Betriebssysteme und Sicherheit – Sicherheit

Buffer Overflows

Software Vulnerabilities

- Implementation error
  - Input validation
- Attacker-supplied input can lead to
  - Corruption
  - Code execution
  - ...
- Even remote exploitation

Famous Buffer Overflow Attacks

- Morris worm (1988): overflow in fingerd
  - 6,000 machines infected (10% of existing Internet)
- CodeRed (2001): overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes** (!!)
- Sasser (2004): overflow in Windows LSASS
  - Around 500,000 machines infected
- Conficker (2008-09): overflow in Windows Server
  - Around 10 million machines infected (estimates vary)

Source: Vitaly Shmatikov

Buffer Overflow

- C / C++ program
  - char buf[16];

- What happens if we execute the following statement?
  - buf[16] = '0';

Dangerous Library Functions

- Null-terminated strings
  - Variable length vs. fixed buffer length
- String Functions
  - strcpy()
  - sprintf()
  - ...

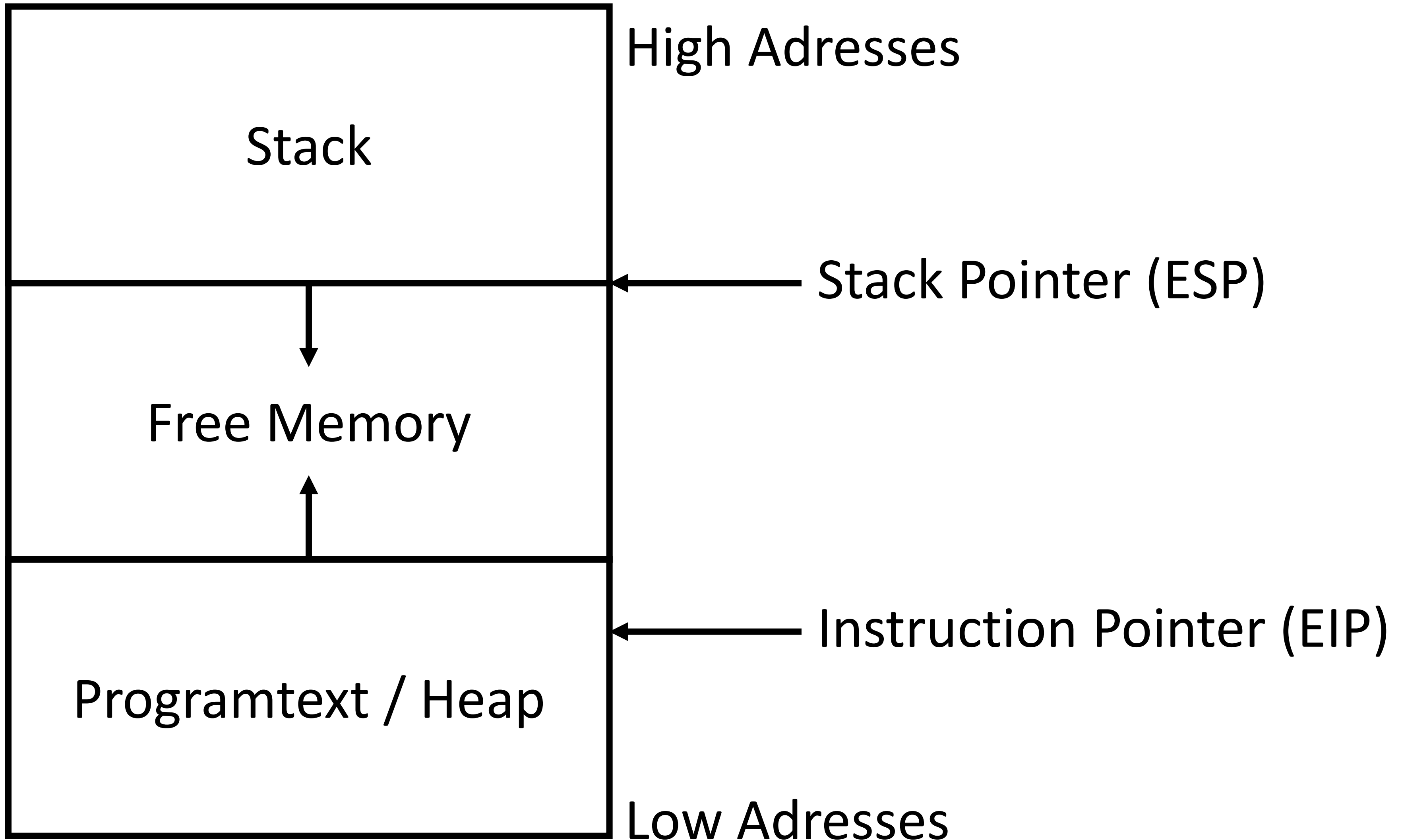
A Primer

- Memory layout
- Intel x86 assembly language
- Compiler artefacts (without optimization)

Stack

- The stack is storage for
  - Local variables
  - Function parameters
  - Return addresses

Process Memory Layout



Register

- Data Register
  - a, b, c, d
  - Divided into 8, 16, 32-Bits
    - 8: al, ah
    - 16: ax
    - 32: eax
- Index Register: esi, edi
