



Universität Hamburg

# C Insecurities

## An overview

Secologic Treffen 11.07.2005

Martin Johns



Fachbereich Informatik

SVS – Sicherheit in Verteilten Systemen



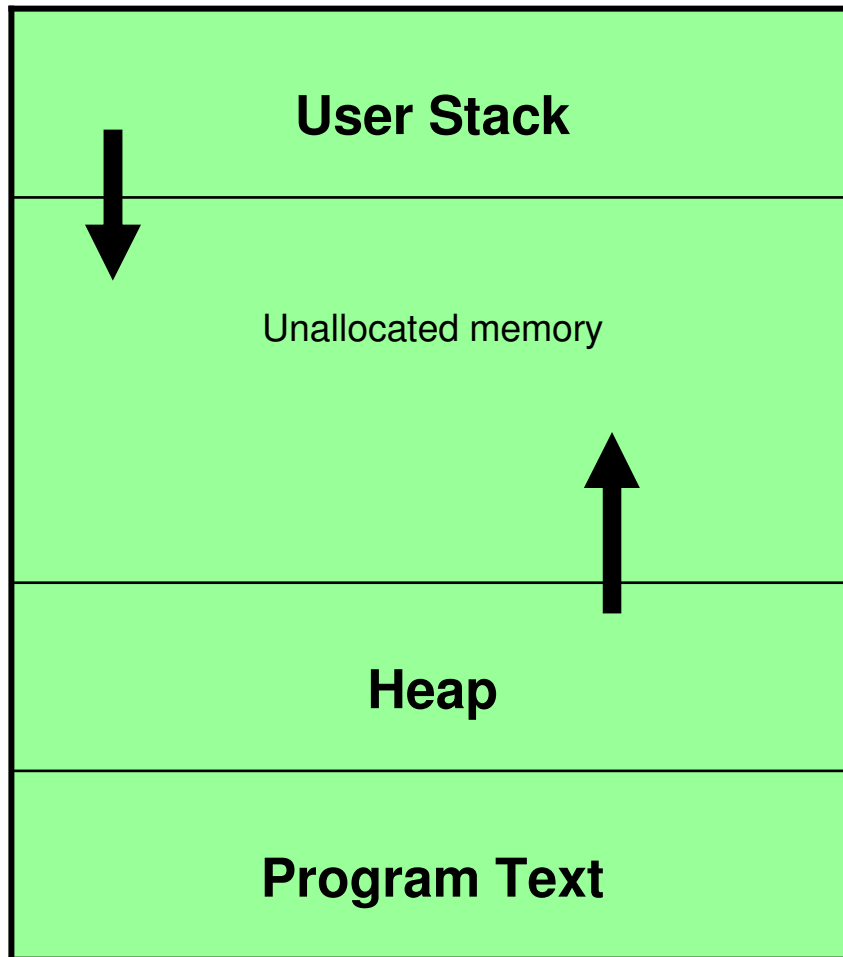
# Agenda

- **Basics: The Stack**
- **Buffer Overflows**
- **Format String Vulnerabilities**
- **Heap Overflows**

## DISCLAIMER

- **The following slides are geared towards:**
  - ◆ **The C language**
  - ◆ **The Intel x86 processor family**
  - ◆ **Unix-like and or Windows operating systems**
- **Depending on used language, processor or operating system some or all addressed topics may differ**

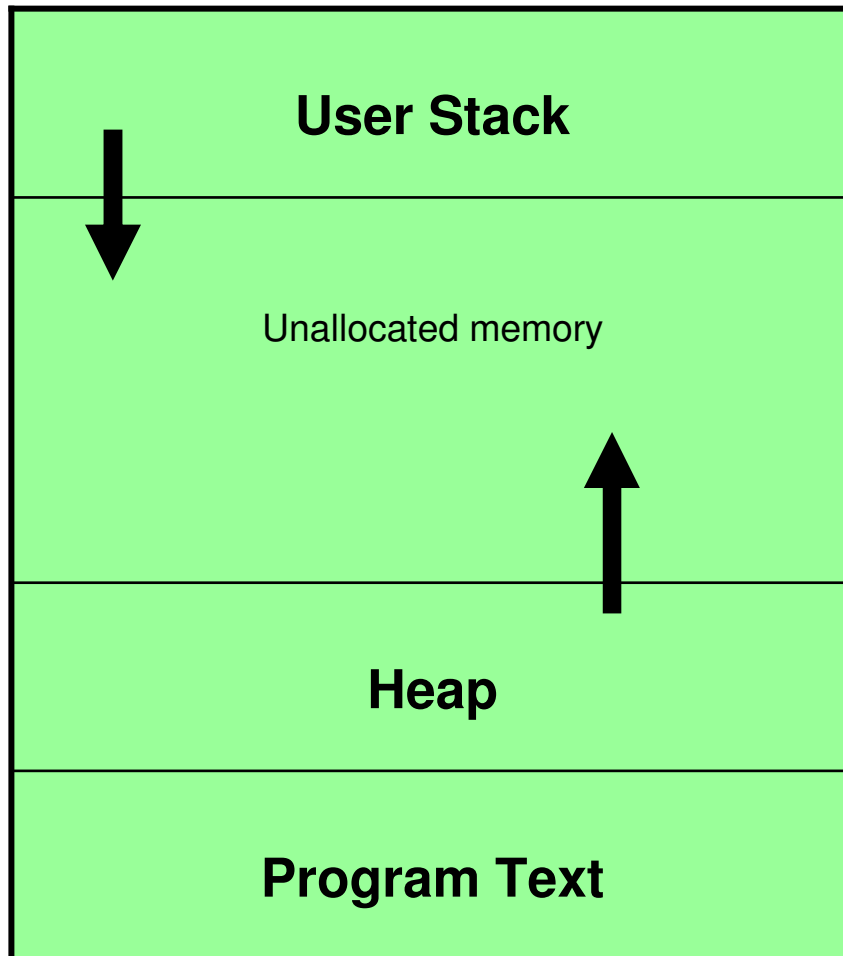
Higher addresses



Lower addresses

- Depending on the operating system the actual memory layout may somewhat differ
  - ◆ Usage of memory below the heap
  - ◆ Usage of memory above the stack
- Coherent memory is always read and written from lower addresses to higher addresses (not considering little/big endian)

Higher addresses



Lower addresses

- **The Stack grows from the higher to the lower addresses**
- **The Heap (initially) grows from the lower to the higher addresses**
- **The Stack is used to store data that is local to functions**
- **The Heap is used to store data that is persistent between function calls**
- **Stack allocation is implicit**  

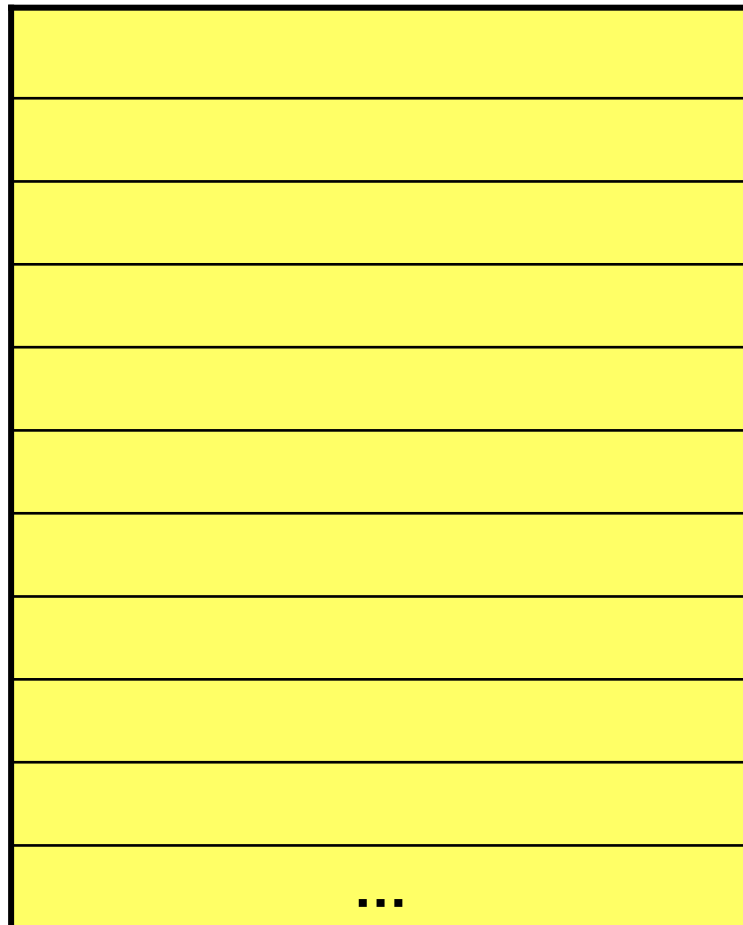
```
char test[42];
```
- **Heap allocation is (usually) explicit**  

```
char *test = malloc(42*sizeof(char));
```



# Basics: The Stack

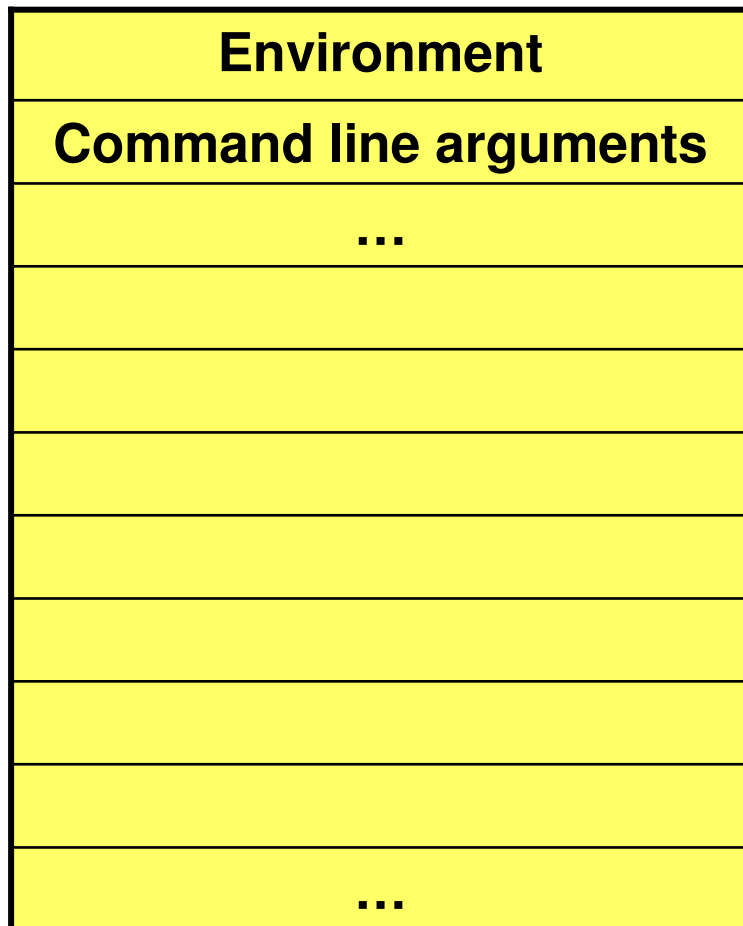
Higher addresses



Direction of growth

Lower addresses

Higher addresses

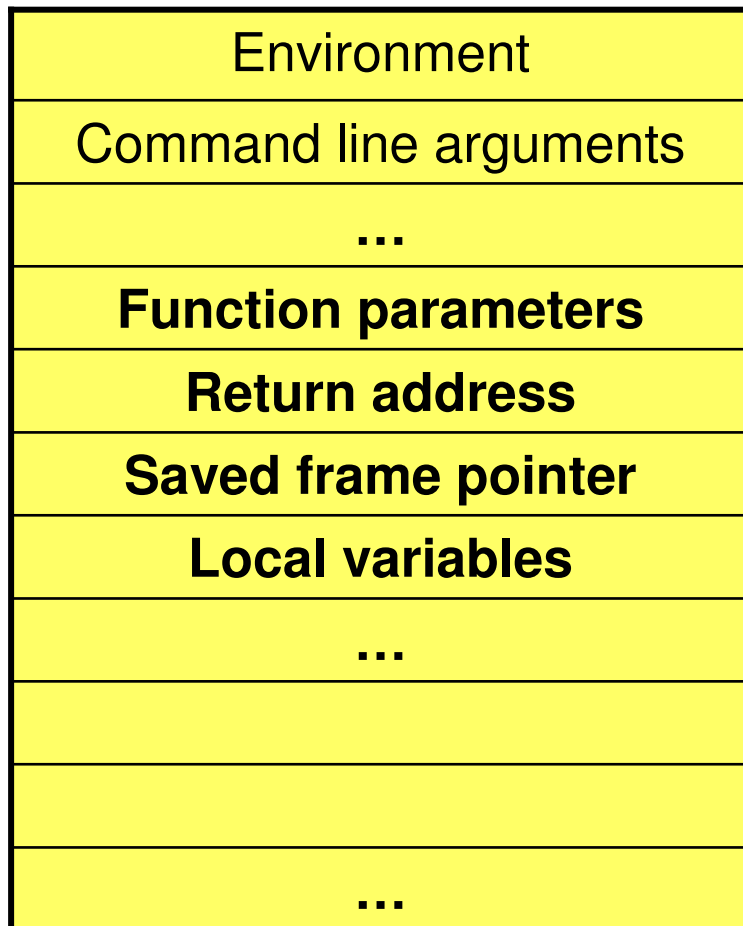


Lower addresses

The Stack is used for:

- The Environment
- Command line arguments

Higher addresses



Stack frame  
of the function

Lower addresses

## The Stack is used for:

- The Environment
- Command line arguments
- **Function parameters**
- **Return addresses**
- **Saved frame pointers**
- **Local variables**



## Basics: The Stack – Processor view (x86)

Several processor inherent constructs define the Stack

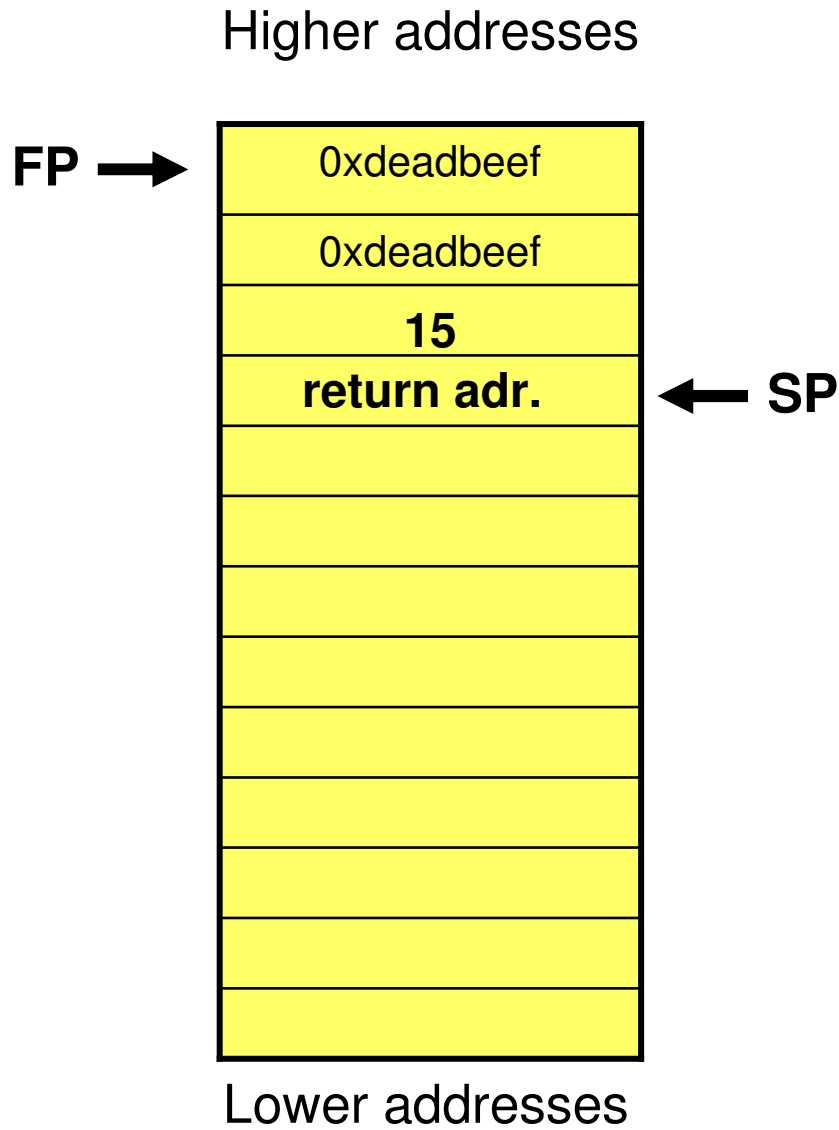
- **The register SP points to the top of the stack**
  - ◆ **Instructions that are stack-aware modify the value of SP automatically**
    - push, pop
    - call, return
    - (enter, leave)

