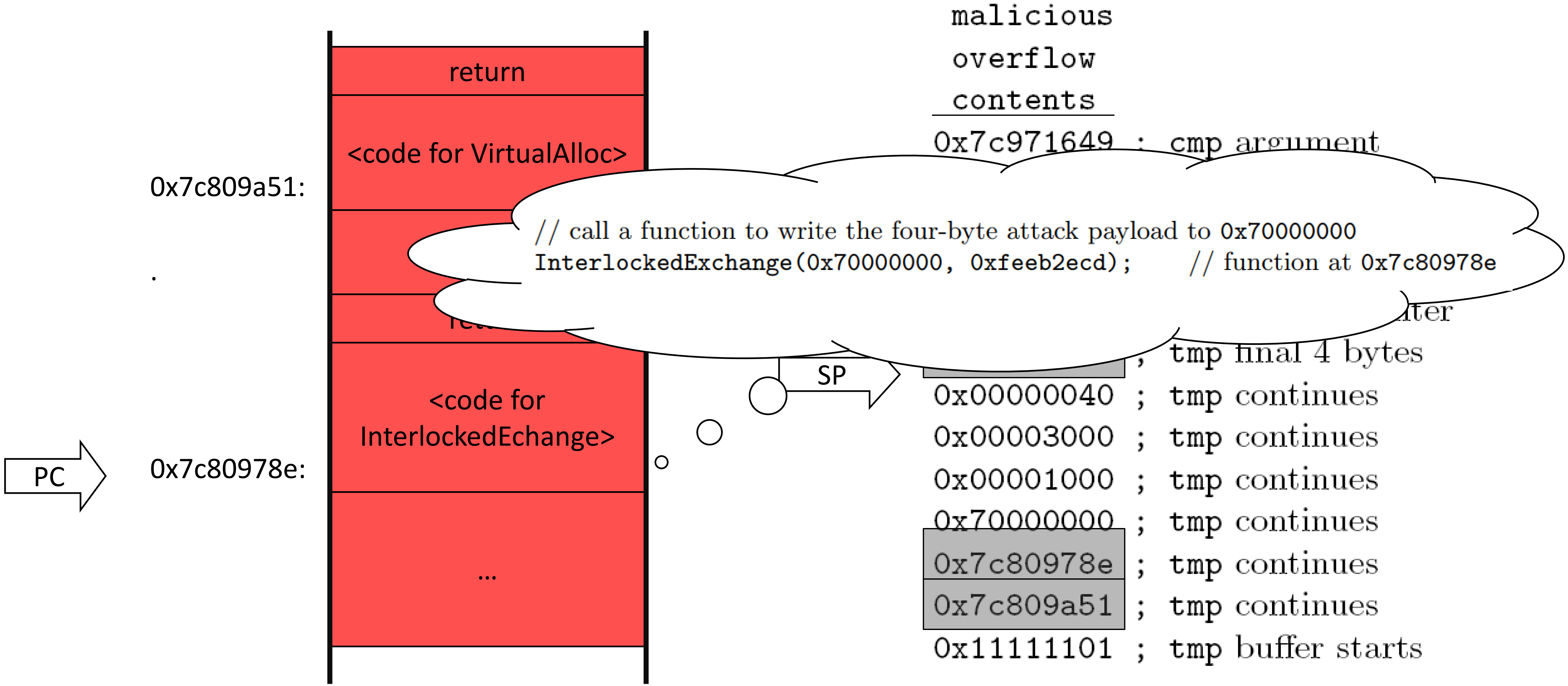
// call a function to allocate writable, executable memory at 0x70000000
VirtualAlloc(0x70000000, 0x1000, 0x3000, 0x40); // function at 0x7c809a51
PC
0x7c809a51:
return
<code for VirtualAlloc>
…
return
<code for InterlockedEchange>
…
0x7c80978e:
SP
0x7c971649 ; cmp argument
0x1111110c ; len argument
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



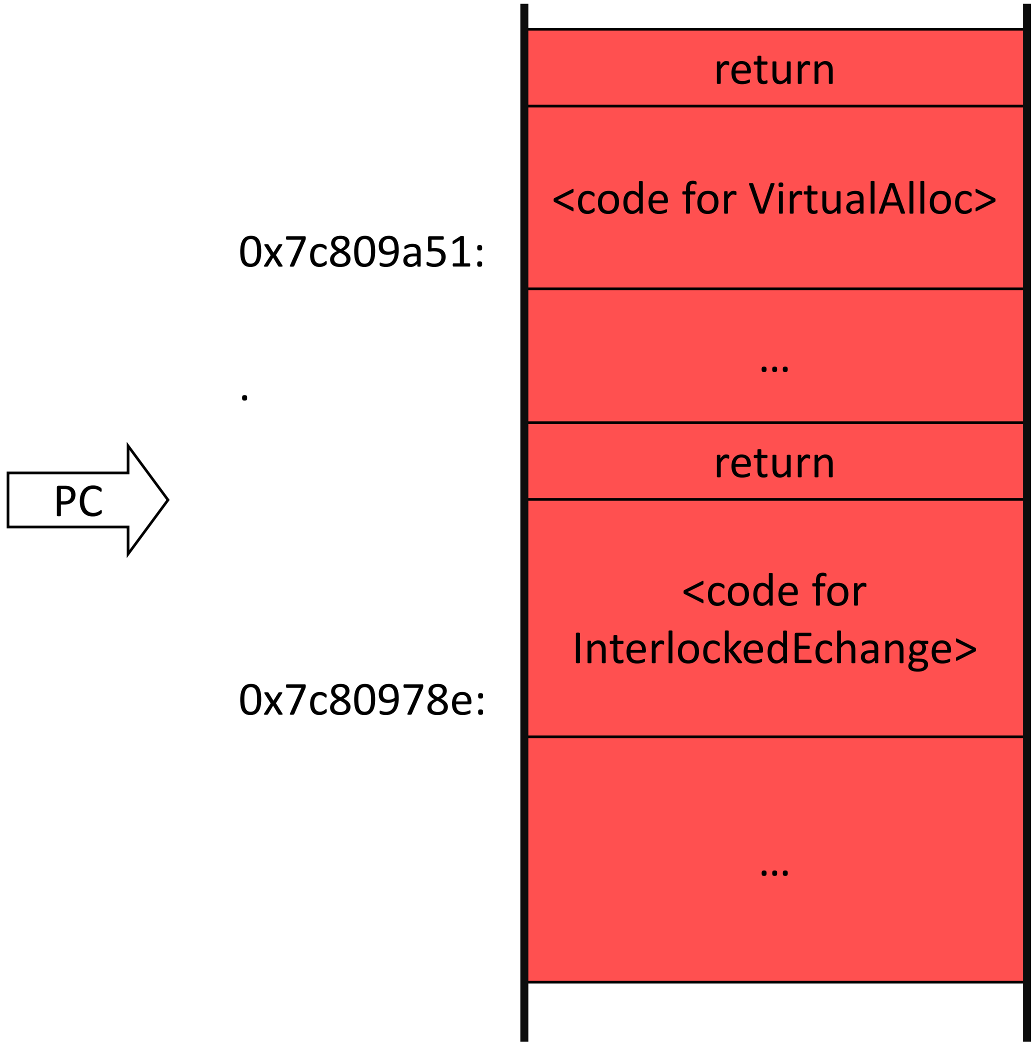
	malicious overflow contents	
	0x7c971649	; cmp argument
	0x1111110c	; len argument
	0x1111110b	; data argument
	0xfeeb2ecd	; return address
	0x70000000	; saved base pointer
	0x70000000	; tmp final 4 bytes
	0x00000040	; tmp continues
	0x00003000	; tmp continues
	0x00001000	; tmp continues
	0x70000000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

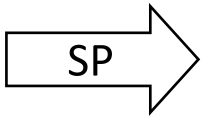



```
malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues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

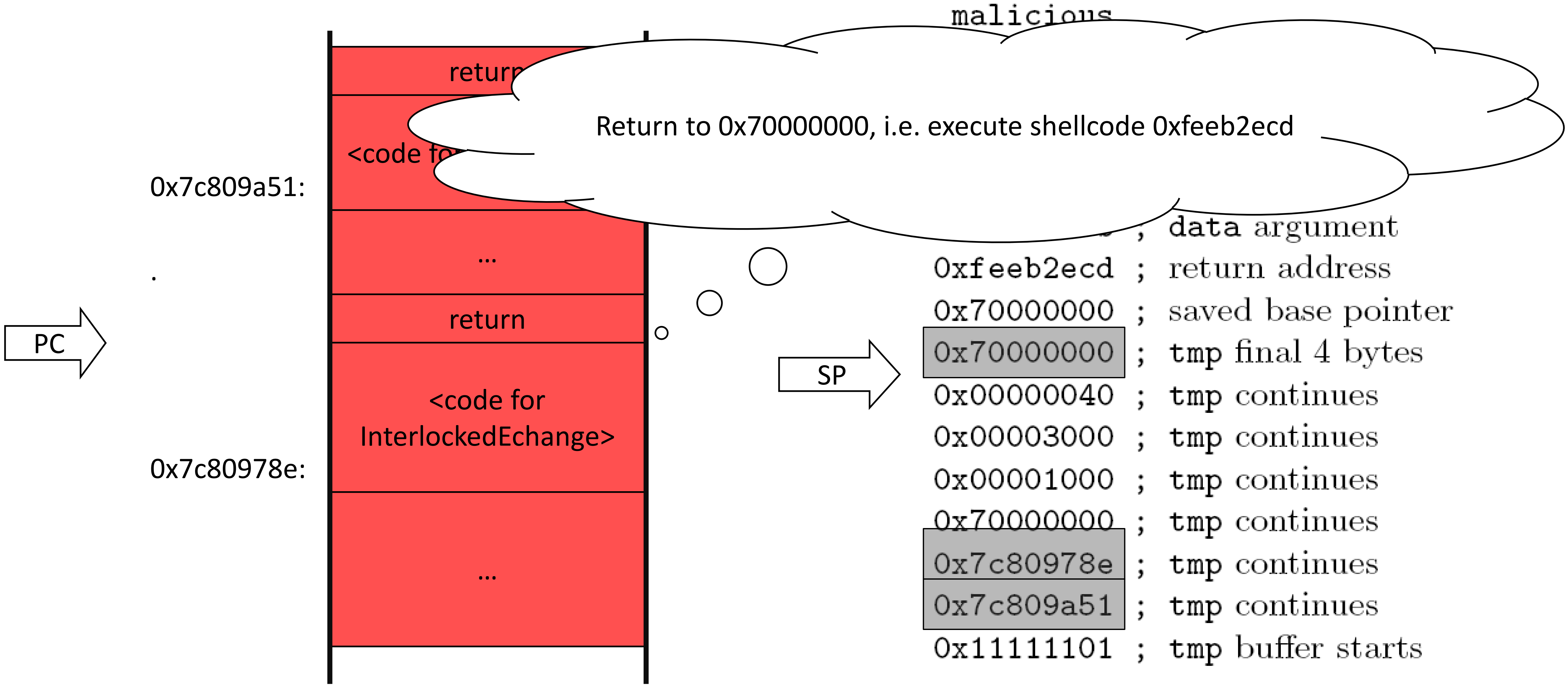


return
<code for VirtualAlloc>
0x7c809a51:
<code for InterlockedEchange>
0x7c80978e:
PC
...
SP
malicious overflow contents
0x7c971649 ; cmp argument
// call a function to write the four-byte attack payload to 0x70000000
InterlockedExchange(0x70000000, 0xfeeb2ecd); // function at 0x7c80978e
; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts






```
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
```



malicious
Return to 0x70000000, i.e. execute shellcode 0xfeeb2ecd
return
<code fo
0x7c809a51:
...
.
PC
return
<code for InterlockedEchange>
0x7c80978e:
...
SP
; 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

Modern variant: Return-Oriented-Programming (ROP)

- Key idea:
  - Instead of using the stack to “return into” functions, use it to chain “gadgets”
  - A *gadget* is a small piece of machine code ending in return
  - By finding a Turing-complete set of gadgets, one can “compile” arbitrary code into a fake stack calling these gadgets

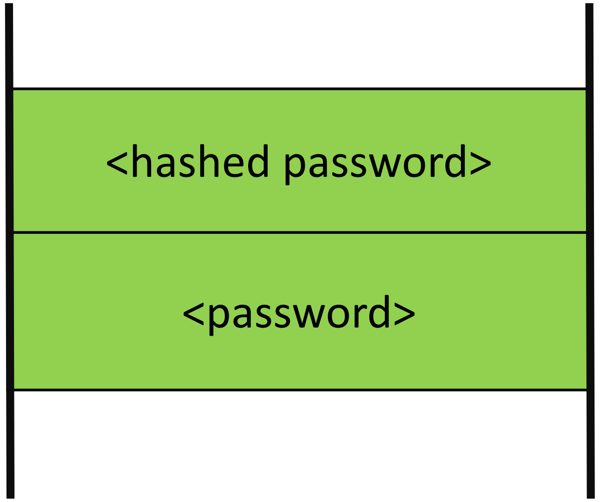
Attack scenario 4: Data-only attacks

- *Data-only attacks* proceed by changing only data of the program under attack
- Depending on the program under attack, this can result in interesting exploits
- We discuss two examples:
  - The unix password attack
  - Overwriting the environment table

Unix password attack

- Old implementations of login program looked like this:

Stack

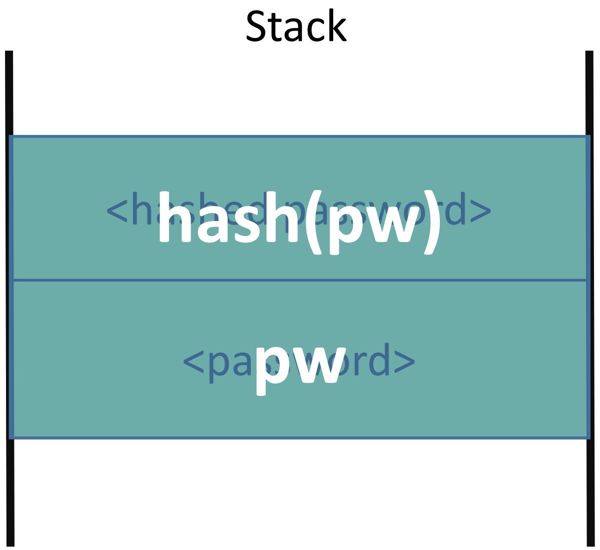


Password check in login program:

1. Read loginname
2. Lookup hashed password
3. Read password
4. Check if
   hashed password = hash (password)

Unix password attack

- Old implementations of login program looked like this:



Password check in login program:

1. Read loginname
2. Lookup hashed password
3. Read password
4. Check if
   hashed password = hash (password)

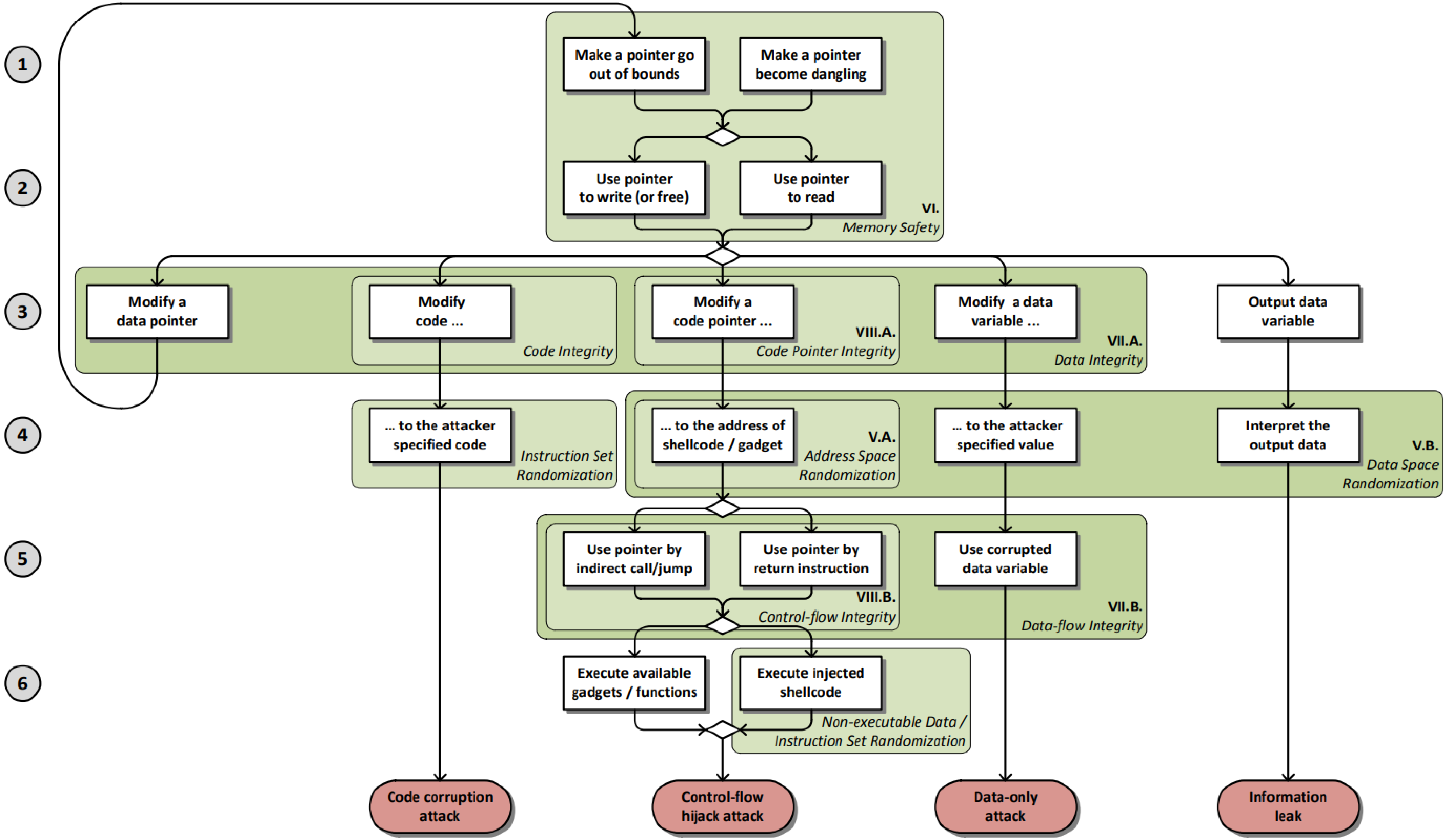
- Hence, typing in a password of the form pw ++ hash(pw) always succeeds

Overwriting the environment table

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```

Other attack scenarios

- We have discussed 4 attack scenarios:
  - Call stack smashing
  - Function pointer overwrite
  - Code-reuse attacks like return-oriented-programming
  - Data-only attacks
- Other variations exist and attacks can also be combined
- A structured overview of attacks is in:
  - Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song: *SoK: Eternal War in Memory*. IEEE Symposium on Security and Privacy 2013.


1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI.
Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A.
Code Pointer Integrity
Modify a data variable ...
VII.A.
Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A.
Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B.
Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B.
Control-flow Integrity
Use corrupted data variable
VII.B.
Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Overview

- System model
- Attack scenarios
- Mitigating attacks
- Avoiding vulnerabilities
- Conclusions

Mitigating attacks

- The first line of defense developed against the attacks we discussed builds in countermeasures in compiler/operating system/hardware to make attacks harder
- We discuss the widely deployed mitigations
- But all these defenses are *partial* – they just make it harder to develop an effective attack
- We will illustrate that by means of a running example taken from the SYSSEC 10K Challenge (http://10kstudents.eu/material/)

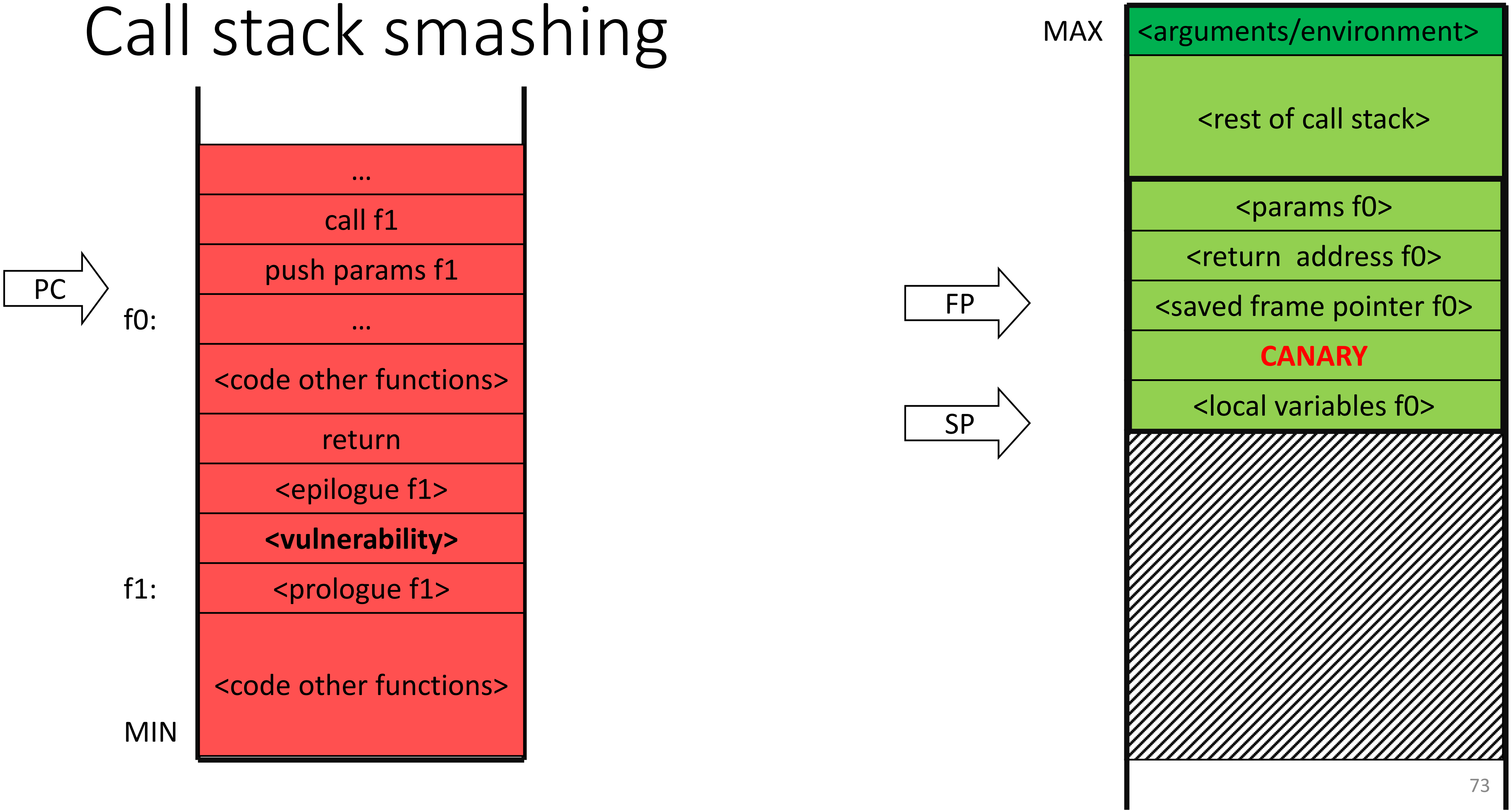
```c
char gWelcome[] = "Welcome to our system!";
void echo (int fd) {
  int len;
  char name[64],reply[128];