- Special Register
  - Instruction Pointer: eip
  - Stack Pointer: esp
  - Frame Pointer: ebp

Memory Access Instructions

- $loc_1 \leftarrow val/loc_2$
  - mov loc_1, val/loc_2
  - lea loc_1, loc_2 (load effective address)
- Values
  - Constants: 4
- Locations
  - Register: eax, ...
  - Memory Address: []
    - Fixed address: 0x...
    - Pointer (register content): eax
    - (Some) pointer arithmetic: [esp + 4]

Stack Access Instruction

- Writing to the stack
  - push val/loc
    - Decreases stack pointer by 4 bytes
    - Moves values/loc to the stack pointer location
- Reading from the stack
  - pop loc
    - Moves value from stack pointer location to loc
    - Increases stack pointer by 4 bytes

Arithmetic Instructions

- $loc_1 \leftarrow loc_1$ operator val/loc_2
  - Addition: add
  - Subtraction: sub
- Sets register flags
  - Zero (equal), negative, overflow, ...
- Comparison
  - cmp loc_1, val/loc_2
  - No storage, just subtraction for register flags

Control Flow Instructions

- Jump: jmp val/loc
  - Absolute (loc) vs. relative (val)
- Conditional jump: je, jne, jl, ...
- Function call: call val/loc
  - Absolute address
  - Pushes instruction pointer (of next instruction) on the stack
- Function return: ret
  - Pops instruction pointer from the stack

C Function Call

C

- f(1, 2);

- int f(int a, int b) {
- char buf[10];