### The Frame Pointer

- **The Frame Pointer is utilized by a function to access the local function context on the stack (function parameters, local variables)**
- **The register BP is used as Frame Pointer (assigned by the compiler)**
  - ◆ **Instructions to modify this register are (originally) generated by the compiler**
  - ◆ **Local variables and function parameters are referenced relatively to the Frame Pointer**
    - **Function parameters have positive offsets**
    - **Local variable have negative offsets**



# Basics: The function call



```
b = something(15);
```

- Move SP to reserve space for function parameter
- Write value of function parameter on the stack
- Call function (implicitly moves return address on the stack)







# **Buffer Overflows on the stack**

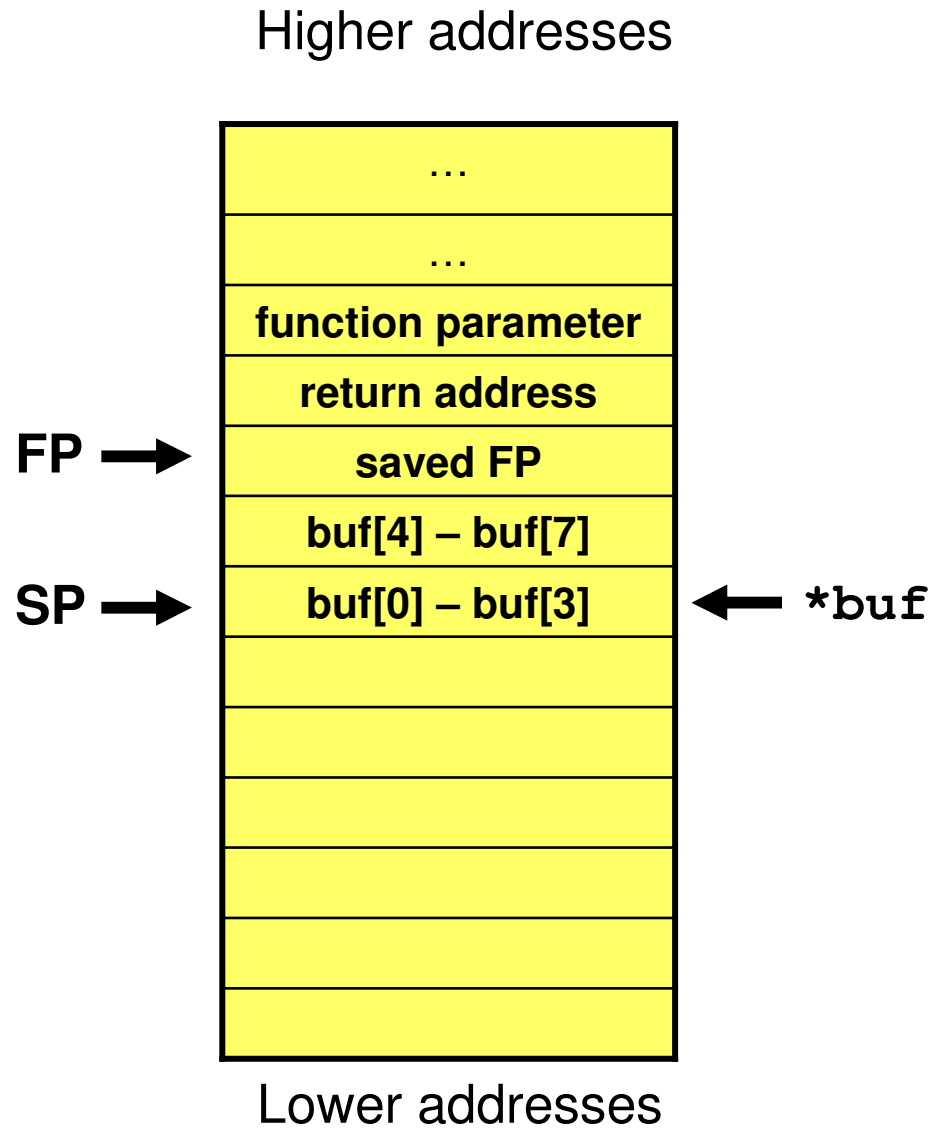


## Buffer Overflows

- **A C buffer is represented by a pointer variable which points to the first address of the buffer**
- **The programming language has no mechanism to enforce the buffer boundaries**
- **Therefore it lies in the responsibilities of the program to respect the buffer boundaries**
- **If a misbehaving program and ignores the buffer boundaries the adjacent memory regions may get overwritten**



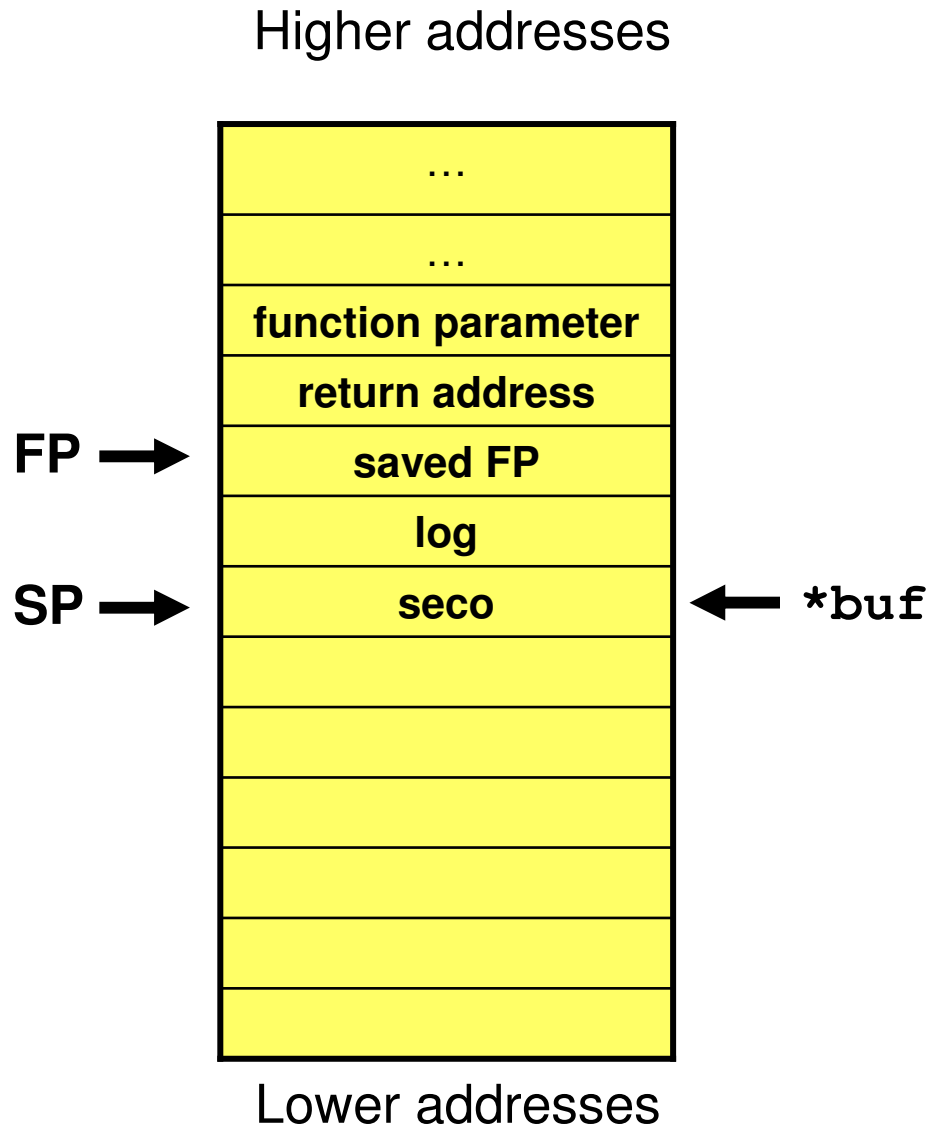
# Buffer Overflow on the stack



```
int something(int para){  
    char buf[8];  
    ...  
    strcpy(buf, "secolog");  
}
```



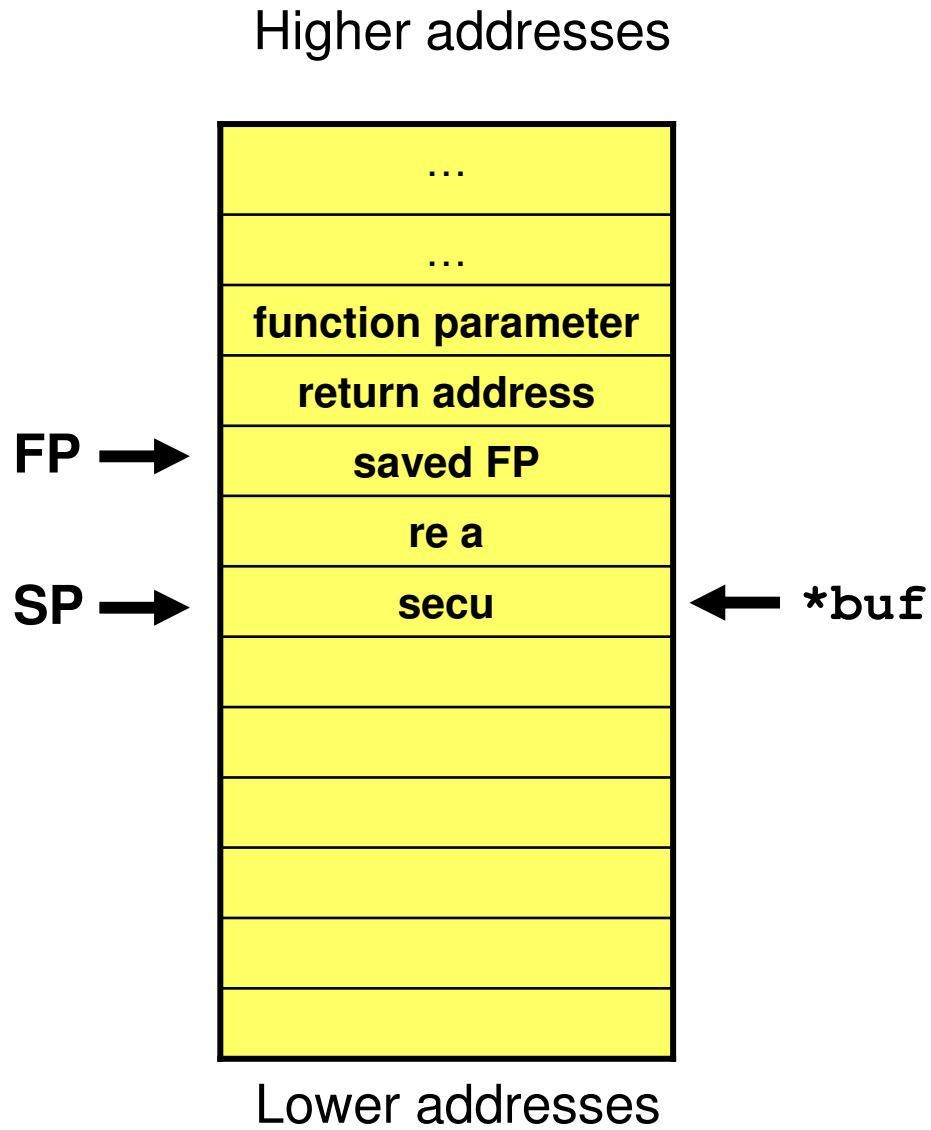
# Buffer Overflow on the stack



```
int something(int para){
    char buf[8];
    ...
    strcpy(buf, "secolog");
    ...
    strcpy(buf,
           "secure analysis");
}
```



# Buffer Overflow on the stack

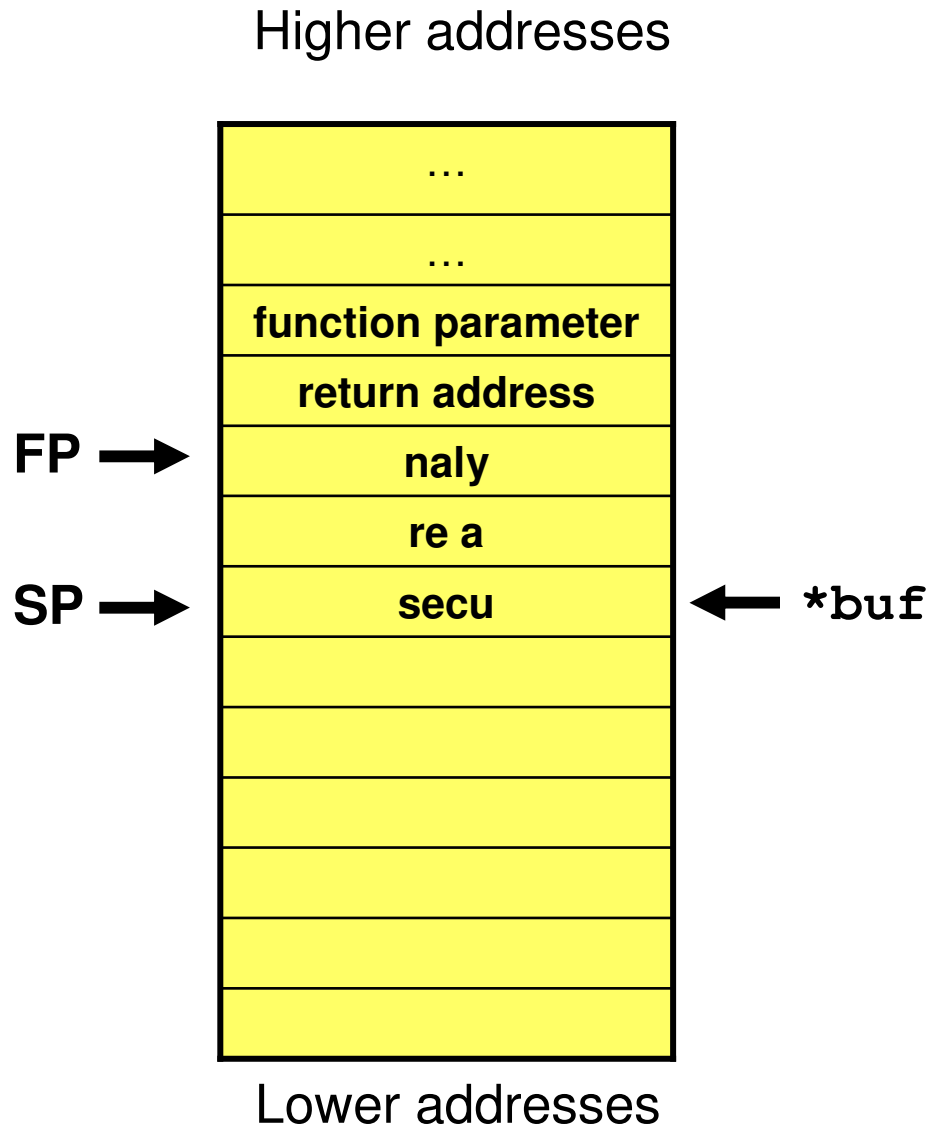


```
int something(int para){
    char buf[8];
    ...
    strcpy(buf, "secolog");
    ...
    strcpy(buf,
        "secure analysis");
}
```