  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);
}
```


Source: http://10kstudents.eu/material/

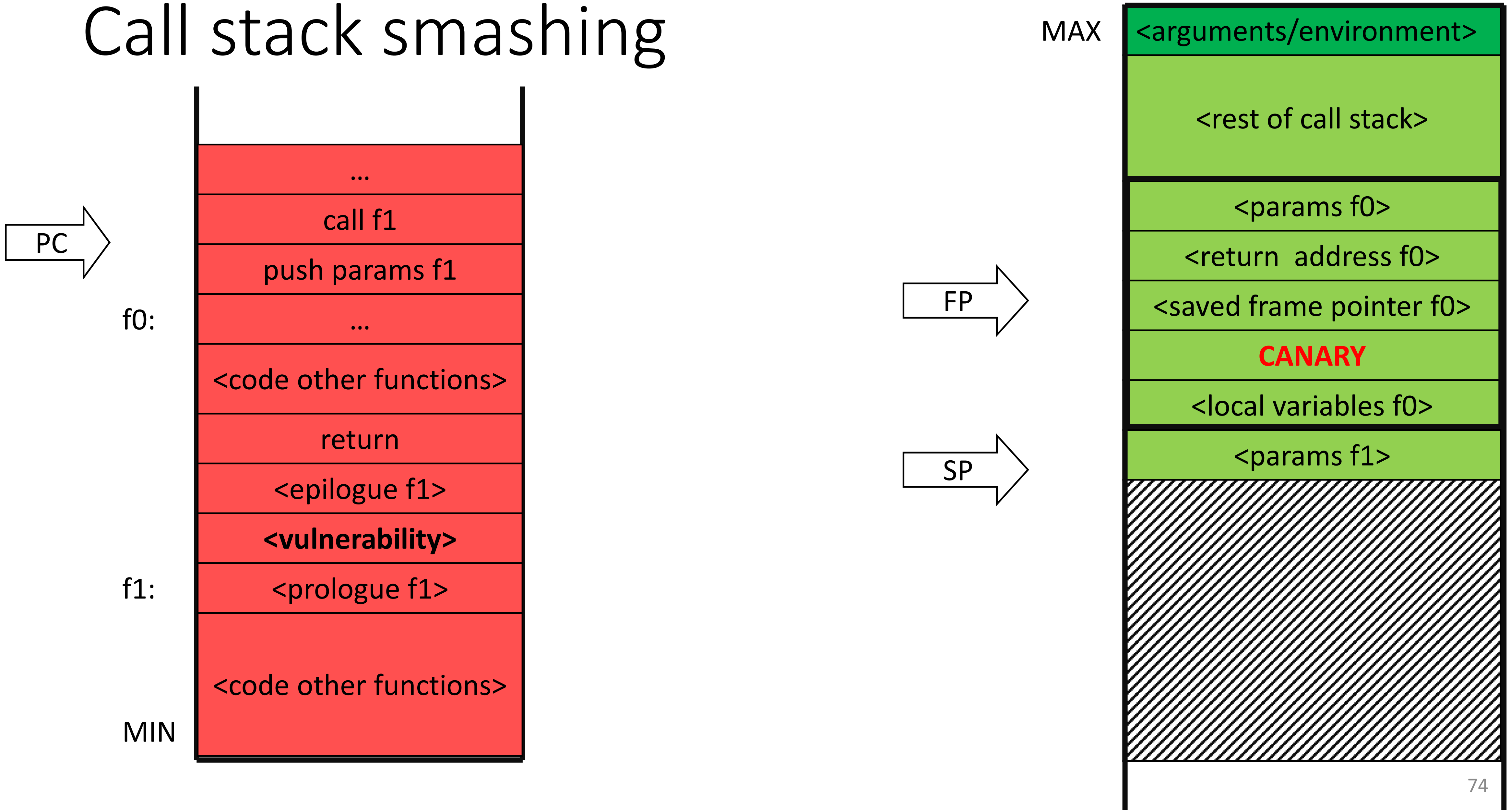
Mitigation 1: Stack canaries

- 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