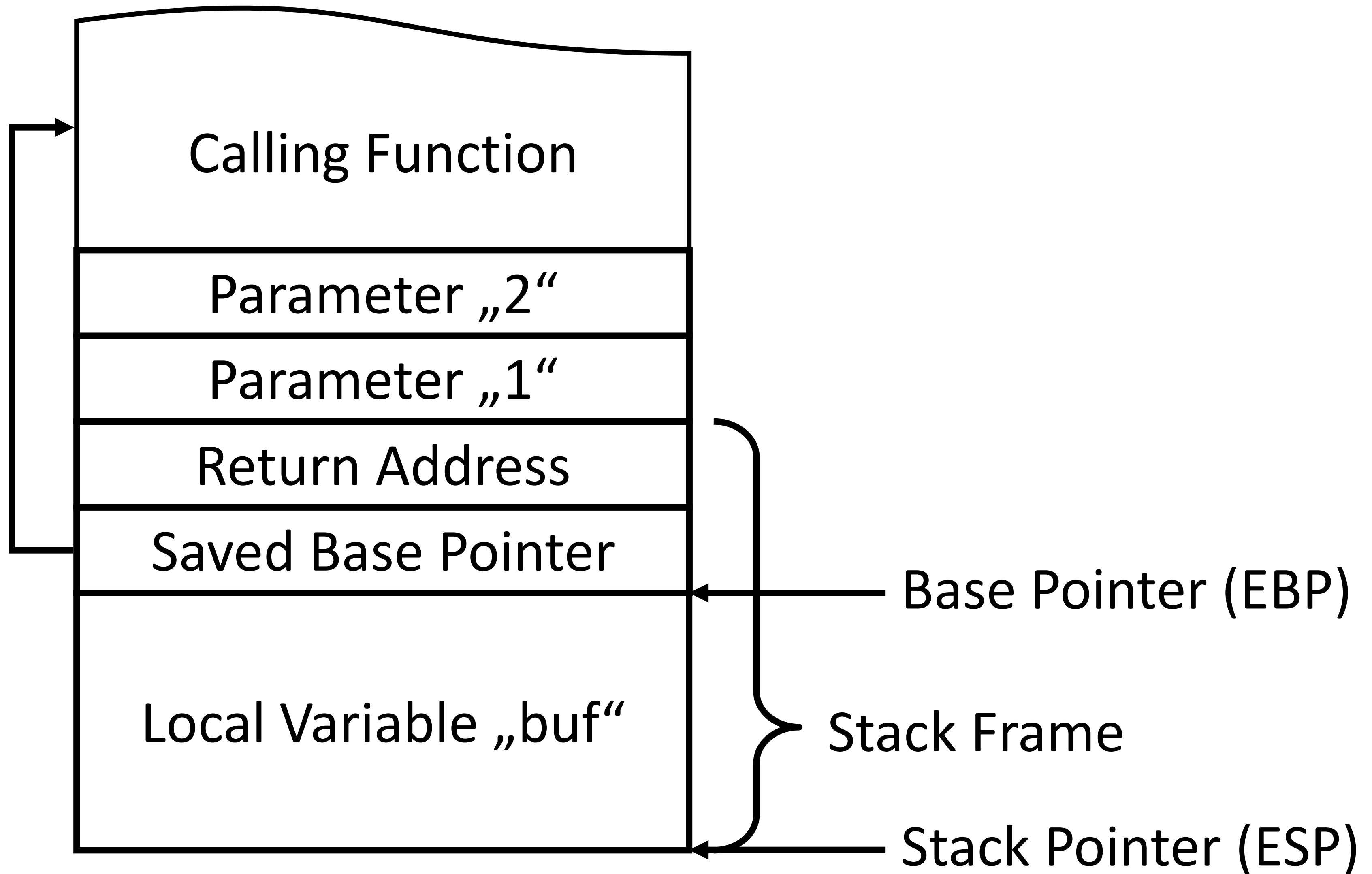
- return 3;

Assembler

- push 2
- push 1
- call 0x... (= &f)
- push ebp
- mov ebp, esp
- sub esp, 12
- mov eax, 3
- mov esp, ebp
- pop ebp
- ret