- The saved Frame Pointer gets overwritten

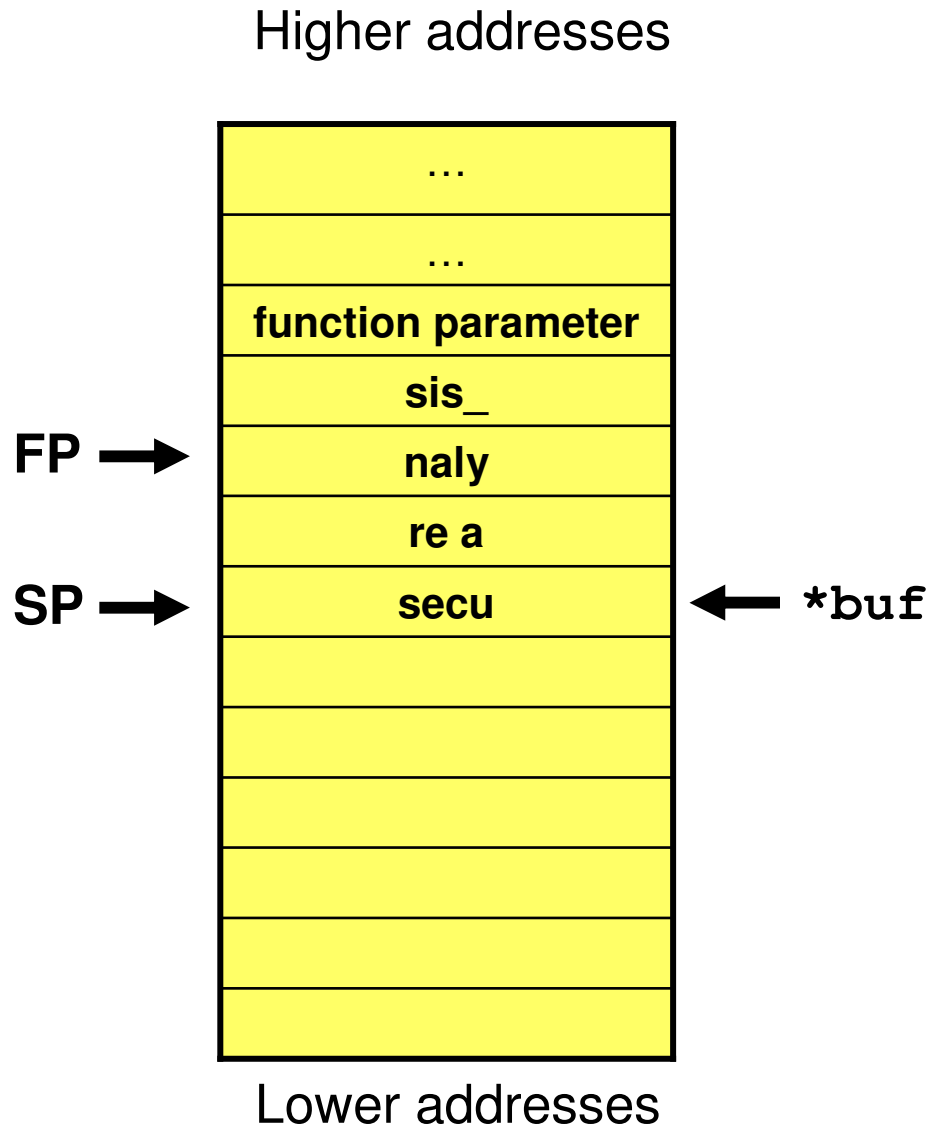


# Buffer Overflow on the stack



```
int something(int para){
    char buf[8];
    ...
    strcpy(buf, "secolog");
    ...
    strcpy(buf,
           "secure analysis");
}
```

- The saved Frame Pointer gets overwritten
- The return address gets overwritten



```
int something(int para){
    char buf[8];
    ...
    strcpy(buf, "secolog");
    ...
    strcpy(buf,
           "secure analysis");
}
```

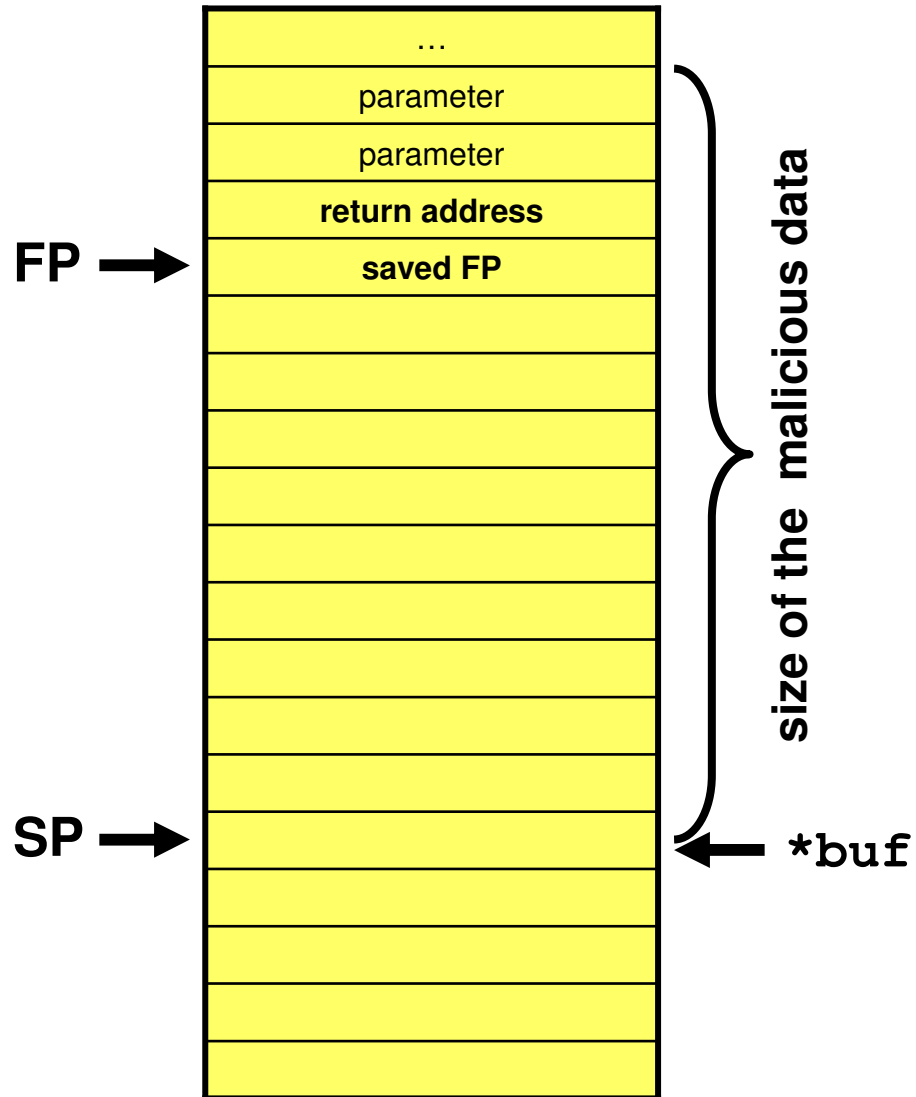
- The saved Frame Pointer gets overwritten
- The return address gets overwritten



## Exploiting a Buffer Overflow

**Using a stack based Buffer Overflow to hijack execution:**

- **Construct a buffer that contains executable code (the shellcode)**
- **Guess the memory address of the vulnerable buffer on the stack**
- **Pad the constructed shellcode**
  - ◆ **Before the shellcode: As many NOP instructions as possible**
  - ◆ **Behind the shellcode: The guessed address of the beginning of the shellcode (or the nops)**
  - ◆ **The constructed buffer has to be big enough to overwrite the return address on the stack**

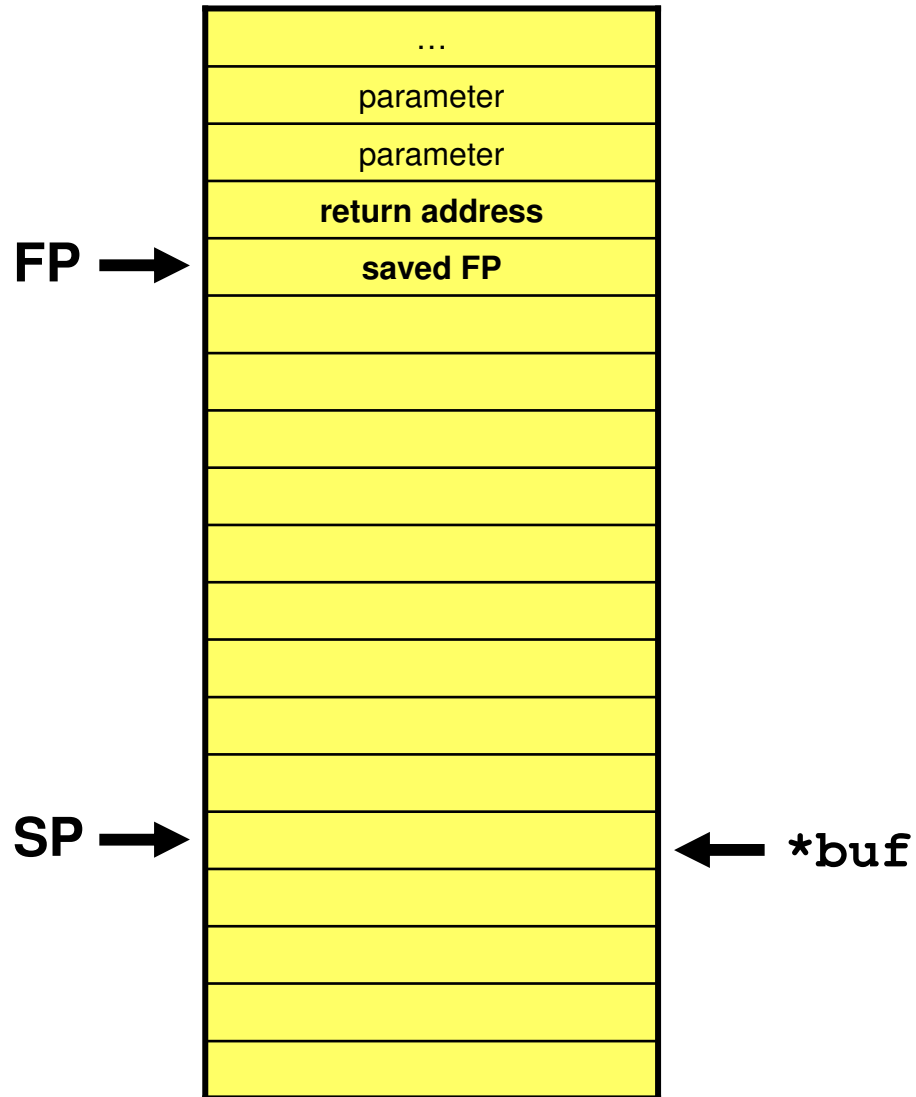


```
int something(int, int) {
    char buf[512];
    ...
    strcpy(buf, argc[1]);
}
```

- **argc[1] contains the constructed malicious data**
- **The malicious data is big enough to overflow the buffer and overwrite the return address**



# Buffer Overflow on the stack

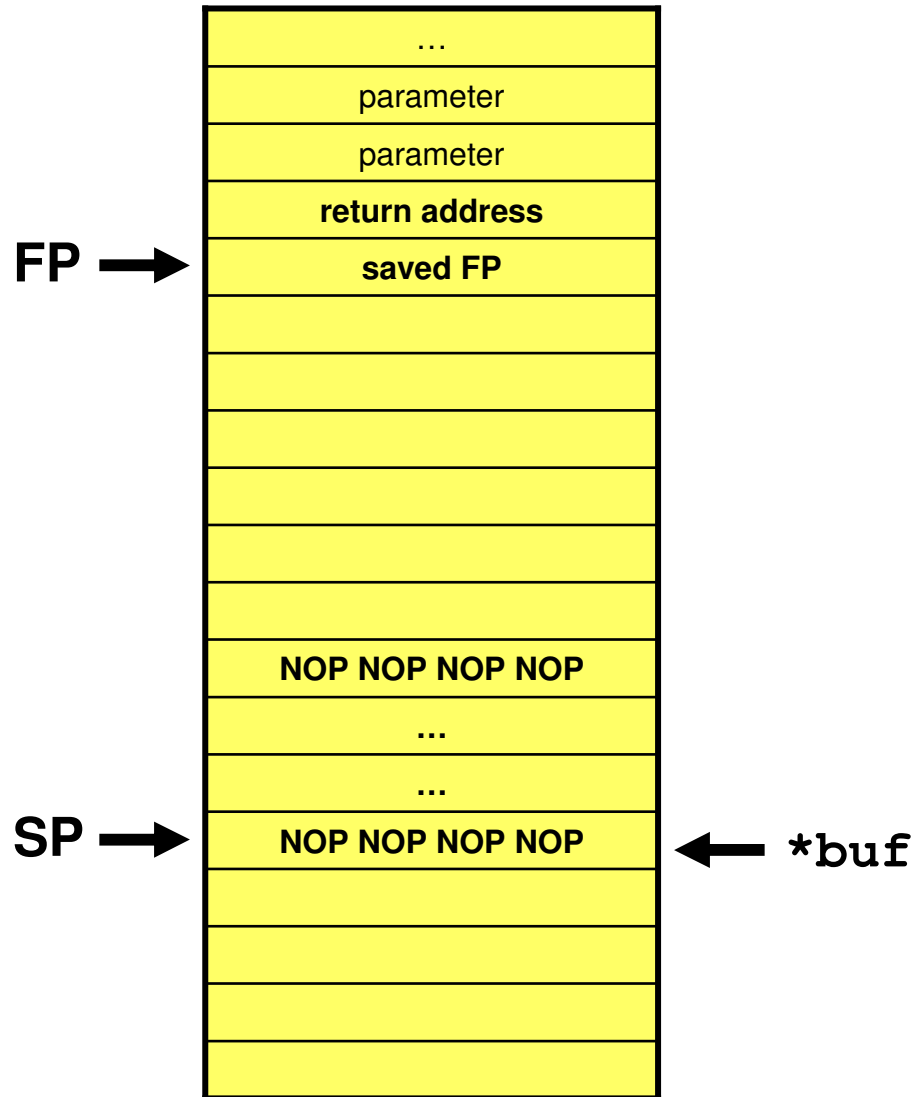


```
int something(int, int) {  
    char buf[512];  
    ...  
    strcpy(buf, argc[1]);  
}
```

- **argc[1] contains the constructed malicious data**
- **The data starts with the NOPs**



# Buffer Overflow on the stack

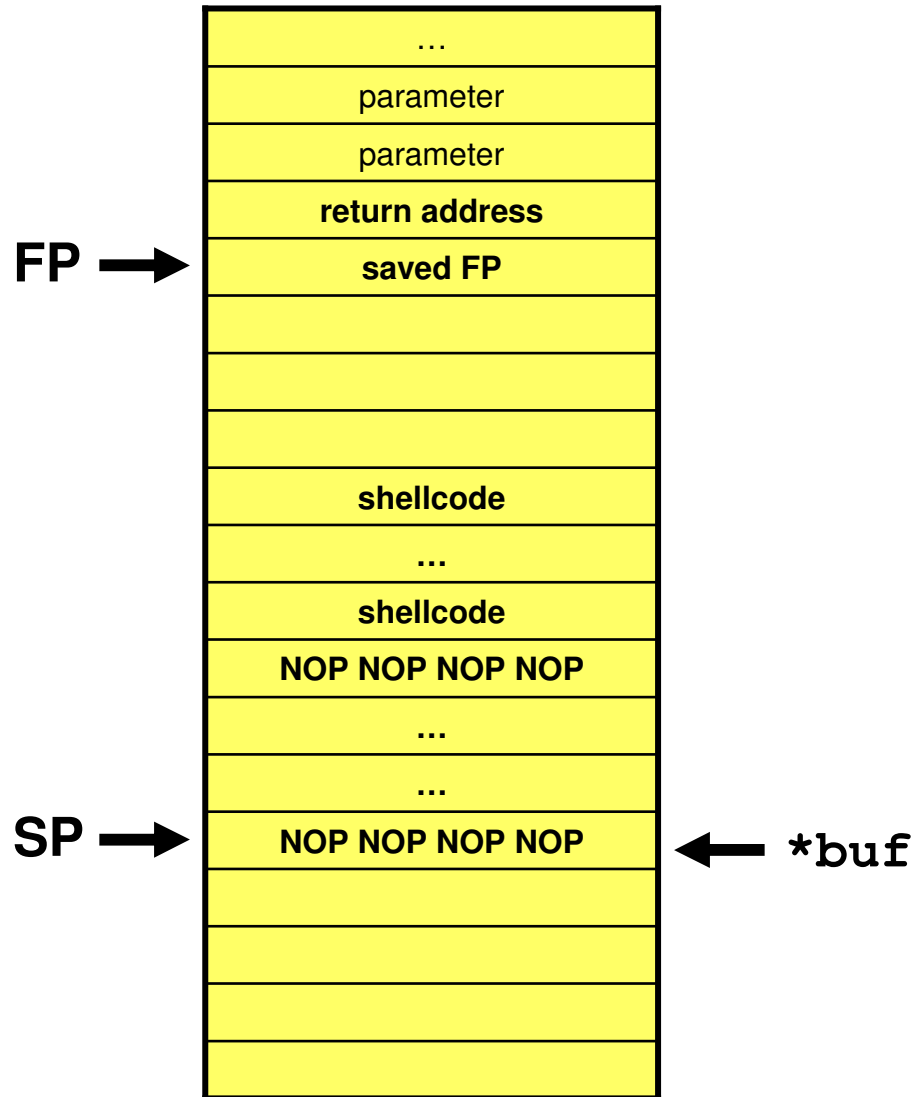


```
int something(int, int) {  
    char buf[512];  
    ...  
    strcpy(buf, argc[1]);  
}
```

- **argc[1] contains the constructed malicious data**
- **The data starts with the NOPs**
- **In the middle follows the Shellcode**