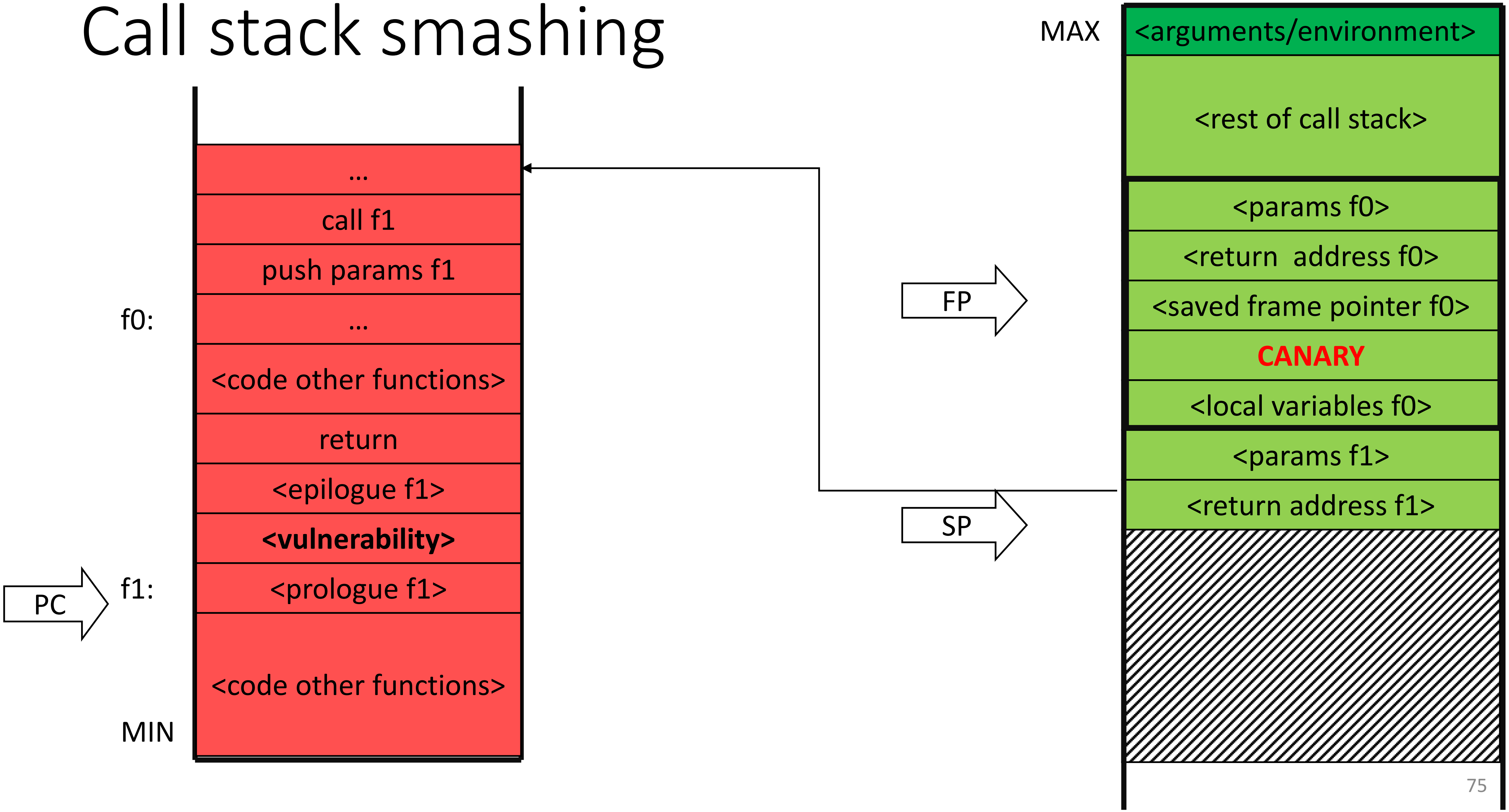
Call stack smashing



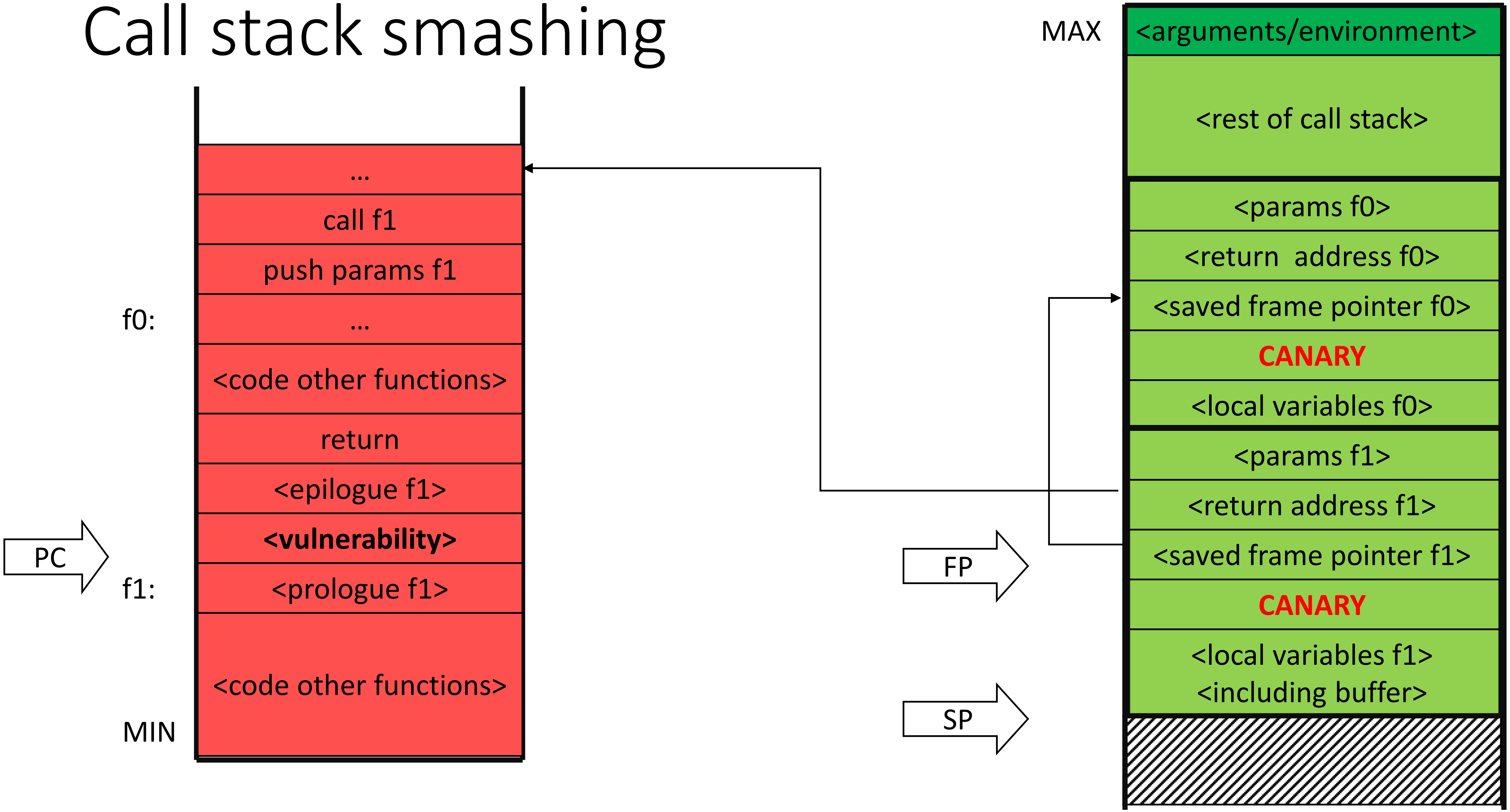
Call stack smashing



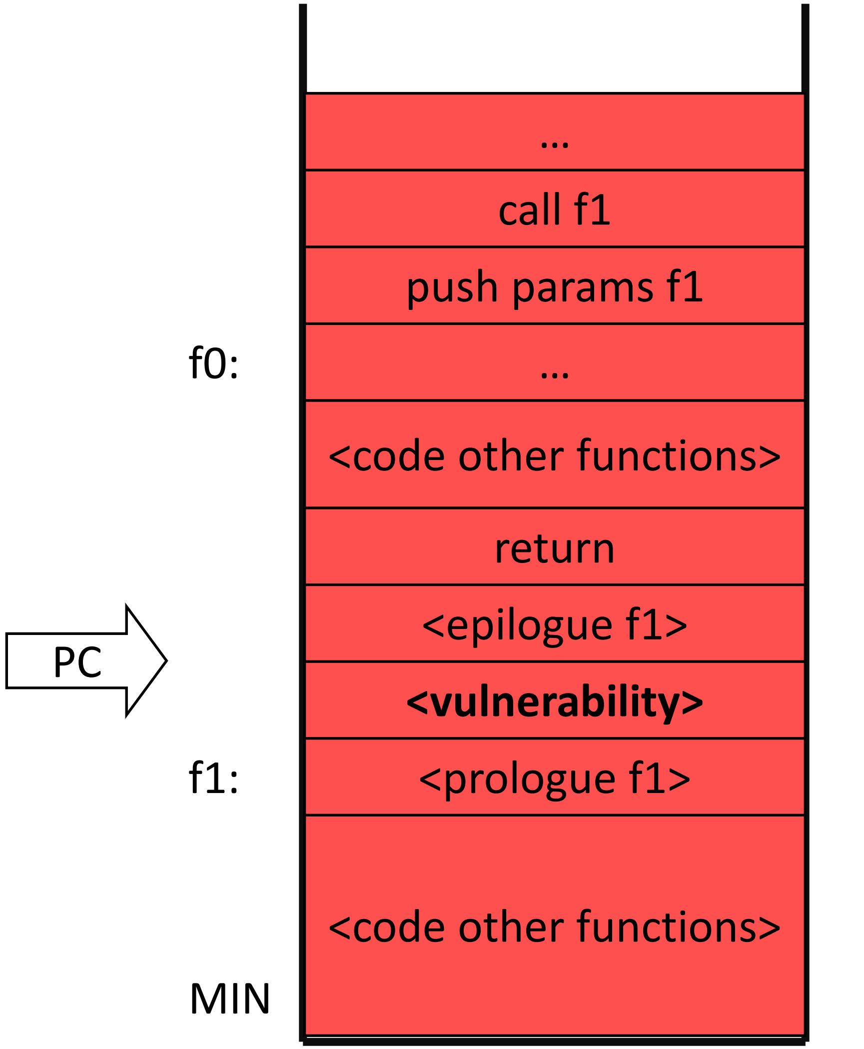
Call stack smashing