leave 3

Stack Frame



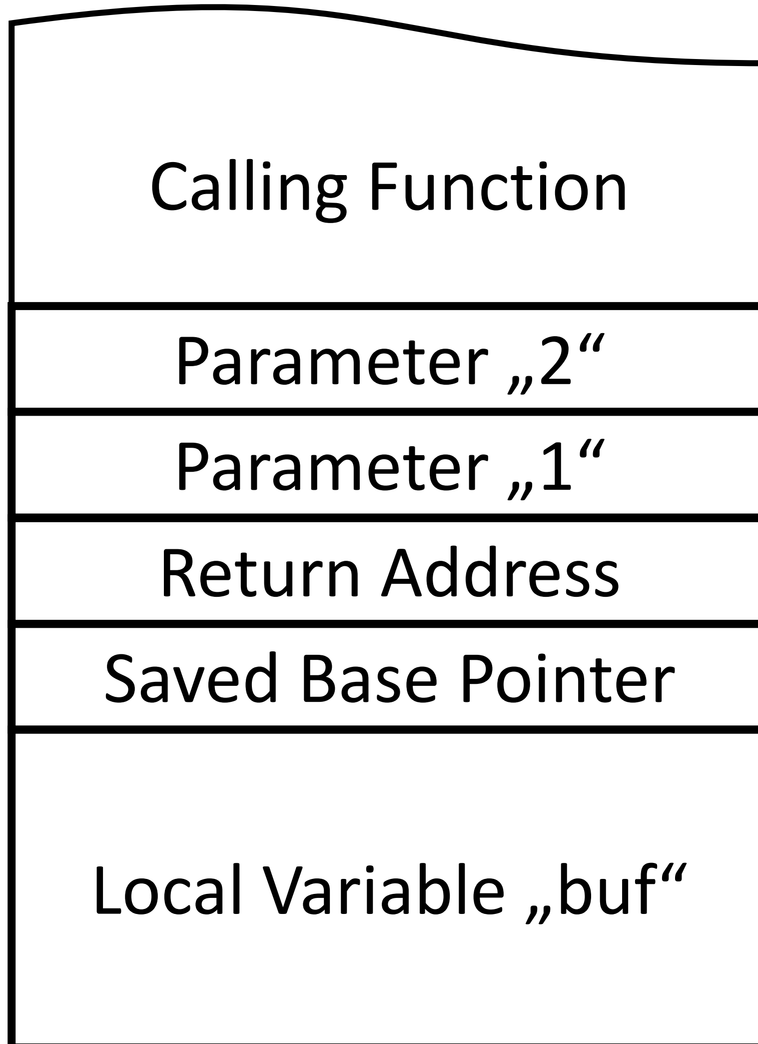
Attack – Vulnerable Code

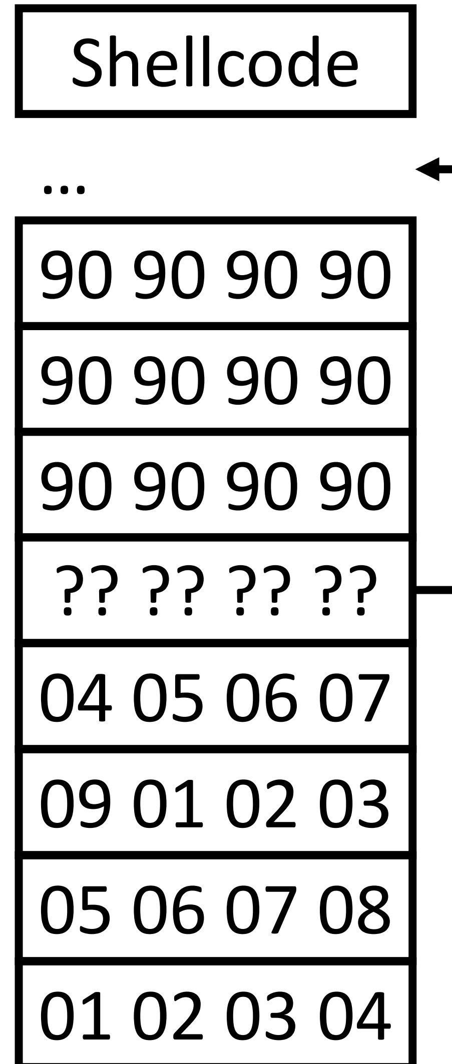
- void f (char* src) {
-     char buf[10];
-
-     strcpy(buf, src);
- }

Attack

- If the parameter *src* contains a string of 20 characters or longer, then the return address is overwritten
- The *ret* instruction may jump **anywhere**
- The attacker may also insert additional code (often called shellcode) in the parameter and write it to the stack. He may then jump **to this code**.

Stack After Attack





Countermeasures I

- Defensive programming
  - Safe library functions
    - strncpy()
    - snprintf()
    - ...
  - Advantage
    - Implementation "only" requires programming discipline
  - Disadvantage
    - By far does not prevent all buffer overflows, e.g. off-by-one errors or self-programmed copy routines.