# Buffer Overflow on the stack

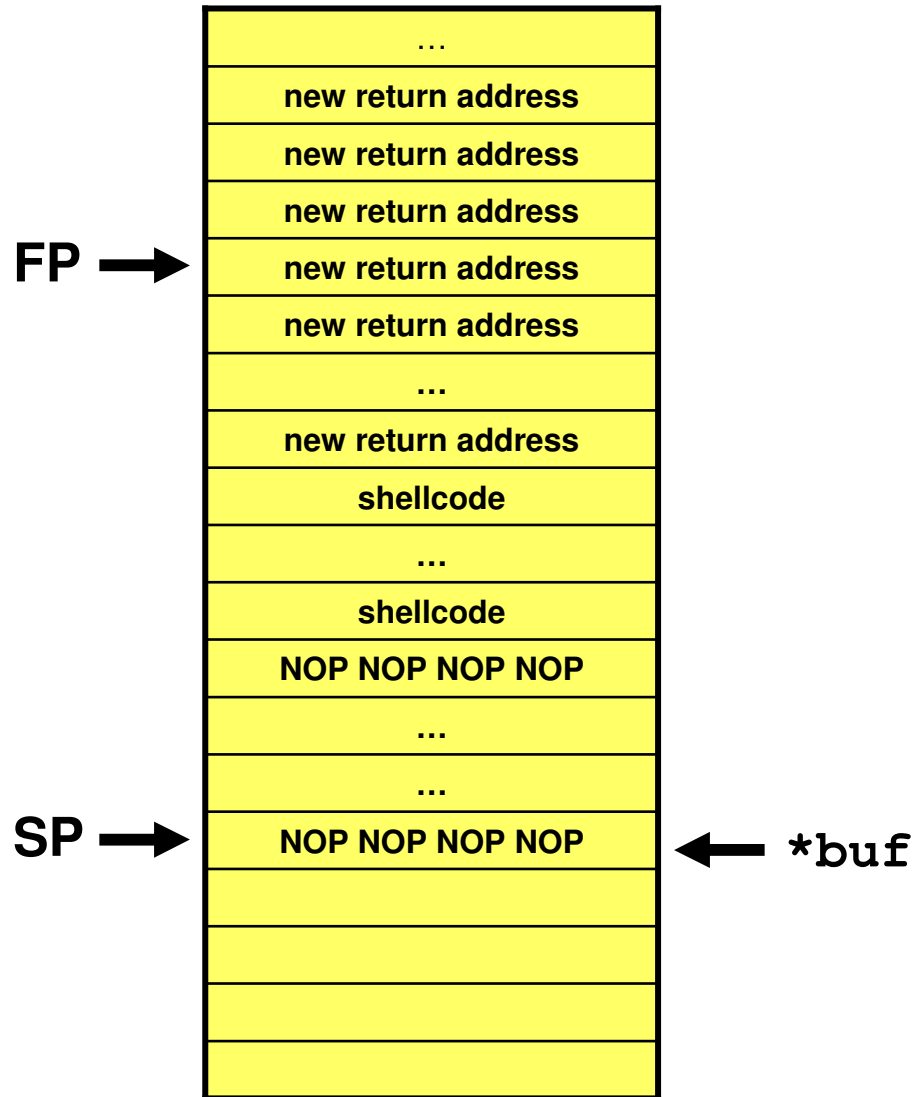


```
int something(int, int) {  
    char buf[512];  
    ...  
    strcpy(buf, argc[1]);  
}
```

- **argc[1] contains the constructed malicious data**
- **The data starts with the NOPs**
- **In the middle follows the Shellcode**
- **The remaining buffer is filled with the (guessed) malicious return address**



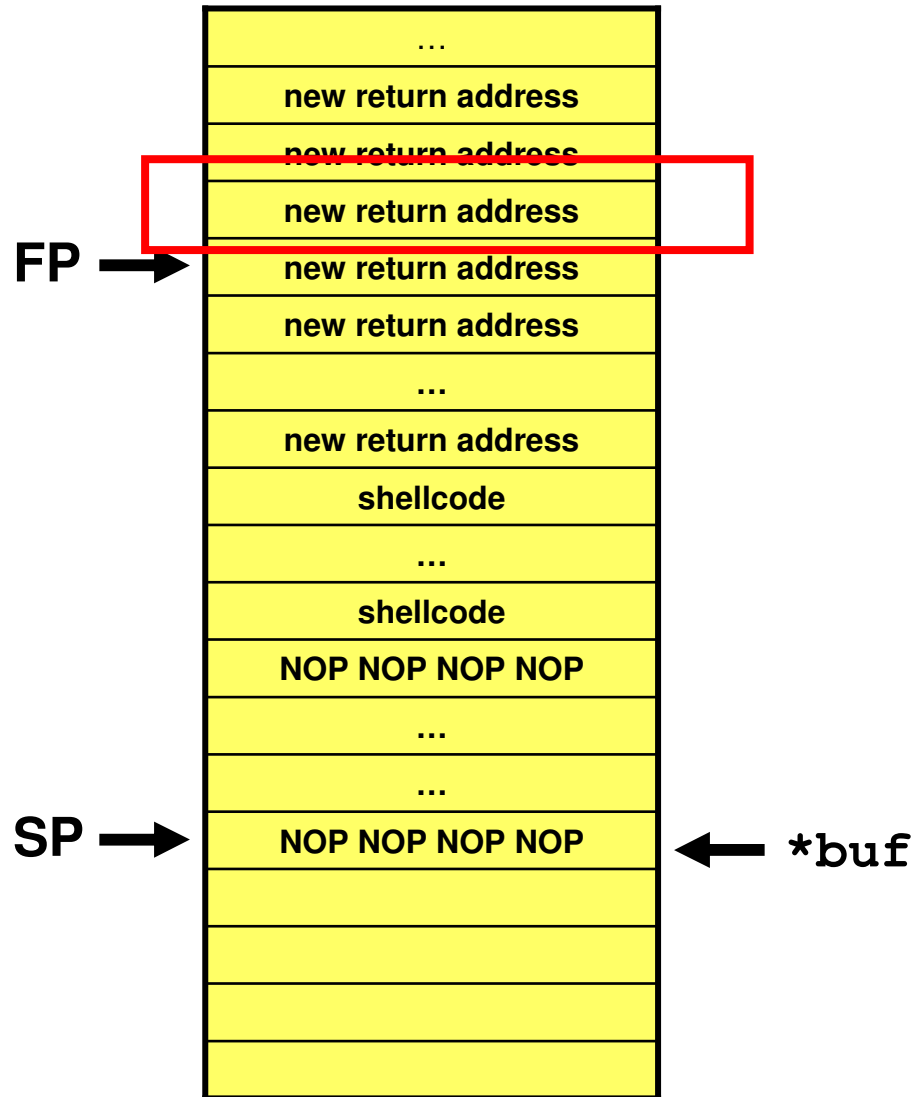
# Buffer Overflow on the stack



```
int something(int, int) {  
    char buf[512];  
    ...  
    strcpy(buf, argc[1]);  
}
```

- **argc[1] contains the constructed malicious data**
- **The data starts with the NOPs**
- **In the middle follows the Shellcode**
- **The remaining buffer is filled with the (guessed) malicious return address**

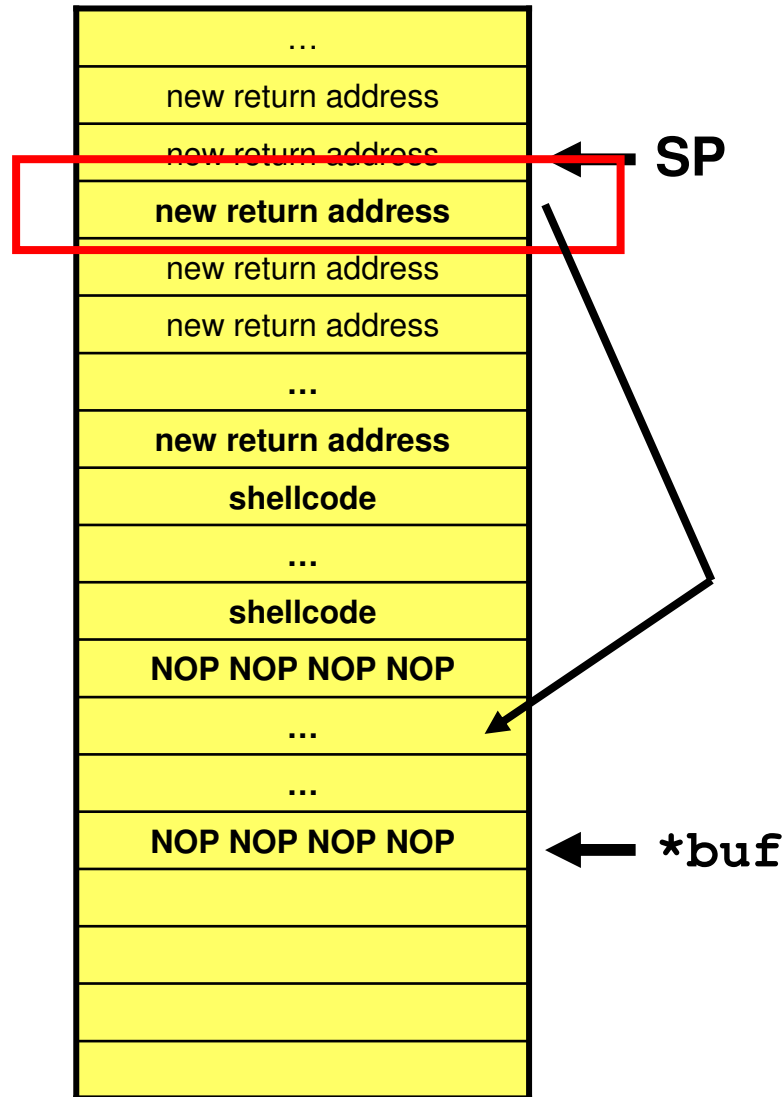




```
int something(int, int) {
    char buf[512];
    ...
    strcpy(buf, argc[1]);
    return(0);
}
```

- The Stack Pointer is moved to the Frame Pointer
- The old Frame Pointer is popped from the stack
- The old value was overwritten, the location of the FP is therefore meaningless
- The return address is popped from the stack into the Instruction Register (IR)

# Buffer Overflow on the stack



```
int something(int, int) {
    char buf[512];
    ...
    strcpy(buf, argc[1]);
    return(0);
}
```

- The new return address points to a location in the buffer
- The program executes the NOPs and the following shellcode



## Causes for Buffer Overflows

- **Library functions, that don't check buffer boundaries:**

`strcpy(), strcat(), sprintf(),...`

- **Library functions, that rely on the programmer to pass correct buffer boundaries:**

```
char buf1[4];
char buf2[8];
buf2 = "1234567";
strncpy(buf1, buf2, 8);
```

- **Careless pointer arithmetic:**

```
while(<some condition>){
    *buf1++ = *buf2++
}
```

**(cause of the Blaster Worm)**

- **Careless type casting of variables**



# Advanced Buffer Overflow techniques

- **Small buffers**
- **Trampolining**
- **Return-into-libc / arc injection**
- [Pointer manipulation]
- [Off By One]



## Exploiting a small buffer

- **Sometimes an overflowable buffer is too small to contain an executable shellcode**
- **In this situations the attacker can try to place the shellcode at a different location in the memory**
  - ◆ **Environment variables**
  - ◆ **Non vulnerable buffers on the stack**
  - ◆ **Buffer on the heap**
- **As long as the attacker is able to guess the memory location of these buffers, he can manipulate the return address accordingly**



## Trampolining

- In some situations the attacker is not able to determine (guess) the absolute address of the manipulated stack region
- Instead of modifying the return address to a location on the stack, the attacker points the return address to a location in the program code (the “*Trampoline*”)
- The reference location in the program code contains a sequence of instructions which enables involuntary a transfer of the control flow to the attackers code on the stack
- Example: JMP to a value that is kept in a register
- The location of those regions in the program code are usually static and therefore easy guessable
- This exploitation technique is especially popular on W32 operating systems (apparently kernel32.dll contains a couple of usable code sequences)

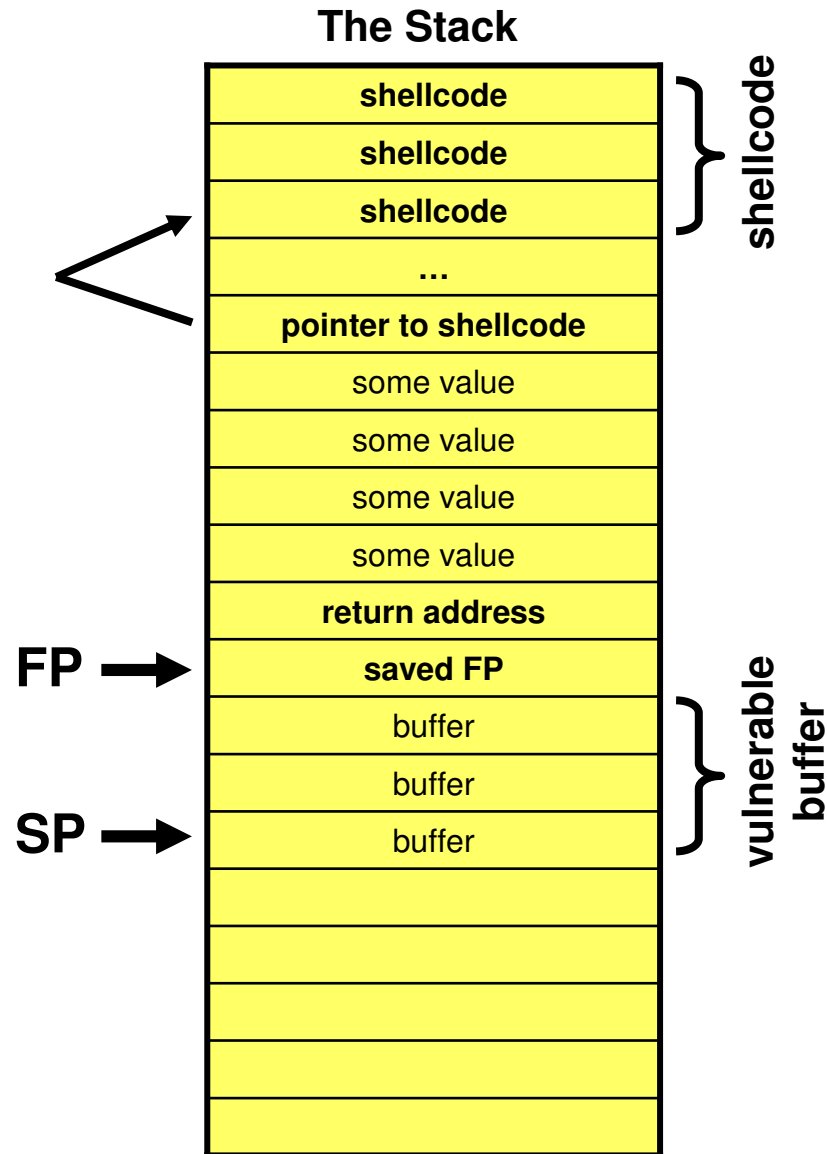


## Trampolining: “pop pop return”

### The situation:

- The attacker is able to place shellcode on the stack
- The attacker is also able to overflow the return address
- The stack contains a pointer to the shellcode (e.g. a function parameter)
- The program code contains a sequence of pop-instructions followed by a return instruction

# Trampolining: “pop pop return” (II)



- The stack contains the shellcode
- The stack contains a pointer to the shellcode
- The attacker is able to overflow a buffer