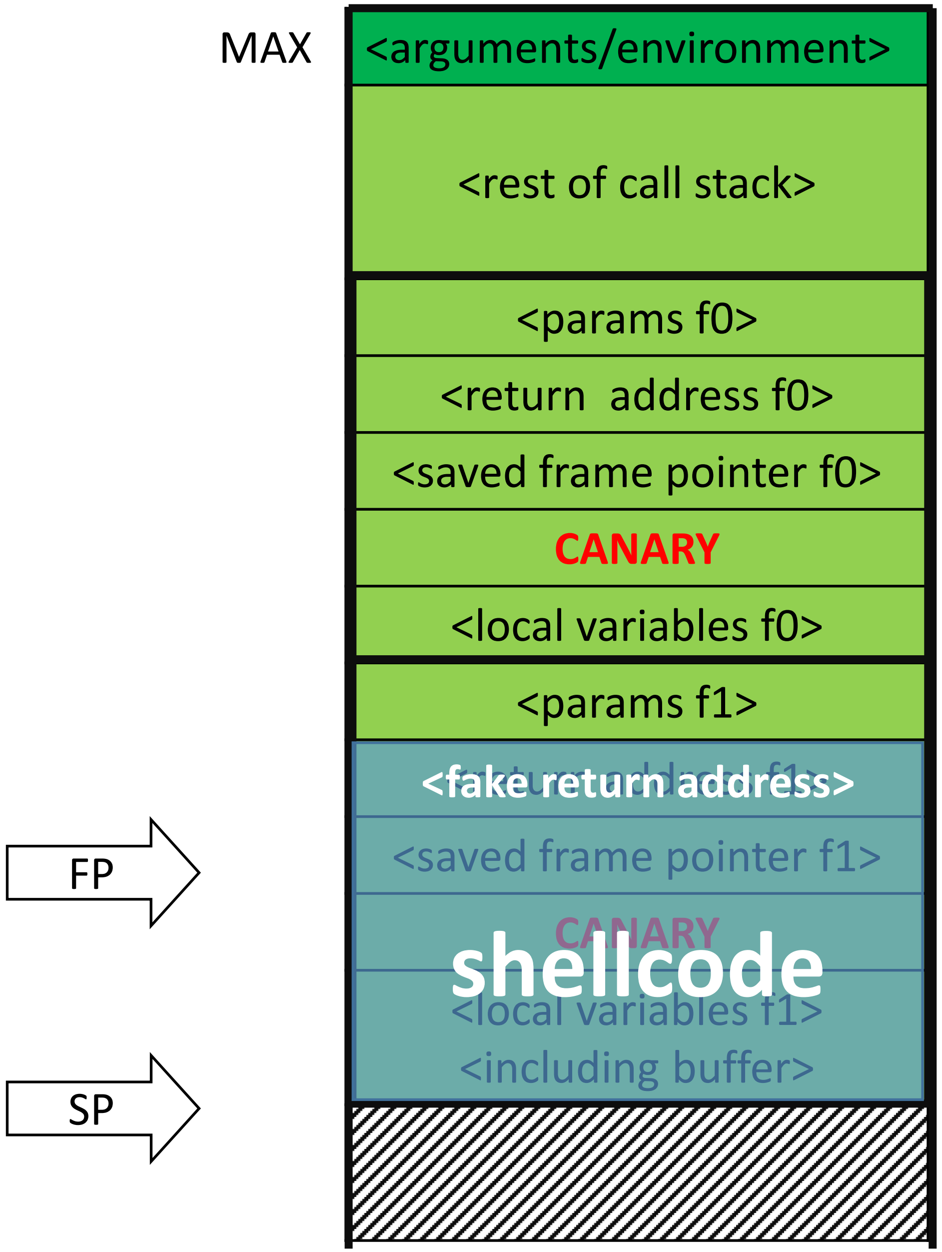


Call stack smashing

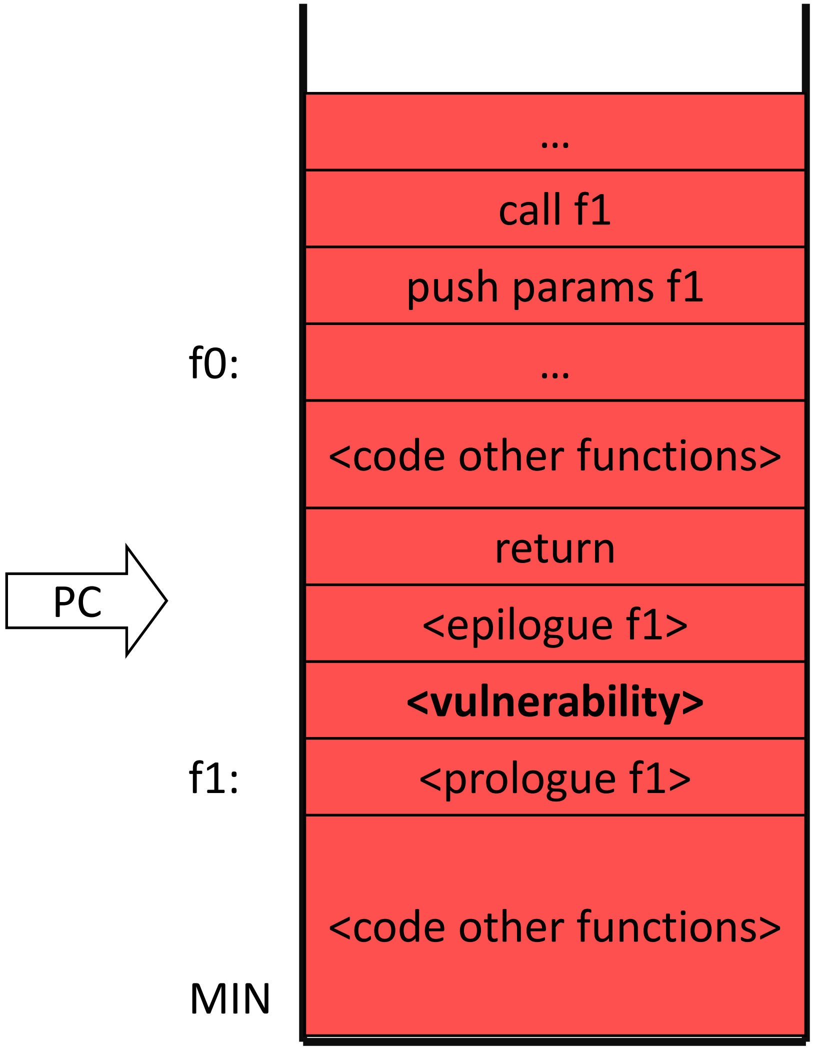


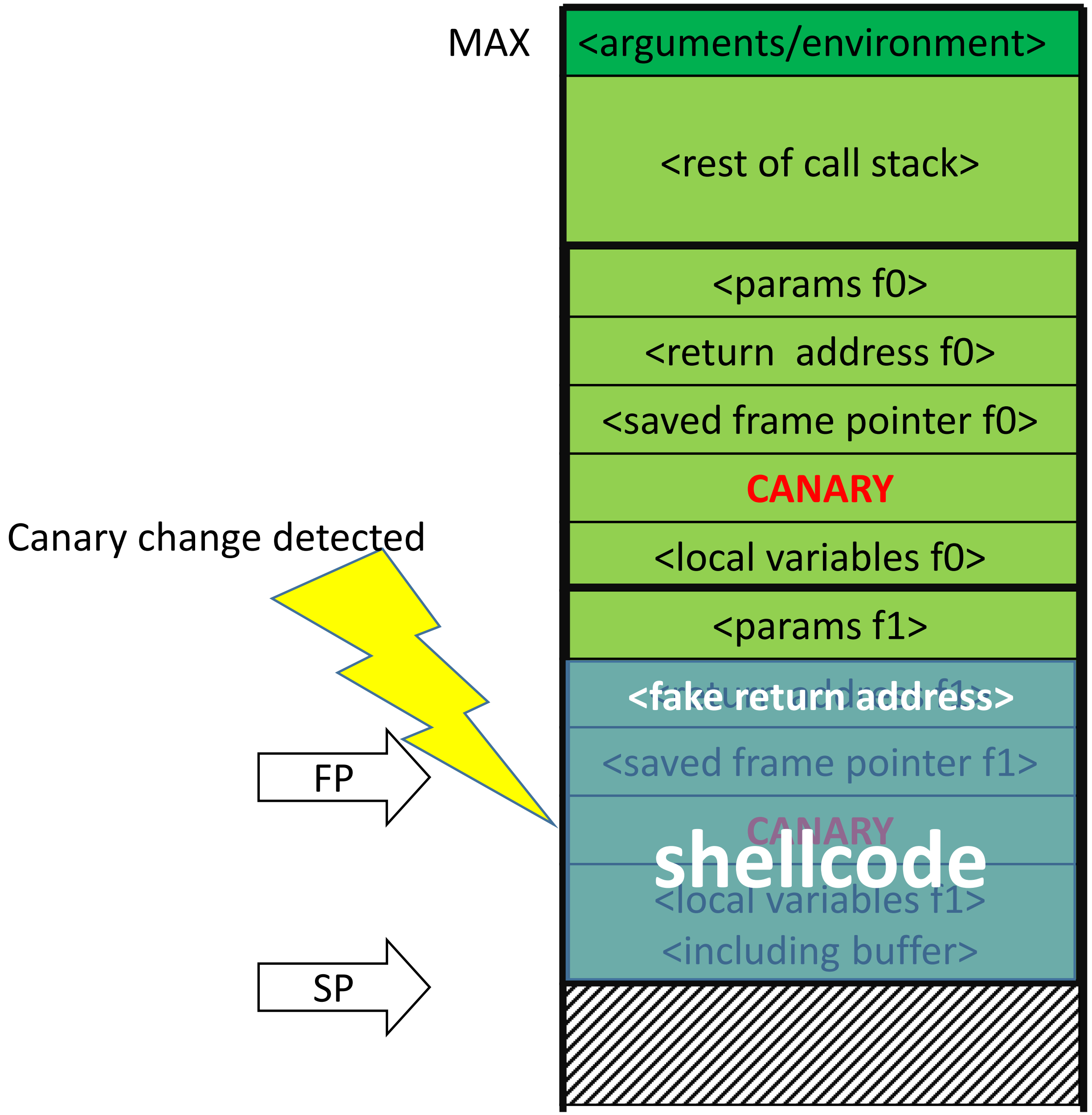
Call stack smashing





Call stack smashing