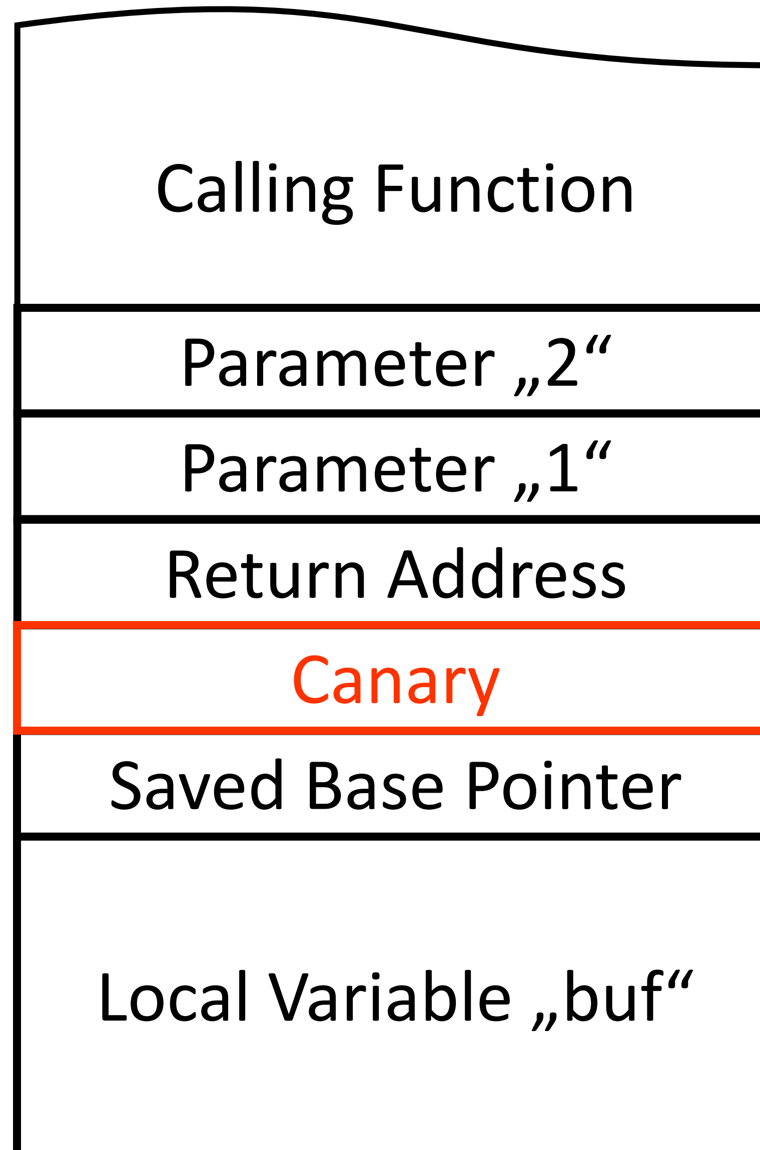
Countermeasures II

- Non-executable stack
  - The memory pages of the stack are marked as non-executable
  - Advantage
    - Shell code placed onto the stack by a buffer overflow can no longer be executed
  - Disadvantage
    - The control flow can still be arbitrarily deflected, e.g. directly to a system call or library (jump to libc).

Countermeasures III

- StackGuard
  - In the function prologue a canary (test value) is placed before the return address. Before returning the integrity of the canary is verified. The instrumentation of the program code is performed during compilation.
  - Two types of canaries
    - Terminators: e.g. 00, EOF, etc.
    - Random value
  - Advantage
    - Stack guard is very efficient
  - Disadvantage
    - Stack guard only protects the stack (there are also heap overflows, etc.)

Stack With StackGuard



Countermeasures IV

- StackShield
  - In the function prologue the return address is copied to the heap. Before returning the address on the stack is compared to the heap. Again, instrumentation is performed during compilation

Countermeasures V

- Array Bound Checks
  - Every index of an access to an array (buffer) is checked against the bounds of the array. This is done automatically the case in type safe languages, e.g. Java.
  - Advantages
    - Prevents buffer overflows for any type of data
  - Disadvantages
    - Every array access has to be check; therefore very slow.
    - Can be difficult to implement for non-type safe languages, e.g. C (e.g. *(buf + 3))

Countermeasures VI

- Address Space Layout Randomization
  - The base addresses of memory segments are chosen randomly, e.g.
    - Stack
    - …
  - Advantage
    - Increases search time for correct parameters
    - Increases the effort for an attacker to perform the same attack on multiple machines

Countermeasures VII

- Instruction Set Randomization
  - The opcodes for instructions are chosen randomly
  - Advantage
    - Every method inserting malicious code is likely to fail.
  - Disadvantages
    - Slow
    - Requires processor/VM support
    - Currently not deployed in practice