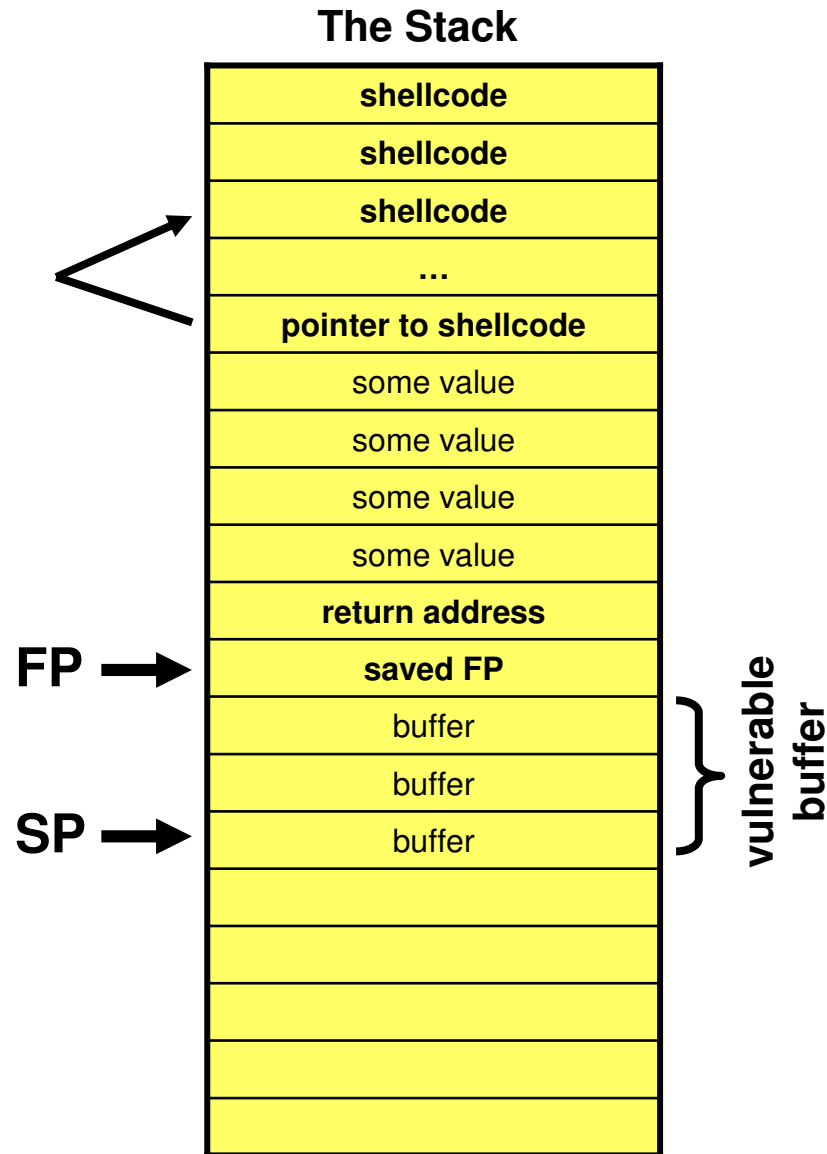
**The program code**

```

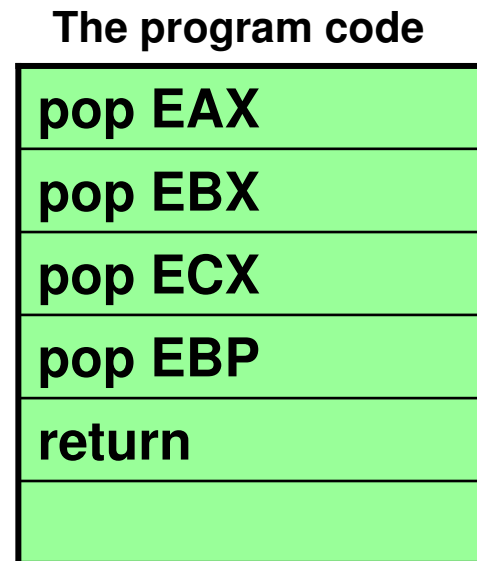
pop EAX
pop EBX
pop ECX
pop EBP
return

```

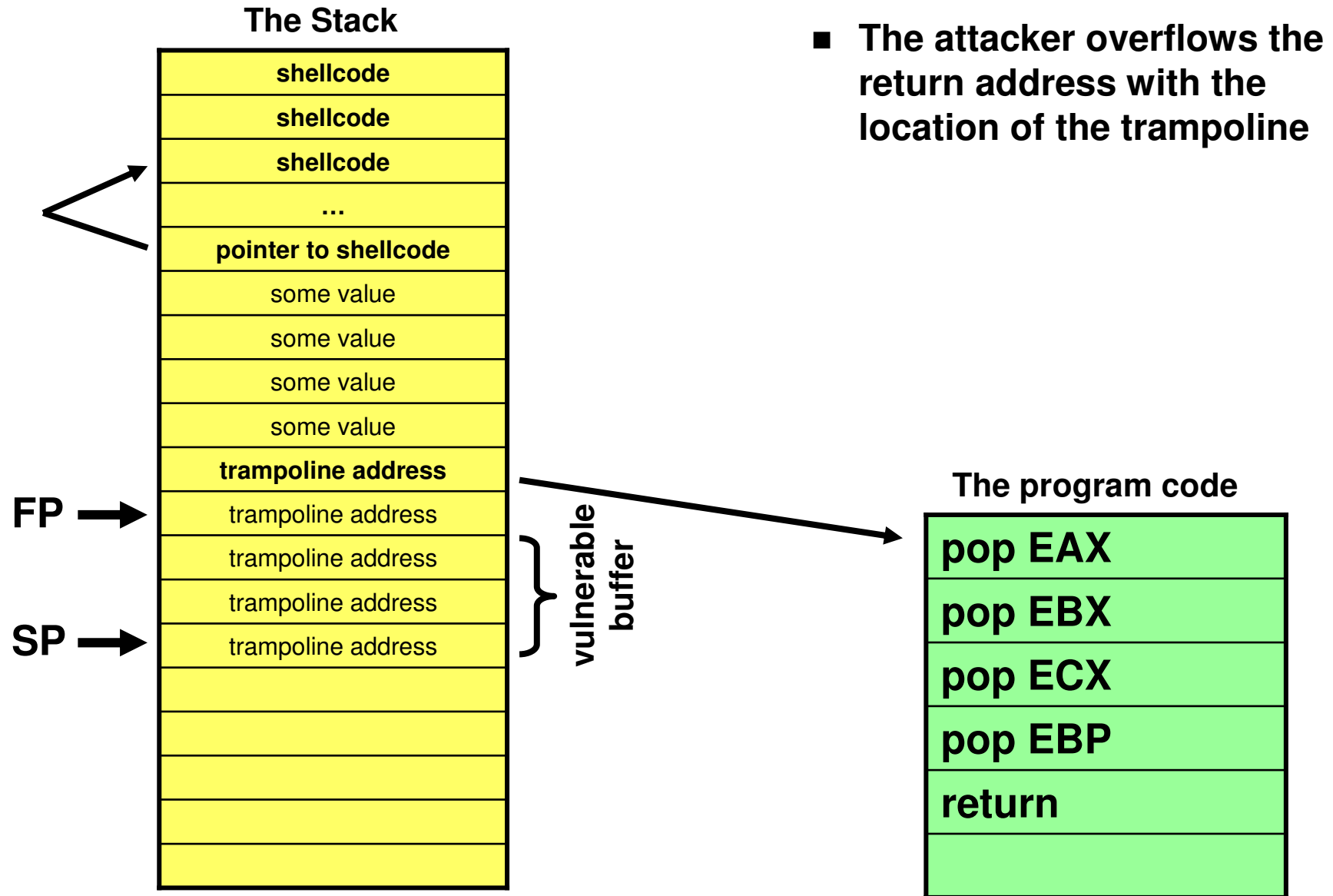
# Trampolining: “pop pop return” (II)



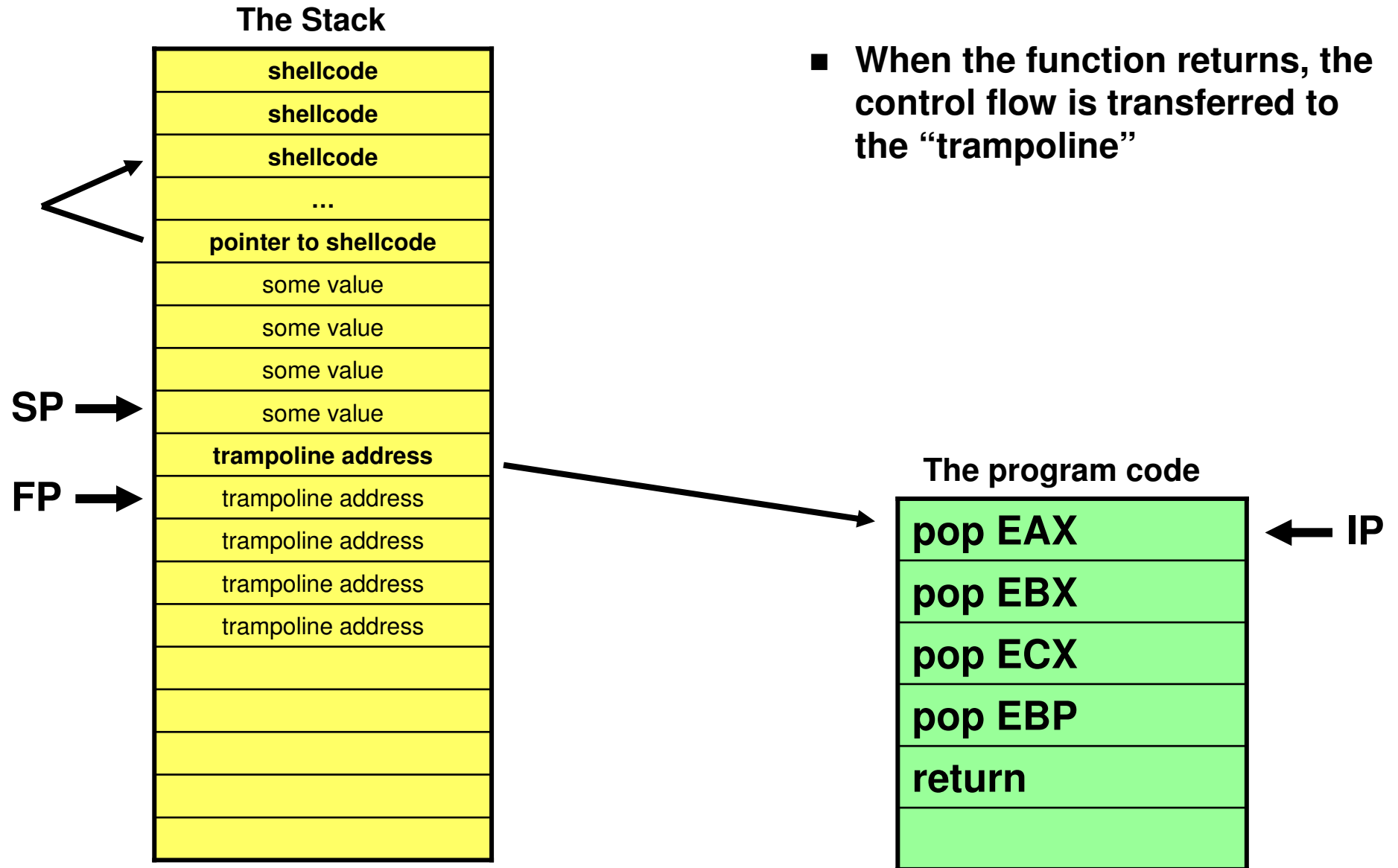
- The attacker overflows the return address with the location of the trampoline



# Trampolining: “pop pop return” (II)

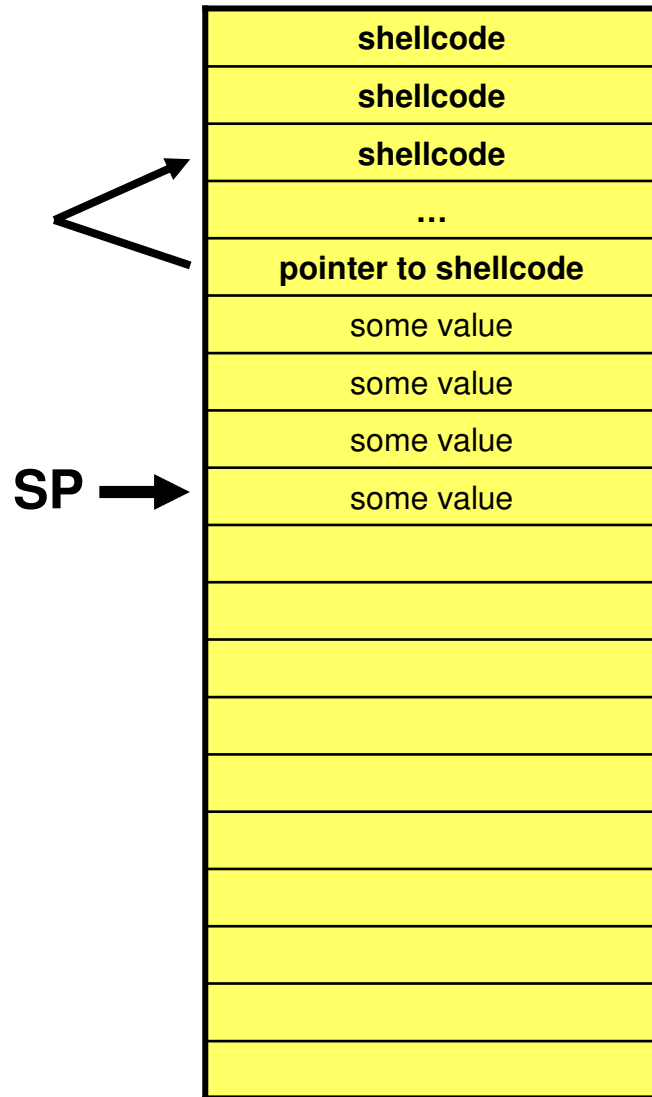


# Trampolining: “pop pop return” (II)



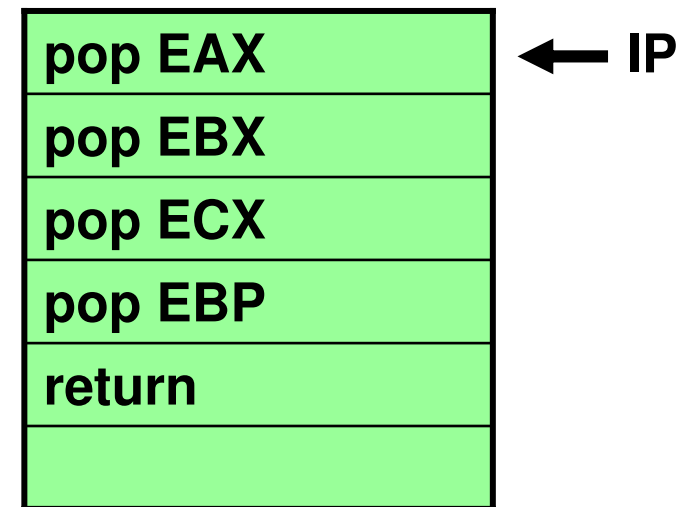
# Trampolining: “pop pop return” (II)

The Stack



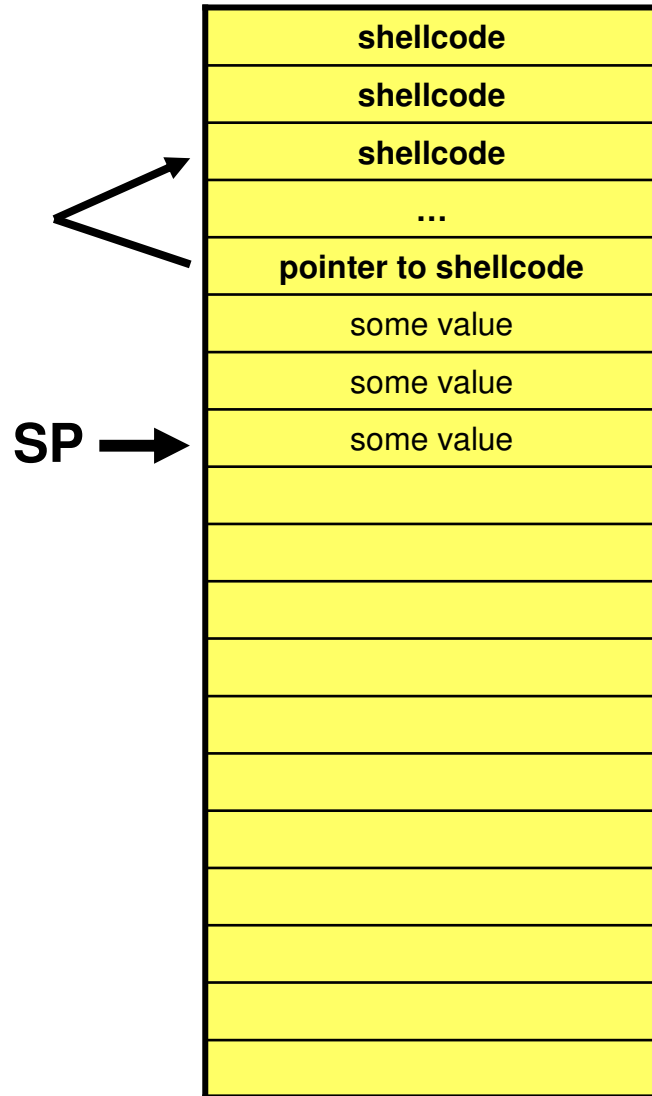
- When the function returns, the control flow is transferred to the “trampoline”
- The trampoline code removes unwanted data from the stack

The program code



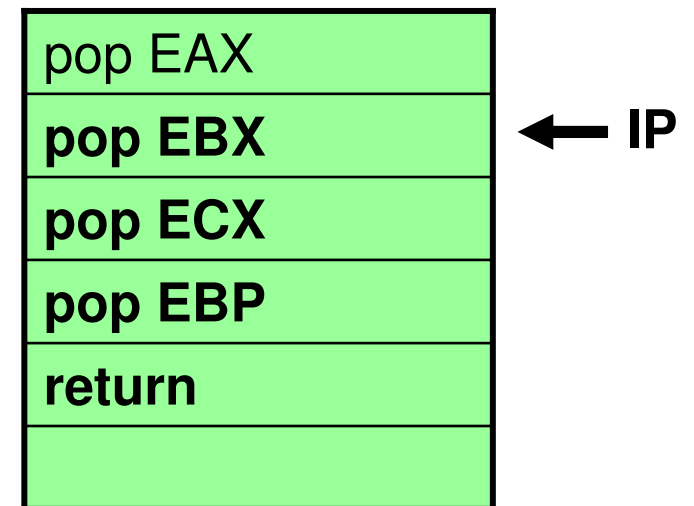
# Trampolining: “pop pop return” (II)

The Stack



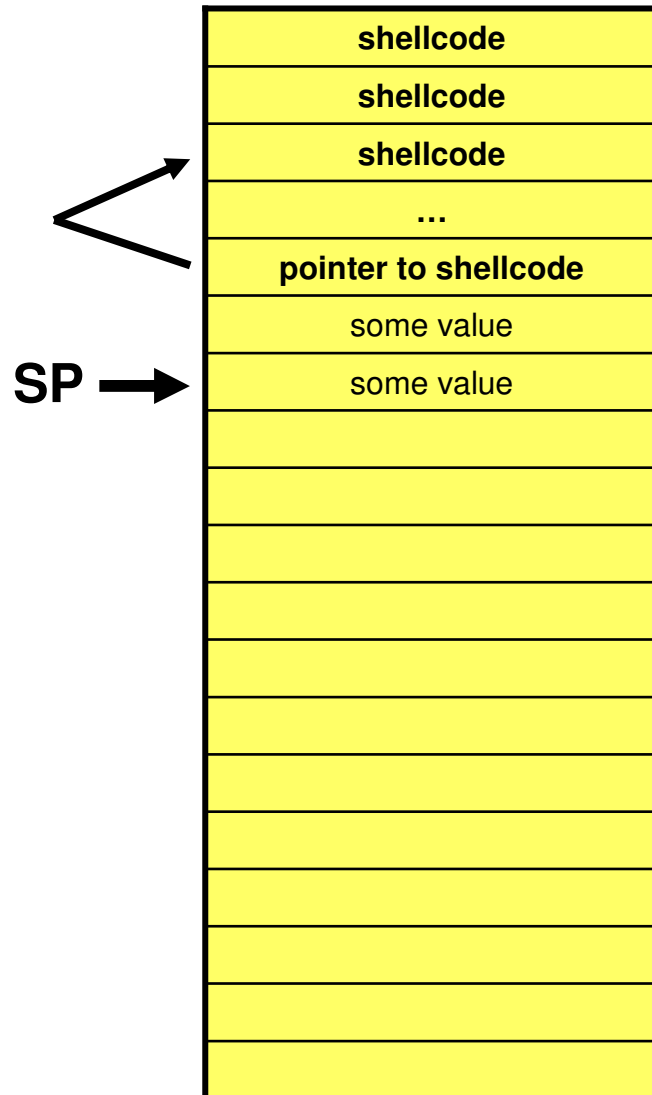
- When the function returns, the control flow is transferred to the “trampoline”
- The trampoline code removes unwanted data from the stack

The program code



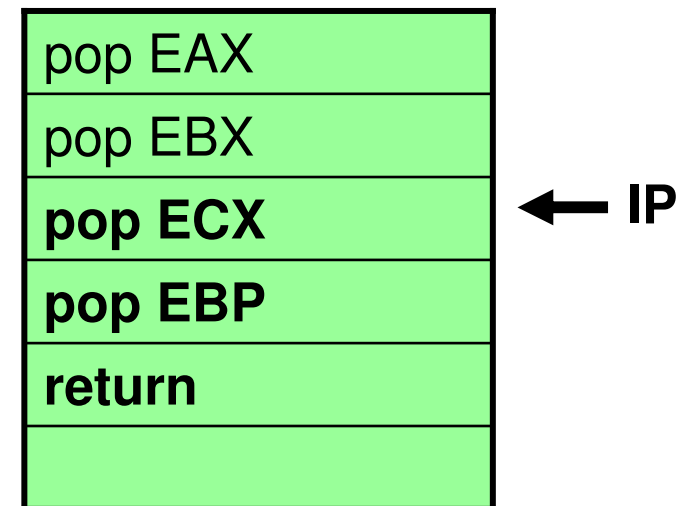
# Trampolining: “pop pop return” (II)

The Stack

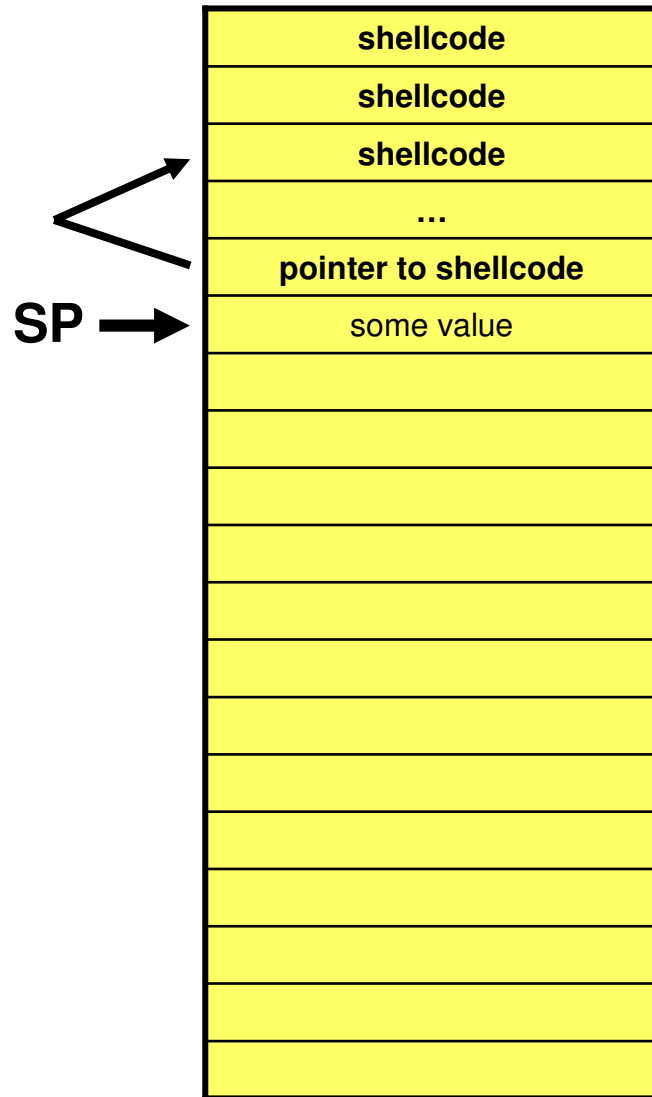


- When the function returns, the control flow is transferred to the “trampoline”
- The trampoline code removes unwanted data from the stack

The program code

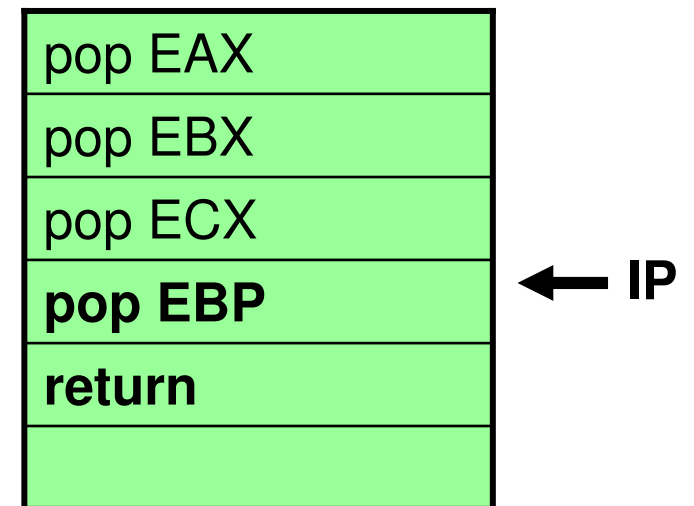


## The Stack

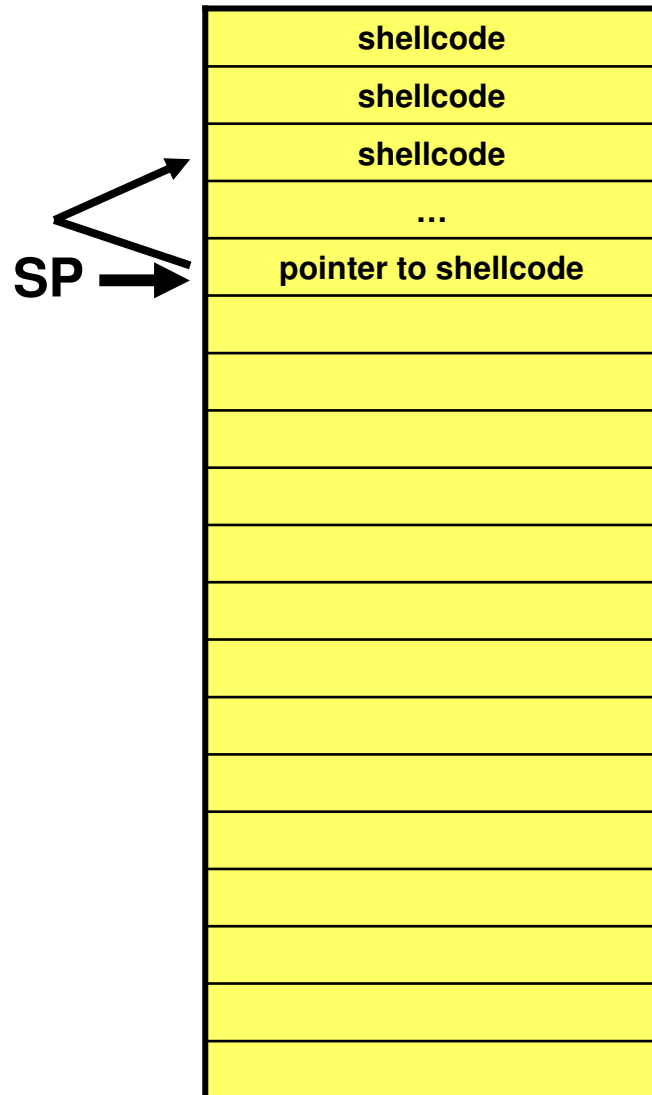


- When the function returns, the control flow is transferred to the “trampoline”
- The trampoline code removes unwanted data from the stack

## The program code

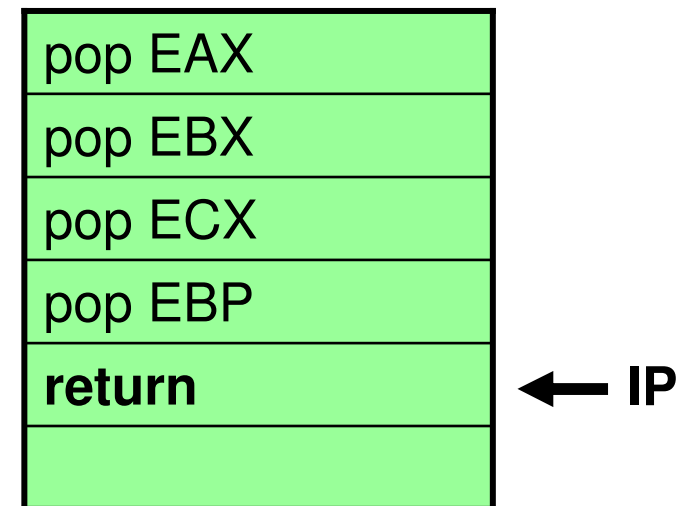


## The Stack



- When the function returns, the control flow is transferred to the “trampoline”
- The trampoline code removes unwanted data from the stack
- The return instruction uses the pointer to the shellcode

## The program code





## Trampolining (III)

### Other “Trampolines”

- **Using a pointer to a buffer on the Heap**
- **Instead of “pop pop return”:** Jump to an address stored in a register



## Return-into-libc / Arc Injection

- Depending on the system configuration, it may be prohibited to execute code on the stack
- Instead of placing malicious code on the stack, the control flow is directed to a location in the program code
  - ◆ Before the modification of the control flow the stack is manipulated, to contain a valid stack frame
  - ◆ This stack frame will trick the program code to perform as the attacker wishes
  - ◆ A popular target is the libc
- These kind of attacks are sometimes also calls “Arc Injection”
  - ◆ A “standard” Buffer Overflow adds a new node to the control flow graph
  - ◆ This kind of attack adds a new ark



## Return-into-libc / Arc Injection (II)

### Example: system()

- High level libc system call, which spawns a new process

```
void system(char* arg) {  
    ...  
}
```

```
system("/bin/sh");
```

- **system()** expects a stack frame which contains a pointer to a C-string as only function parameter
- This string contains the path to the binary (and the arguments)
- A successful return to libC attack would construct a stack frame which provides the described conditions



# **Format String Vulnerabilities**



## Format String Vulnerabilities

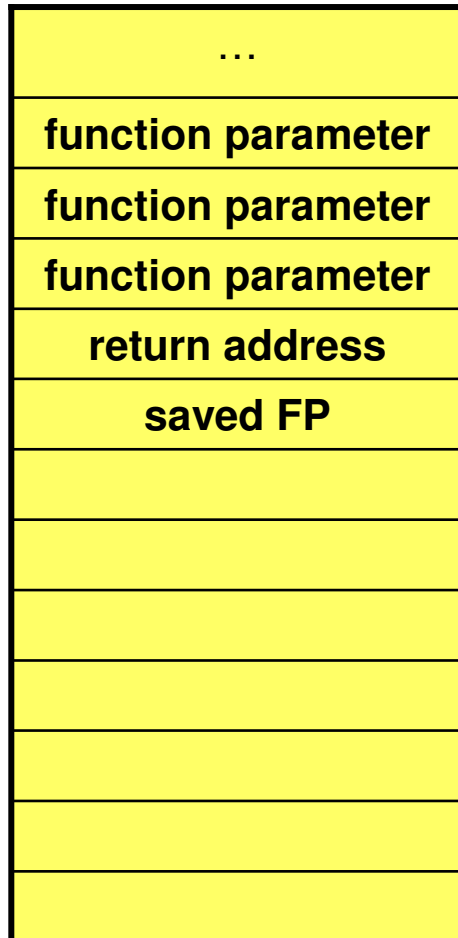
- **ANSI C contains a mechanism to allow a function to use a dynamic number of arguments**
- **Usually the first argument of the function contains the information, how many more arguments were passed to the function**
  - ◆ **Example: The first argument of `printf()` is the format string, which determines the number and type of the additional arguments:**

```
printf("%s %d", charpointer, int);
```
- **What happens, when the number of expected arguments and the number of passed arguments differ?**



## Format String Vulnerabilities (II)

Higher addresses



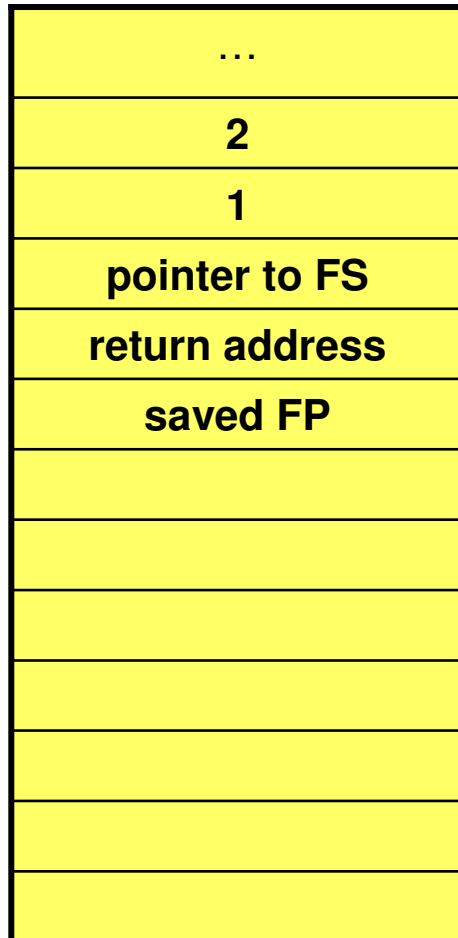
Lower addresses

```
a = 1;  
b = 2;  
printf ("%d %d %d", a, b);
```



## Format String Vulnerabilities (II)

Higher addresses



Lower addresses

```
a = 1;  
b = 2;  
printf ("%d %d %d", a, b);
```

- The first function argument is a pointer to the format string
- The function determines the number of arguments
- The first referenced argument is the value 1
- The second referenced argument is the value 2
- **The third referenced argument is ???**



## Format String Vulnerabilities (III)

### The printf() function family

- **printf() uses format specifiers to identify the number and types of passed values**
  - ◆ **%i, %d, %u, %x: Integer or char arguments**  
The value is directly printed
  - ◆ **%p: Pointer argument**  
The value is directly printed
  - ◆ **%s Pointer argument**  
The value is interpreted as an char pointer
- **Standard format specifiers are processed sequentially**
- **Direct argument access**
  - ◆ **It is possible to access an argument directly**
  - ◆ **%5\$d accesses the 5th argument and interprets it as an `int`**
  - ◆ **A malicious format string is therefore able to directly access every location on the stack**



## Format String Vulnerabilities (IV)

### Writing to memory

- `printf()` is able to write the amount of printed bytes to a given memory location
- To achieve this, the format specifier `%n` is used

- **Example:**

```
int i;  
printf("Hello world\n%n", &i);  
printf("Last output: %d bytes\n", i);
```

### Scenario:

- The attacker is able to control (parts of) the format string
- The attacker is able to store data on the stack
- The attacker is able to overwrite arbitrary memory locations



## Format String Vulnerabilities (V)

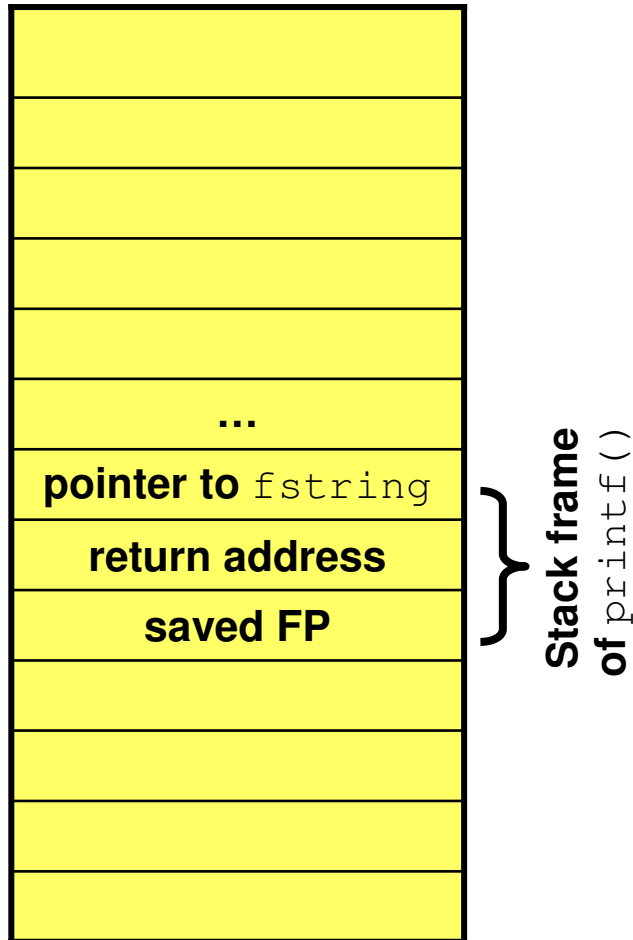
### Possible attack targets

- Return addresses on the stack
- Internal pointers, function pointer, C++-specific structures like VTABLE entries
- Overwriting a NULL terminator in some string to create a possible buffer overflow
- Changing arbitrary data in memory



# Format String Vulnerabilities (VI)

Higher addresses

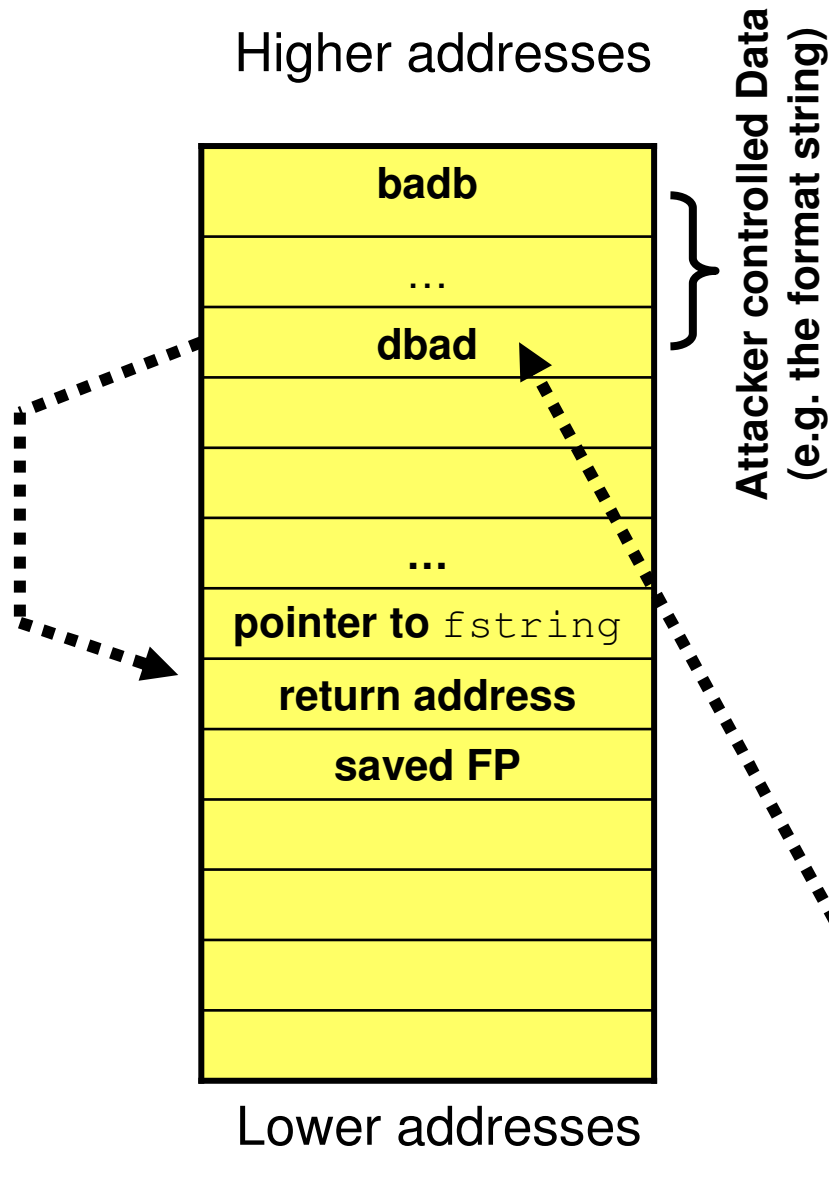


Lower addresses

```
printf (fstring);
```

- The attacker is able to control the content of `fstring`
- The attacker was able to place data on the stack

```
fstring = "...%12$n"
```



```
printf (fstring);
```

- The attacker is able to control the content of `fstring`
- The attacker was able to place data on the stack
- This data contains a pointer to the return address
- The malicious format string references this pointer
- The attacker is able to overwrite the return address
- The value which is written is controlled by the length of the format string

```
fstring = "...%12$n"
```

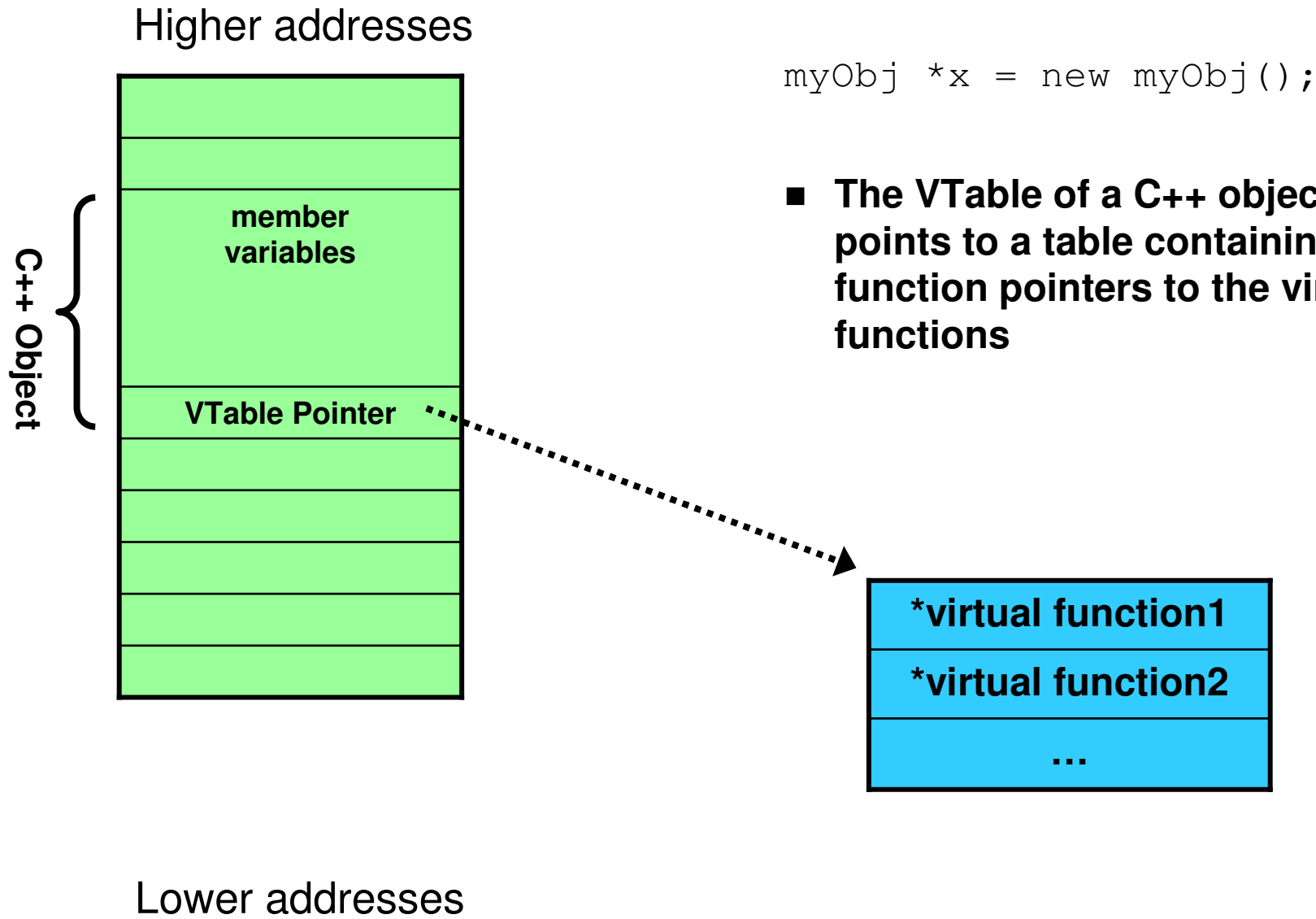


# Heap Overflows



# Heap Overflows

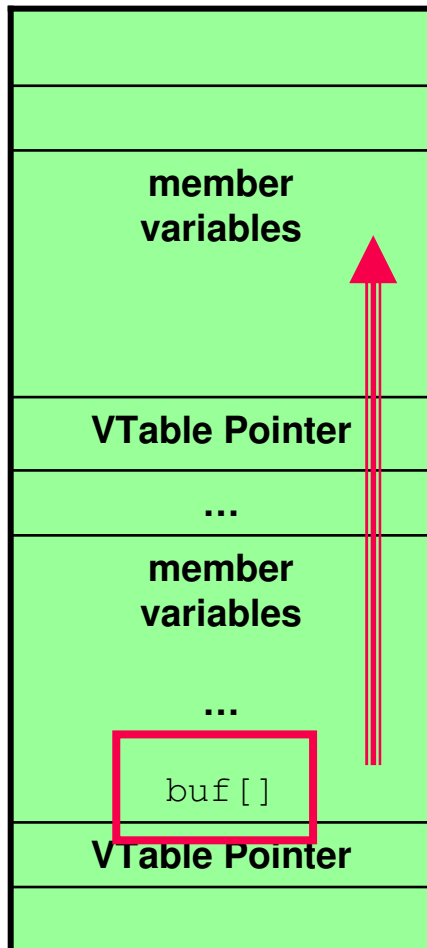
- **The causes for heap overflows are the same as the reasons for buffer overflows on the stack**
  - ◆ **Unsafe library functions**
  - ◆ **Careless pointer arithmetic**
- **The exploitation of heap overflows is fundamentally different**
  - ◆ **The heap contains no return addresses that could be overwritten**
- **Possible attacks**
  - ◆ **Overwriting of data structures**
    - **Function pointers**
    - **VTable entries (C++)**
  - ◆ **Manipulating control data**



```
myObj *x = new myObj();
```

- The VTable of a C++ object points to a table containing function pointers to the virtual functions

Higher addresses

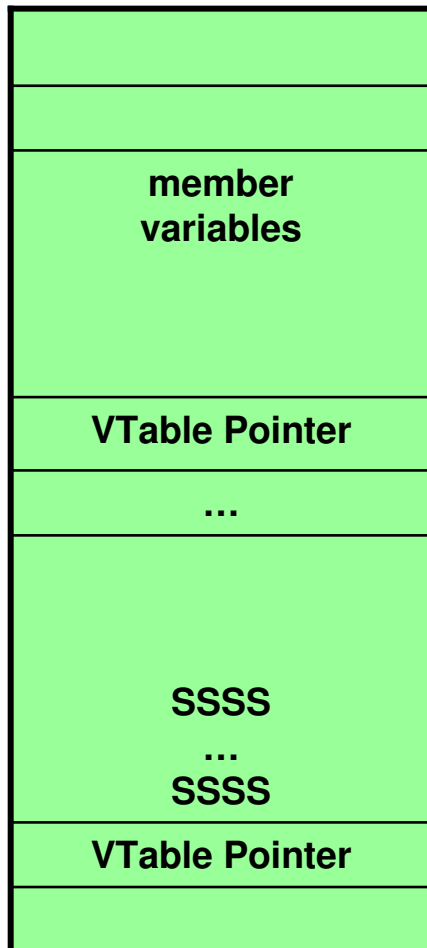


Lower addresses

**The scenario:**

- The attacker is able to overflow a buffer on the heap
- The malicious data:
  - ◆ The shellcode

Higher addresses

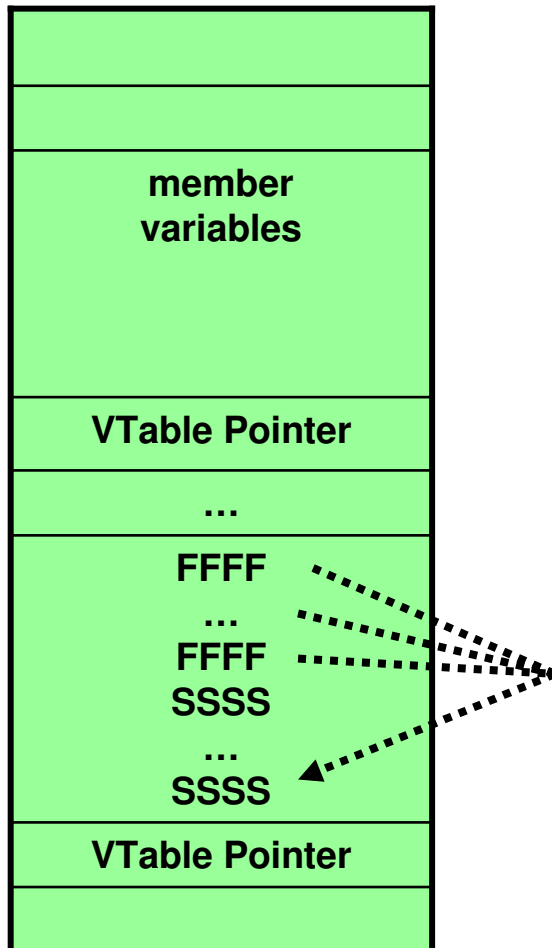


Lower addresses

**The scenario:**

- The attacker is able to overflow a buffer on the heap
- The malicious data:
  - ◆ The shellcode
  - ◆ Fake VTable-entries pointing to the shellcode

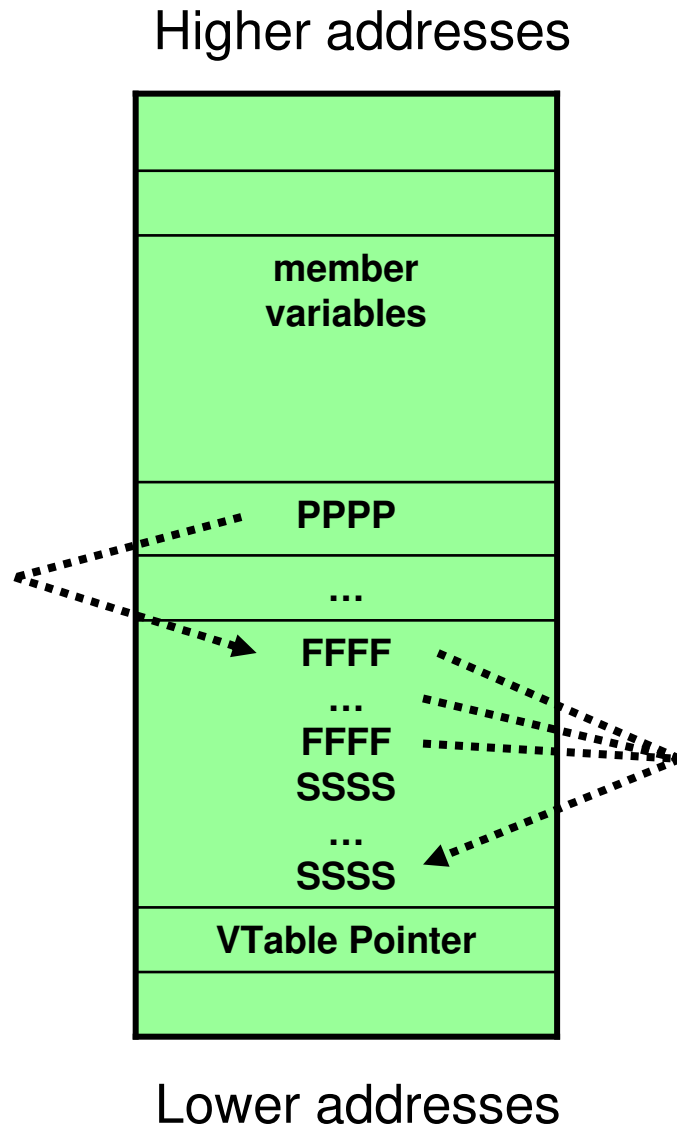
Higher addresses



Lower addresses

## The scenario:

- The attacker is able to overflow a buffer on the heap
- The malicious data:
  - ◆ The shellcode
  - ◆ Fake VTable-entries pointing to the shellcode
  - ◆ Pointer to the fake VTable replacing the original VTable pointer



## The scenario:

- The attacker is able to overflow a buffer on the heap
- The malicious data:
  - ◆ The shellcode
  - ◆ Fake VTable-entries pointing to the shellcode
  - ◆ Pointer to the fake VTable replacing the original VTable pointer
- On execution of one of the objects virtual functions the shellcode is called

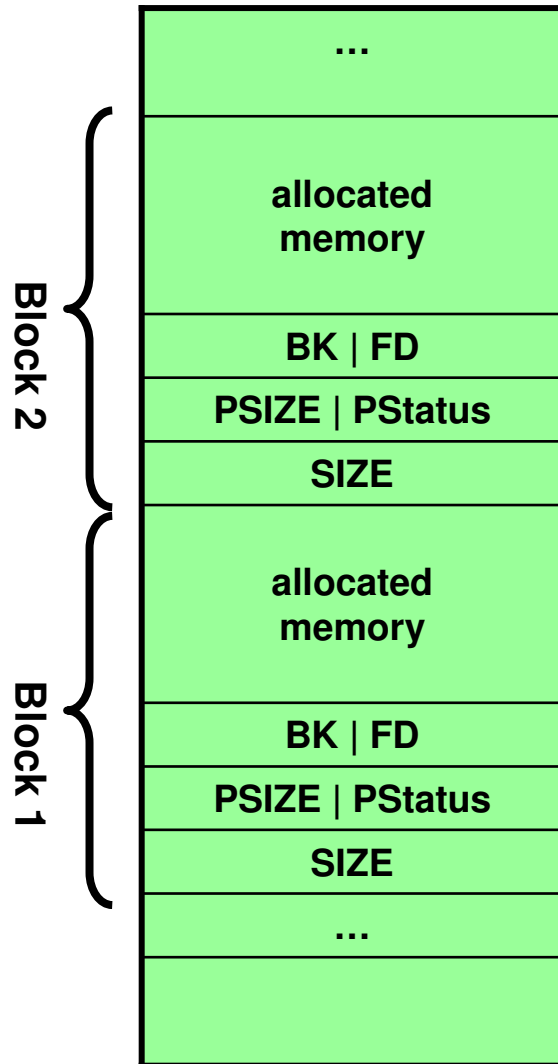


## Manipulating control data

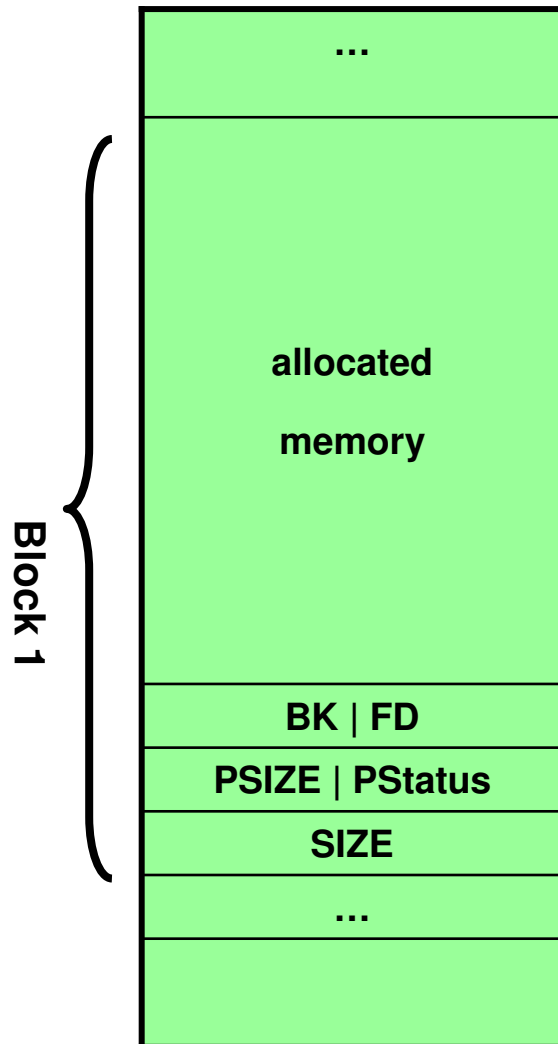
- C programs rarely store function pointers on the heap
- Most heap implementations store control information next to the allocated memory
- An attack on this control information is therefore implementation depended

### Example: Doug Lea `malloc()` (Linux)

- Free memory blocks are administered in doubly linked lists
  - ◆ These lists are called “bins”
  - ◆ Every bin links to blocks of certain sizes (8, 64, 512, 4096,...)
- Every allocated memory block begins with a header of control data
  - ◆ Size of the block (SIZE)
  - ◆ Size of the previous block (PSIZE)
  - ◆ Status of the previous block (PStatus)
  - ◆ Pointer to pervious block in bin (BK)
  - ◆ Pointer to next block in bin (FD)



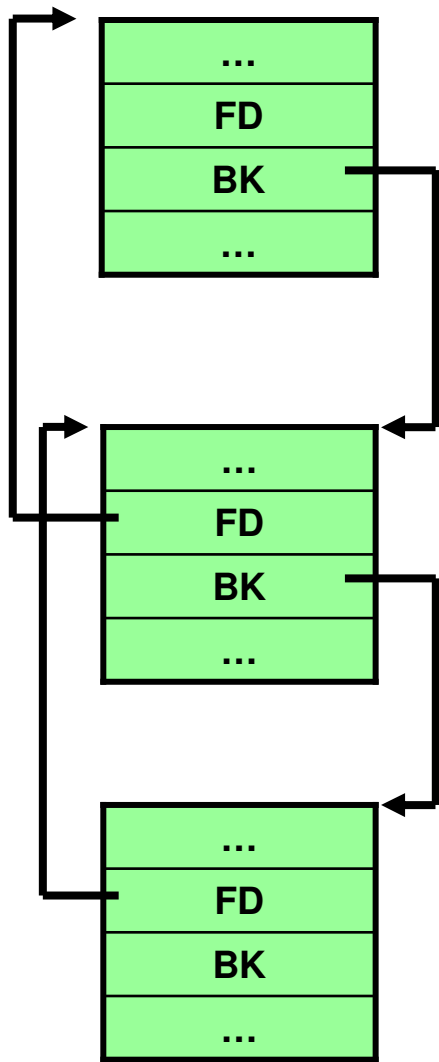
- **Note: BK and FD point to the previous/next block in the bin – not to the neighbors on the heap**
- **If two neighboring blocks are freed, their memory is united**

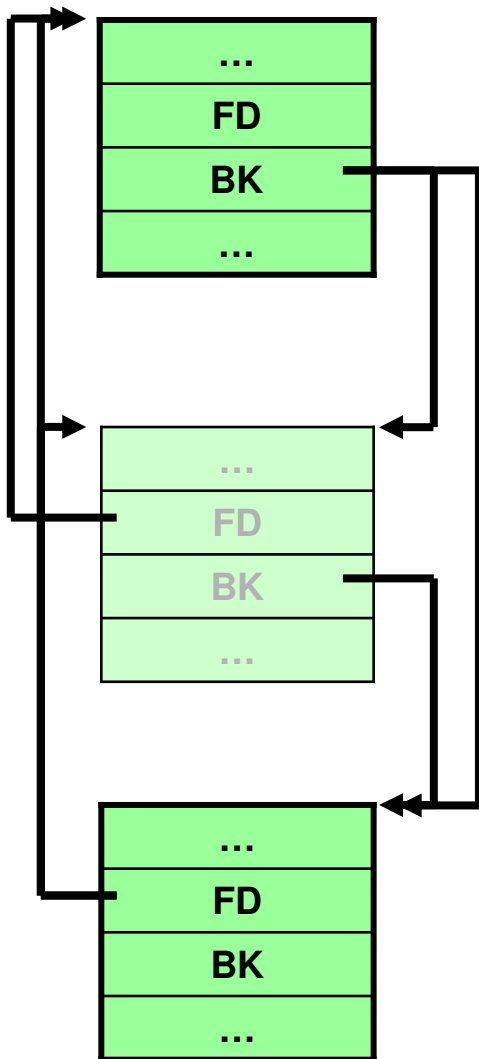


- **Note: BK and FD point to the previous/next block in the bin – not to the neighbors on the heap**
- **If two neighboring blocks are freed, their memory is united**
- **Block 2 is removed from its bin**



# Overwriting C++ VTables (II)





- When a block is removed from its bin, the pointers of the list are modified accordingly

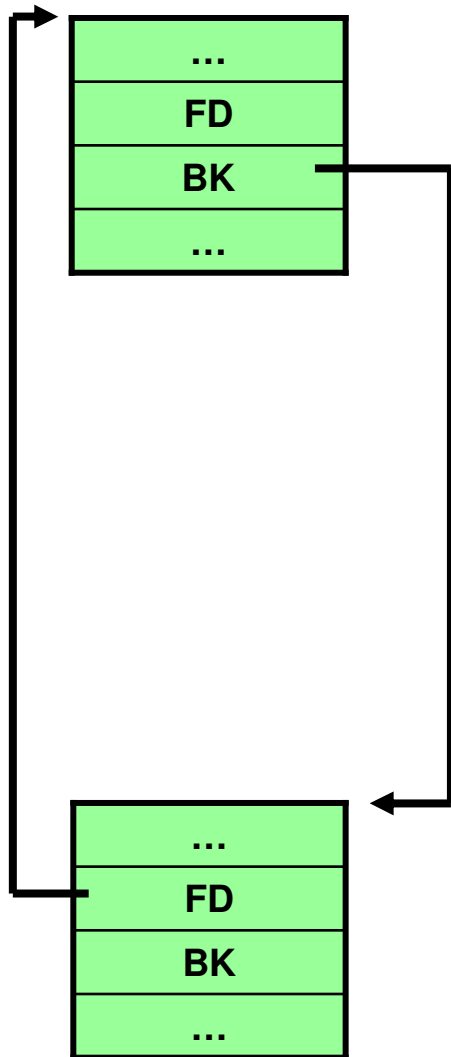
- Pseudocode:

```
this.FD->BK = this.BK
```

```
this.BK->FD = this.FD
```

- **Disclaimer:**

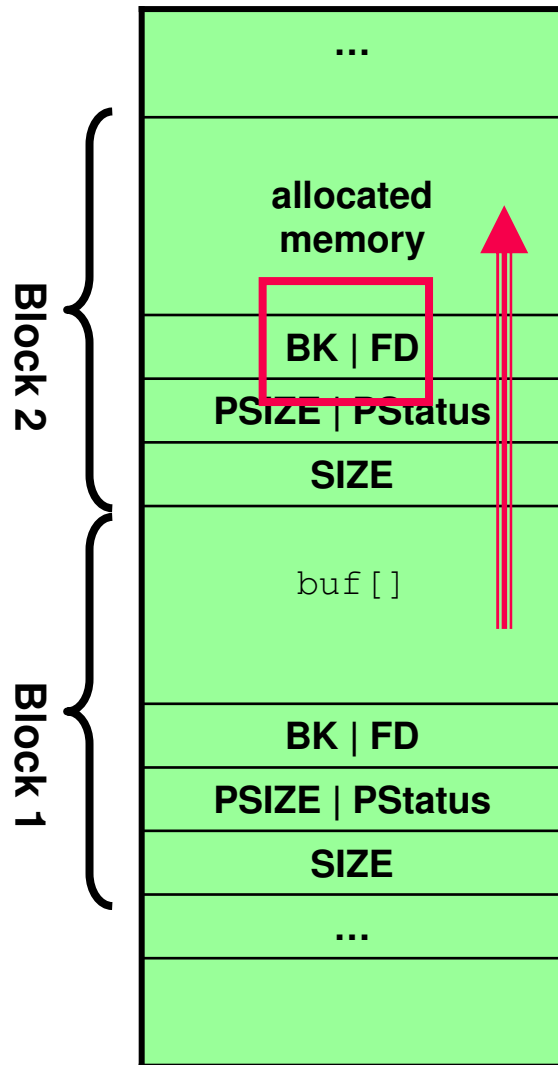
- ◆ This is only a sketch
- ◆ The internals are somewhat more complicated
- ◆ But the concept is valid



- When a block is removed from its bin, the pointers of the list are modified accordingly
- Pseudocode:
 

```

      this.FD->BK = this.BK
      this.BK->FD = this.FD
      
```
- Observation:
  - ◆ `this.FD` represents a pointer to a place in memory that is written to
  - ◆ `this.BK` is the value that is written to this place in memory
- The attack:
  - ◆ If the attacker is able to control FD and BK, he is able to write arbitrary values to arbitrary locations



- When a block is removed from its bin, the pointers of the list are modified accordingly
- Pseudocode:
 

```

      this.FD->BK = this.BK
      this.BK->FD = this.FD
      
```
- Observation:
  - ◆ `this.FD` represents a pointer to a place in memory that is written to
  - ◆ `this.BK` is the value that is written to this place in memory
- The attack:
  - ◆ If the attacker is able to control FD and BK, he is able to write arbitrary values to arbitrary locations



# Conclusion

- **(Almost) every illegal memory access can cause an exploitable vulnerability**
- **Buffer overflows on the stack allow the construction of robust exploit code**
  - ◆ **Exploits can be targeted more loosely**
  - ◆ **Ideal for automatic exploitation via worms**
- **The exploitation of heap overflows and format string vulnerabilities require more detailed knowledge of the situation**
  - ◆ **If this is provided, these vulnerabilities are as serious as stack based buffer overflows**
  - ◆ **Target for tailored attacks**



## Bibliography

- **Michael Howard: “Fix Those Buffer Overruns!”**, 2002,  
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure05202002.asp>
- **Michael Howard: “Reviewing Code for Integer Manipulation Vulnerabilities”**, 2003,  
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp>
- **Michael Howard: “An Overlooked Construct and an Integer Overflow Redux“**, 2003,  
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure09112003.asp>
- **Aleph One: "Smashing The Stack For Fun And Profit"**, Phrack 49, Volume 7,  
<http://www.phrack.org/phrack/49/P49-14>
- **Matt Conover: "w00w00 on Heap Overflows"**,  
<http://www.w00w00.org/files/articles/heaptut.txt>
- **Rix: "Smashing C++ VPTRS"**, Phrack 56, Volume 11,  
<http://www.phrack.org/phrack/56/p56-0x08>
- **J.C. Foster, V. Osipov, N. Bhella, N. Heinen: “Buffer Overflow Attacks”**, Syngress, 2005