```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);
}
```


Source: http://10kstudents.eu/material/

Mitigation 2: 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, and code memory as non-writable
- This counters direct code injection and code corruption, but not code-reuse or data-only attacks
- In addition, this countermeasure may break certain legacy applications

```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);
}
```


Source: http://10kstudents.eu/material/

Mitigation 3: Address Space 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

```c
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
}
```

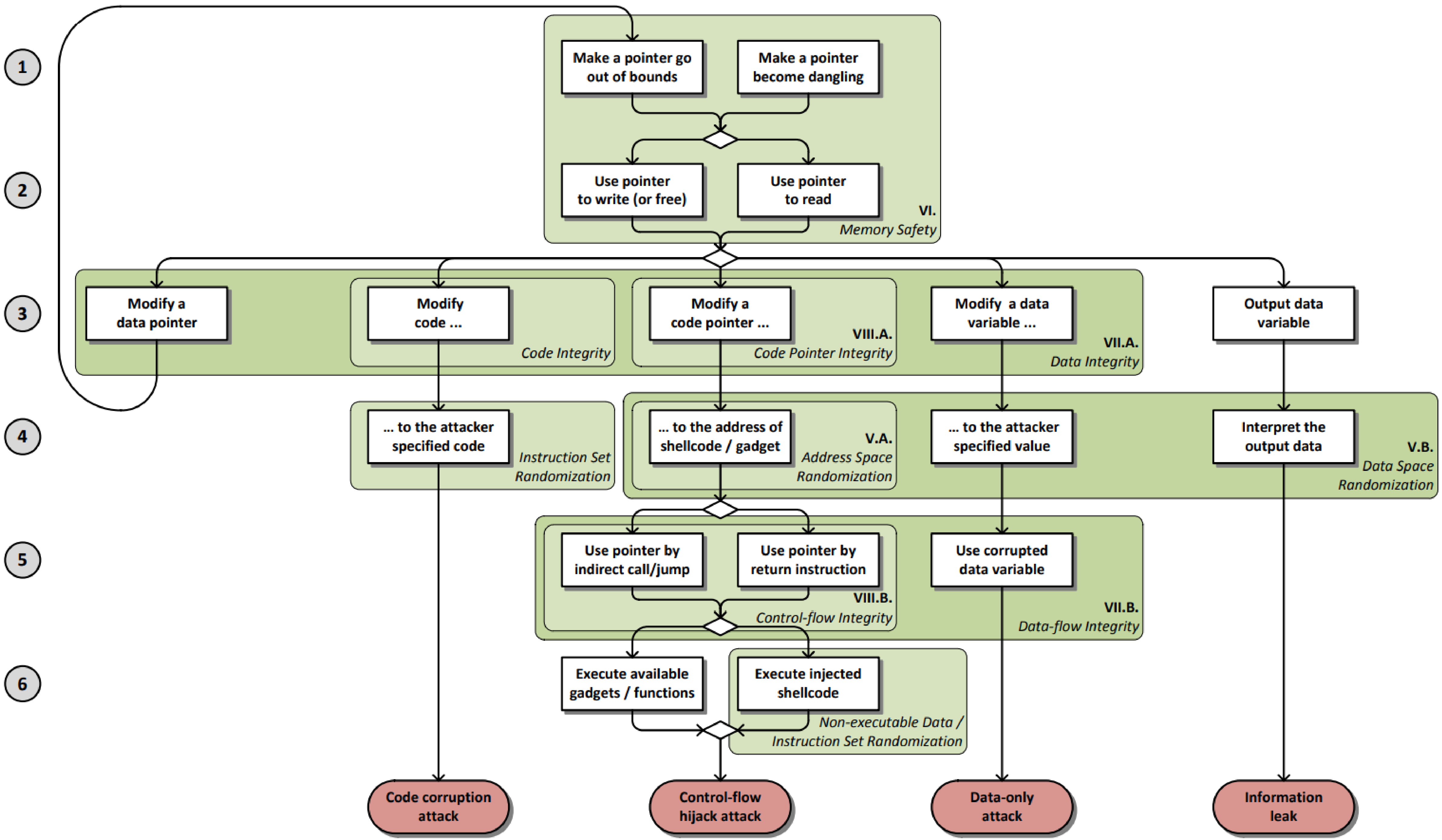
stack one		stack two		
address	contents	address	contents	
0x0022feac	0x008a13e0	0x0013f750	0x00b113e0	; cmp argument
0x0022fea8	0x00000001	0x0013f74c	0x00000001	; len argument
0x0022fea4	0x00a91147	0x0013f748	0x00191147	; data argument
0x0022fea0	0x008a1528	0x0013f744	0x00b11528	; return address
0x0022fe9c	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.

```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);
}
```


Source: http://10kstudents.eu/material/

Other mitigations

- A wide variety of other such mitigations have been investigated
- Again, a good overview is provided by:
    - Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song: *SoK: Eternal War in Memory*. IEEE Symposium on Security and Privacy 2013.
- But the general belief is that these automatic, efficient "mitigate-the-exploit" approaches are just stop-gap measures


1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI.
Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A.
Code Pointer Integrity
Modify a data variable ...
VII.A.
Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A.
Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B.
Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B.
Control-flow Integrity
Use corrupted data variable
VII.B.
Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Overview

- System model
- Attack scenarios
- Mitigating attacks
- Avoiding vulnerabilities
- Conclusions

Low-level vulnerabilities

- A C program can only be attacked using the techniques we discussed if it has a *memory management vulnerability*, a bug that can lead to an incorrect memory access
- These vulnerabilities come in a number of forms:
  - Spatial vulnerability: access allocated memory out of bounds
  - Temporal vulnerability: access memory after it has been freed.
  - Pointer forging: an invalid construction of a pointer, for instance through casting
  - Incorrect call of a function that supports a variable number of arguments, for instance the printf() family of functions

Spatial vulnerabilities

- Programming languages can offer various mechanisms to index into allocated memory regions:
  - Array indexing a[i]
  - Field access in structs, unions or objects
  - Pointer arithmetic, where a pointer “walks” over an allocated region of memory
    - E.g. `for( ; *src != ’\0’; ++src, ++tgt ) *tgt = *src;`
- Each of these mechanisms can lead to spatial memory vulnerabilities

Temporal vulnerabilities

- How long are pointers valid?

```
int global;

int *f(int param) {
    int local;
    int *p1 = &global;
    int *p2 = &param;
    int *p3 = &local;
    int *p4 = malloc(sizeof(int));
    return p1; // or p2, or p3 or p4?
}
```

Example temporal vulnerability

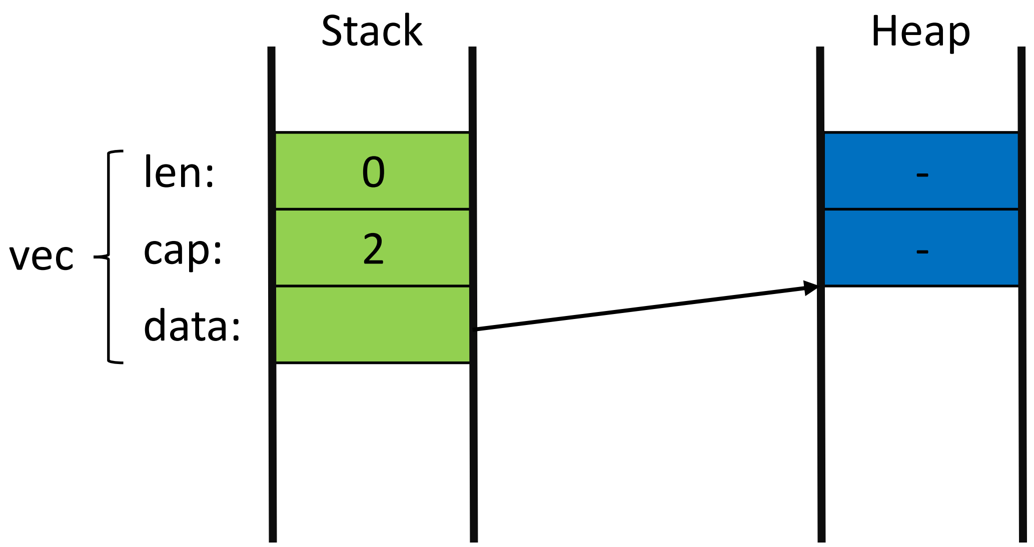
```c
typedef struct {
    int len;
    int cap;
    int *data;
} vec;
vec newvec() {
    vec v;
    v.len = 0;
    v.cap = 2;
    v.data = malloc(2 * sizeof(int));
    return v;
}
void push(vec* v, int i) {
    if (v->len >= v->cap) {
        v->cap *= 2;
        int *new = malloc(v->cap * sizeof(int));
        memcpy(new,v->data, v->len *sizeof(int));
        free(v->data);
        v->data = new;
    }
    v->data[v->len++] = i;
}
```

```c
void printvec(vec v) {
    int i;
    for (int i = 0; i< v.len; i++) {
        printf("%d\n",v.data[i]);
    }
}
int* get(vec* v, int i) {
    return v->data + i;
}
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

Example temporal vulnerability

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

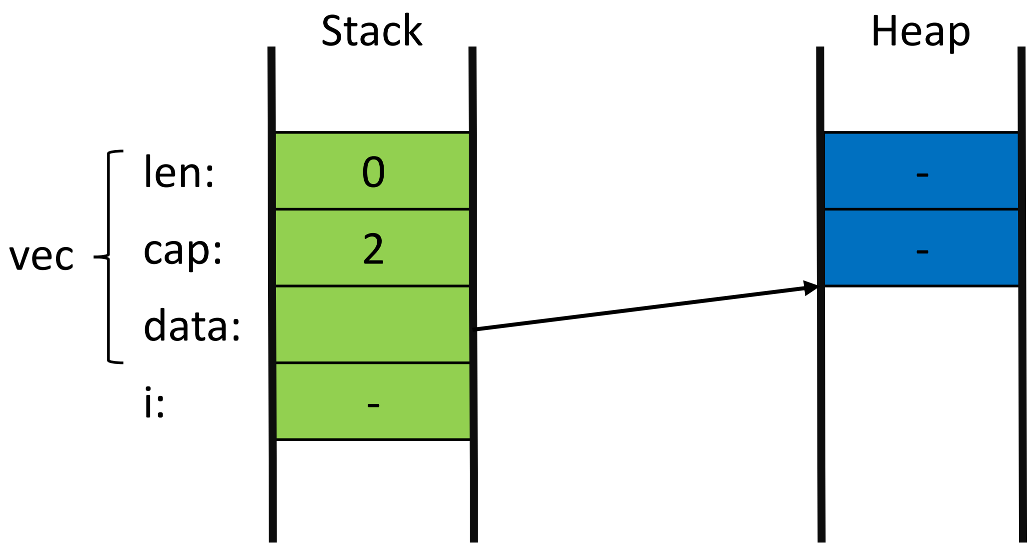
Output:



Example temporal vulnerability

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

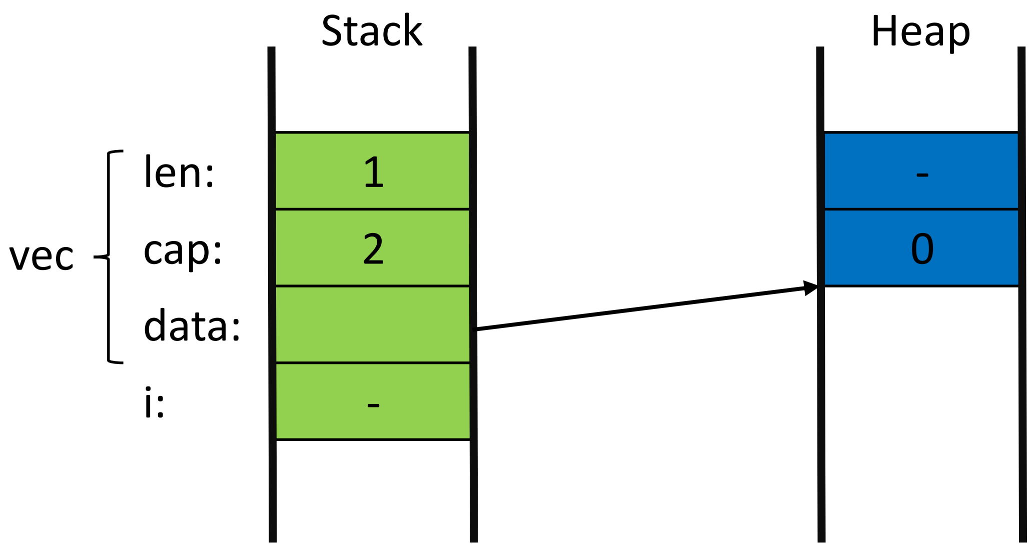
Output:



Example temporal vulnerability

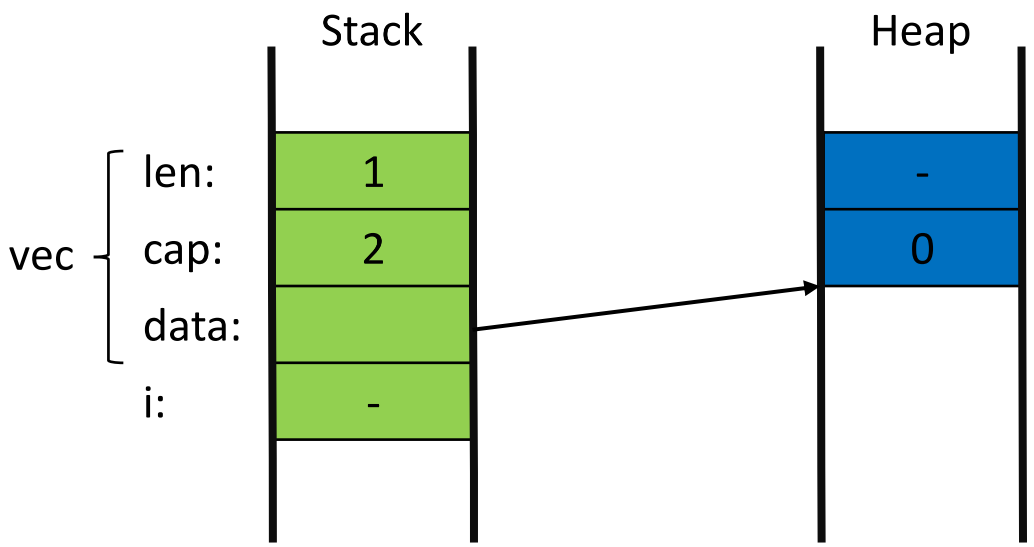
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

Output:



Example temporal vulnerability

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

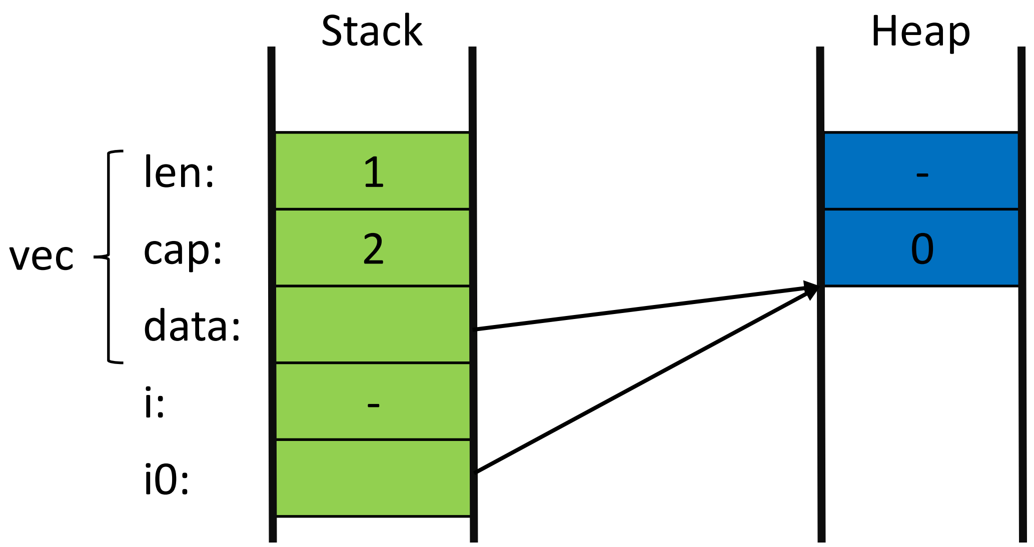


Output:
0

Example temporal vulnerability

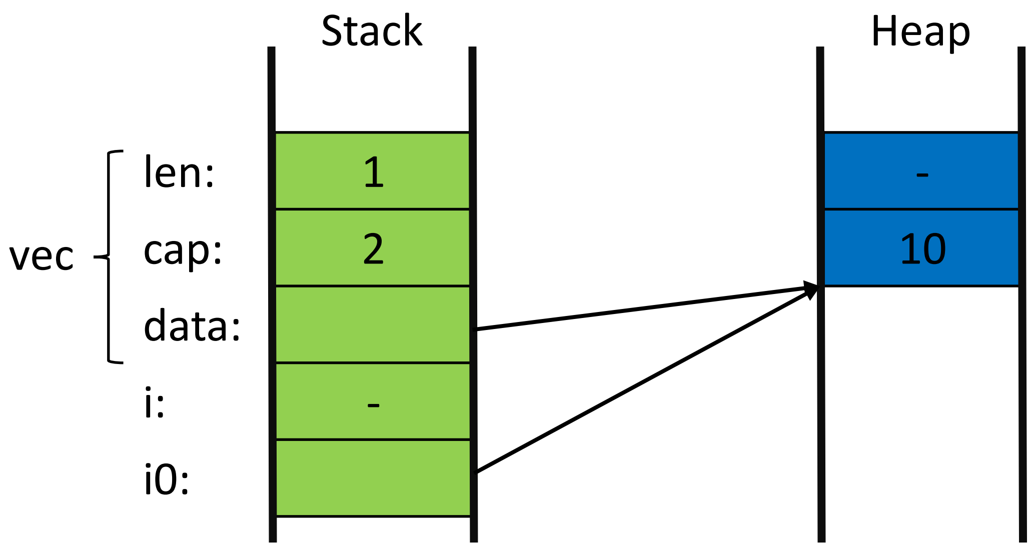
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

Output:
0



Example temporal vulnerability

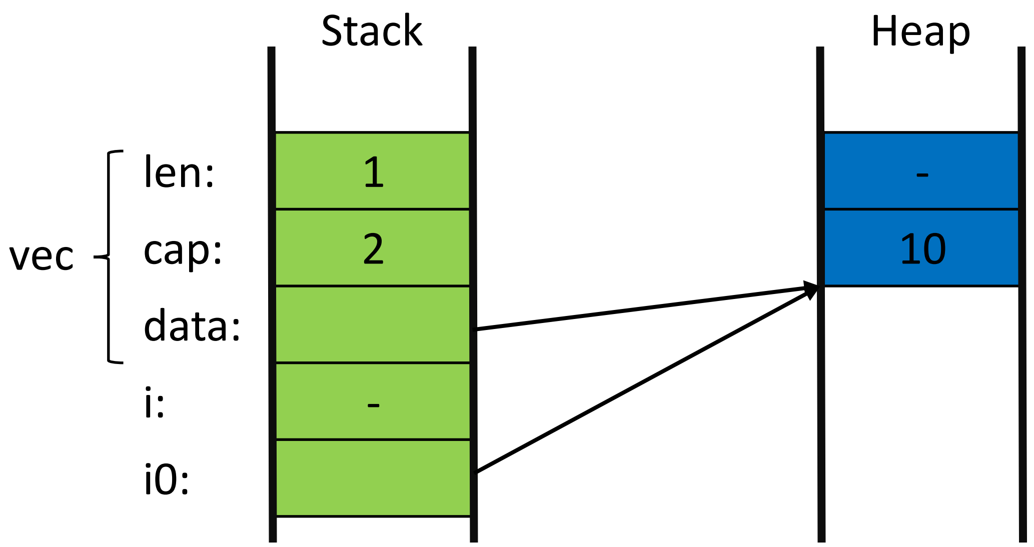
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



Output:
0

Example temporal vulnerability

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

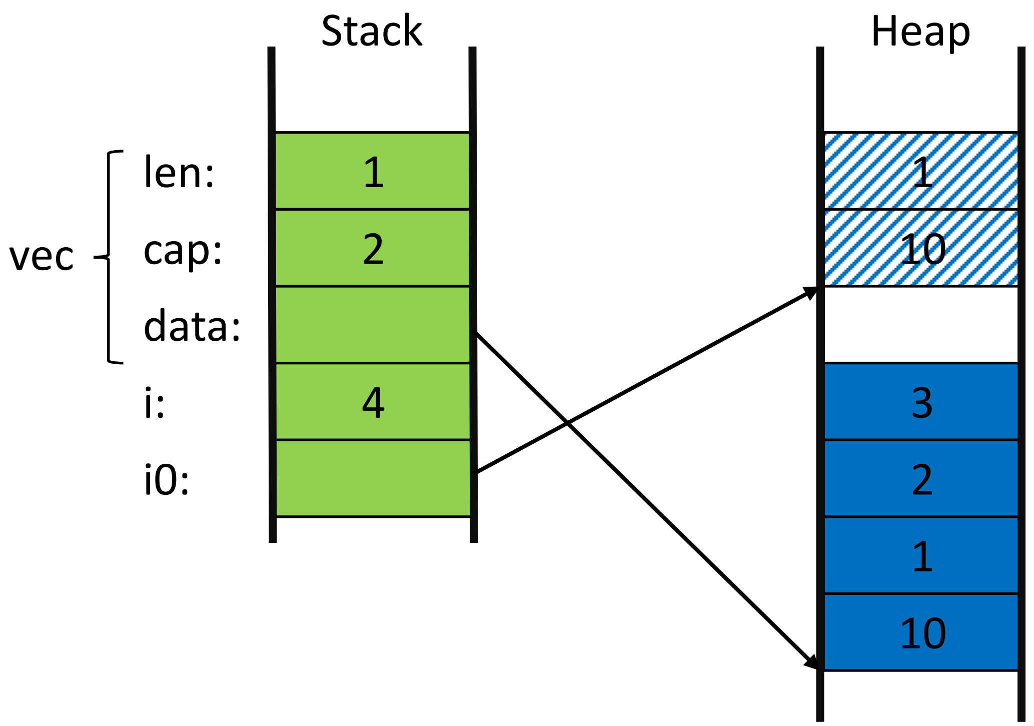


Output:
0
10

Example temporal vulnerability

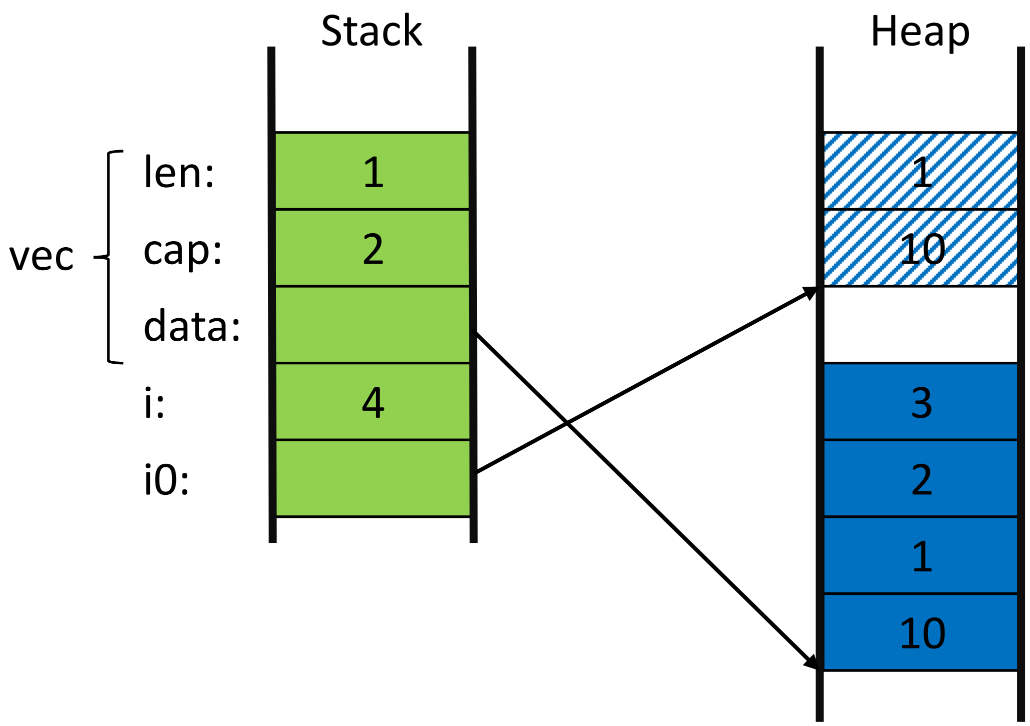
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

Output:
0
10



Example temporal vulnerability

```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```



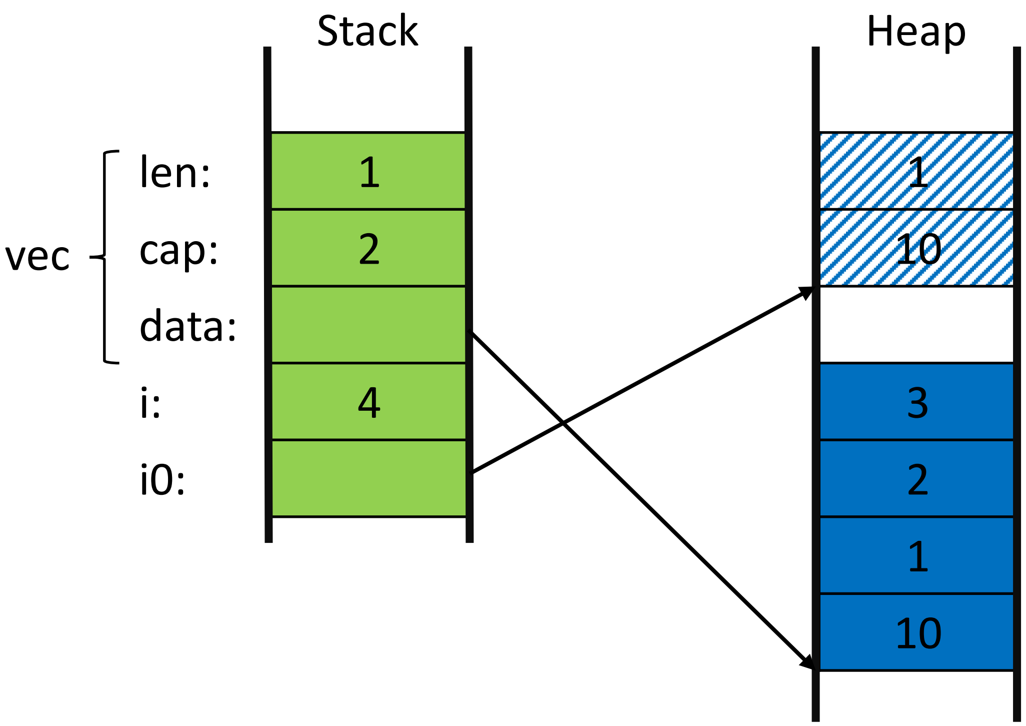
Output:
0
10
10
1
2
3

Example temporal vulnerability

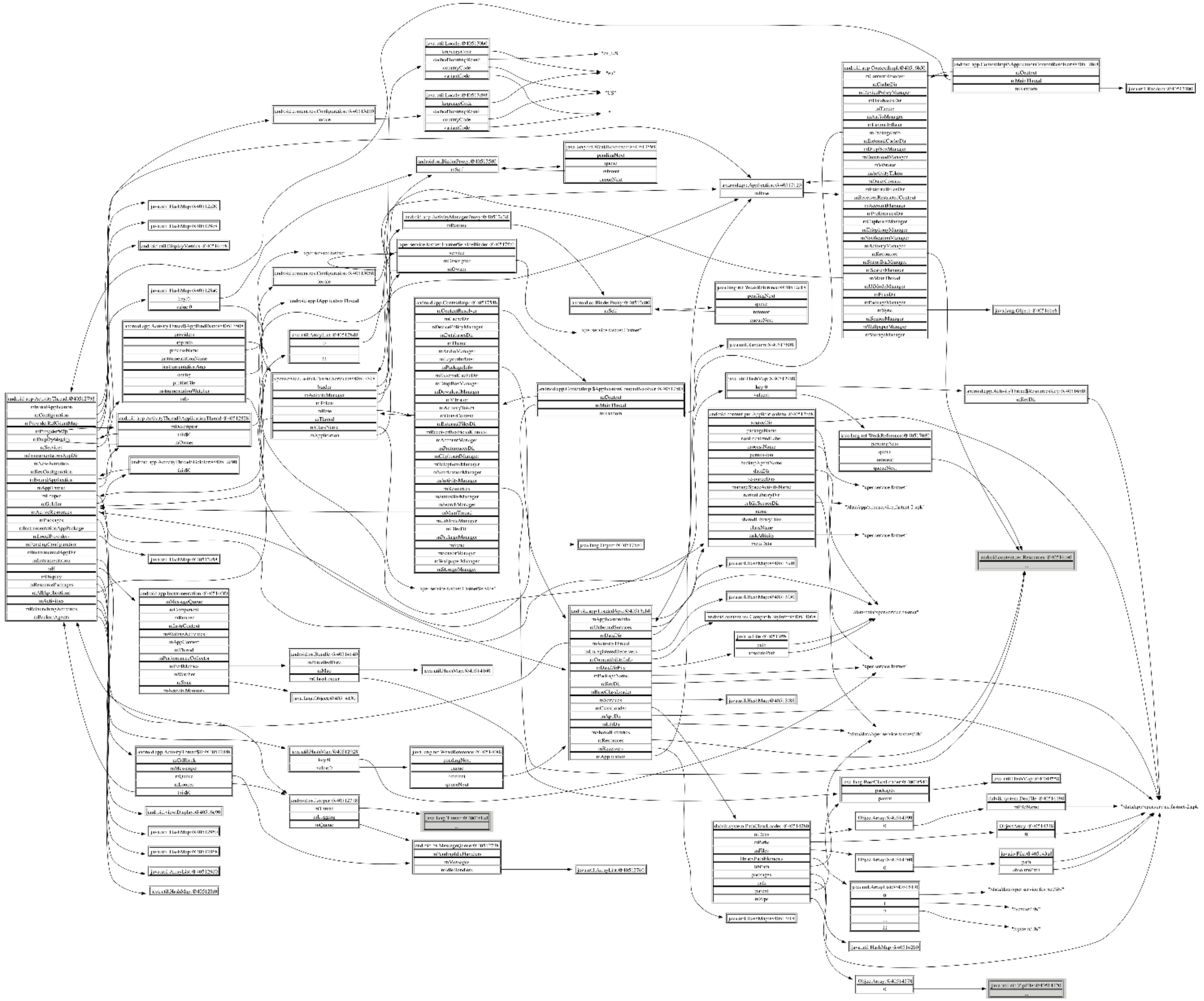
```
void main() {
    vec v = newvec();
    int i;
    push(&v,0);
    printvec(v);
    int* i0 = get(&v,0);
    *i0 = 10;
    printvec(v);
    for (i=1; i< 4; i++) push(&v,i);
    printvec(v);
    *i0 = 20;
    printvec(v);
}
```

Output:
0
10
10
1
2
3

Temporal memory error



Real heap is more complicated …



Avoiding such vulnerabilities

- Preventing introduction
  - Coding guidelines
  - Use of safe languages
- Detecting
  - Code review
  - Static analysis
    - 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
    - Fuzz testing
    - Directed fuzz-testing / symbolic execution
    - Run-time memory safety checkers
      - E.g. AddressSanitizer

Overview

- System model
- Attack scenarios
- Mitigating attacks
- Avoiding vulnerabilities
- Conclusions

Conclusions

- The security of software in C-like languages has been the subject of decades of *attacker-defender race*
- The desire to maintain C's performance and backward compatibility have made rigorous solutions to the problem of memory management vulnerabilities hard
- But some promising solutions are on the horizon
  - Hardware support for safe compilation of C
  - New systems programming languages with better safety guarantees
- Reading material:
  - Mandatory: Erlingsson, Younan, Piessens, *Low-level software security by example*
    - https://lirias.kuleuven.be/retrieve/110131
  - Recommended: Szekeres, Payer, Wei, Song, *SoK: Eternal War in Memory*
    - https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf