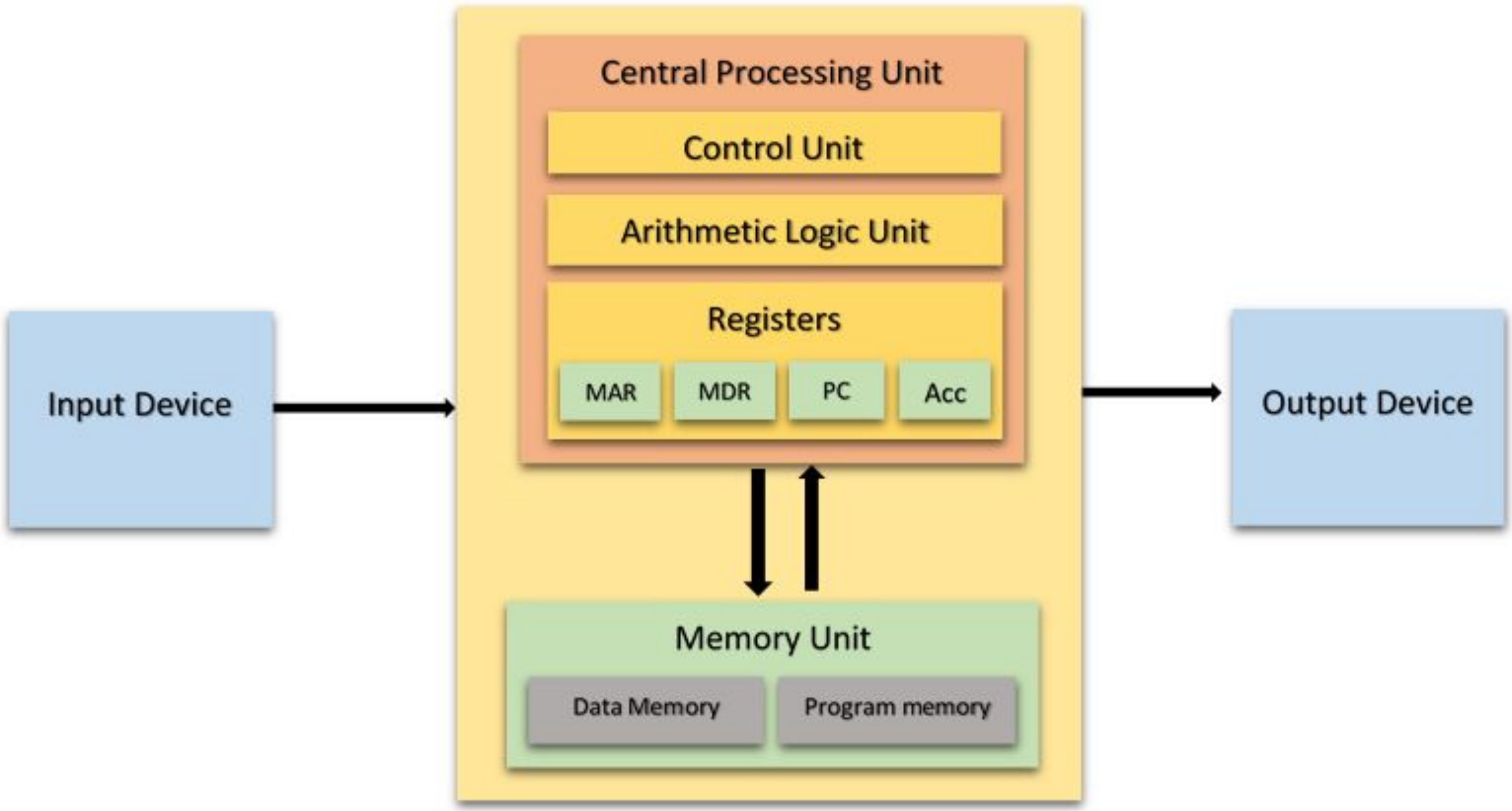
# IERG4130
# Buffer Overflow

By Ke ZHANG
20/9/2022

# Outline

- Background
  - Memory Layout
  - Stack Layout
  - What happens when a function is called
    - Calling Conventions
- Buffer Overflow
  - How to launch
  - How to mitigate
- Example

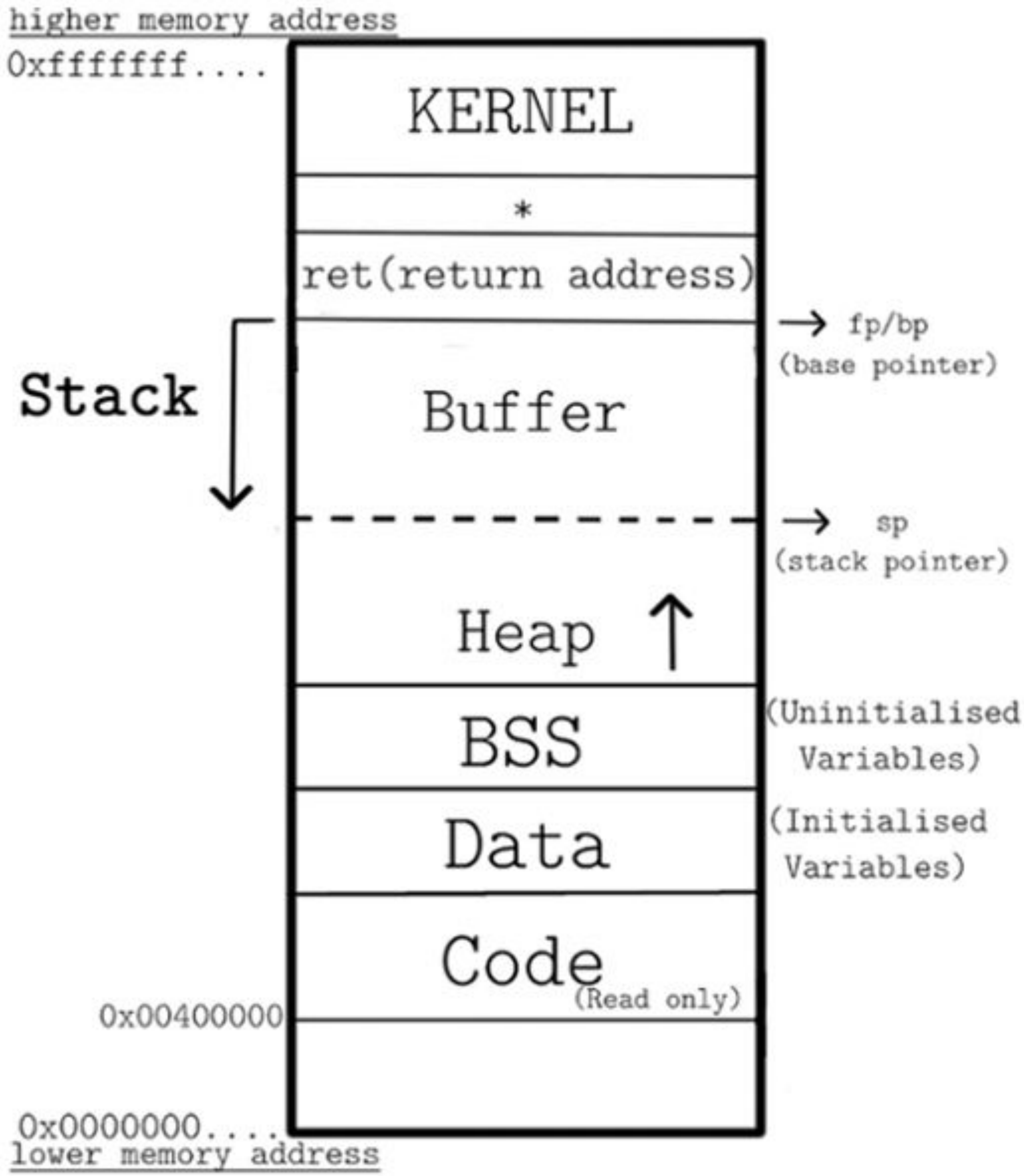
## Assumption: Von-Neumann Architecture



# Memory Layout

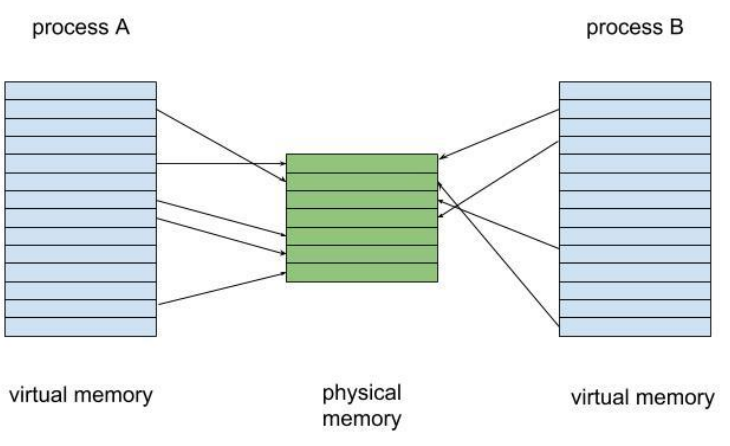
- Program and Process?
    - Program: static; a set of instructions; stored as a file
    - Process: dynamic; when a program runs, it becomes a ~; holds resources
        - A process exists in **memory**
- Memory
    - Memory address space
        - User space vs Kernel space
        - Virtual memory space vs Physical memory space

# Memory Organization



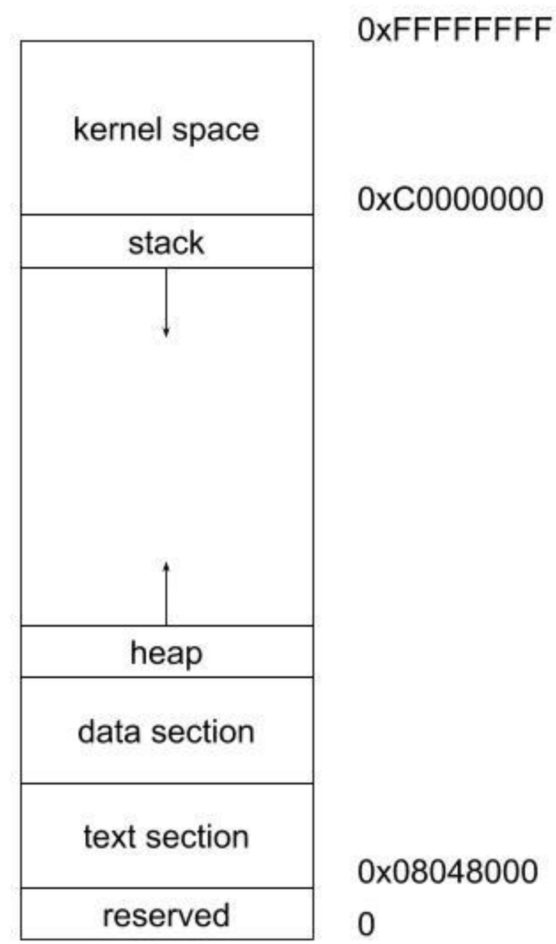
# Virtual Memory

- Each process has its own **virtual** memory
    - virtual memory maps to physical memory
        - size(virtual memory) >> size(physical memory)
    - handle by MMU (Memory Management Unit)
        - Page and Page frame
        - Page table
    - Why we need virtual memory?
        - Abstraction for unified layout
        - Ignore real physical address
        - Isolation
        - Sharing



# Memory Layout

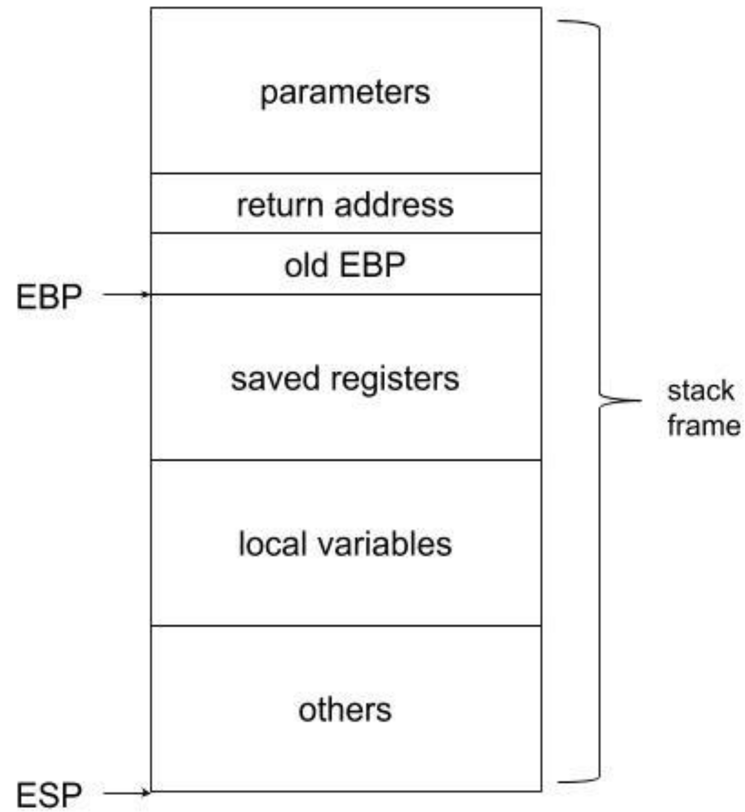
- (Here we only talk about Virtual Memory)
- Stack:
  - 1. data structure: LIFO
  - 2. memory area:
    - grow from high addr to low addr
    - destroy from top to bottom (base)
- For a **function** (procedure)
  - Stack Frame (Activity Record)
  - What does it contain?



# Stack Frame

*what is it all about??*

- (We have come from process-level to function-level)
- You may have a lot of questions:
  - what is EBP and ESP?
  - what does ‘old’ means in ‘old EBP’
  - return address? return where?
  - ……



# Stack Frame

- EBP: Base Pointer, pointing to the stack bottom (high address)
  - fixed
  - EBP + offset: to locate variables
- ESP: Stack Pointer, pointing to the stack top (low address)
  - shifted when POP & PUSH
  - ESP - EBP: the space of a stack
- Note that
  - both EBP and ESP are <u>in CPU</u>.
  - Old EBP and saved registers are values <u>in main memory</u>.

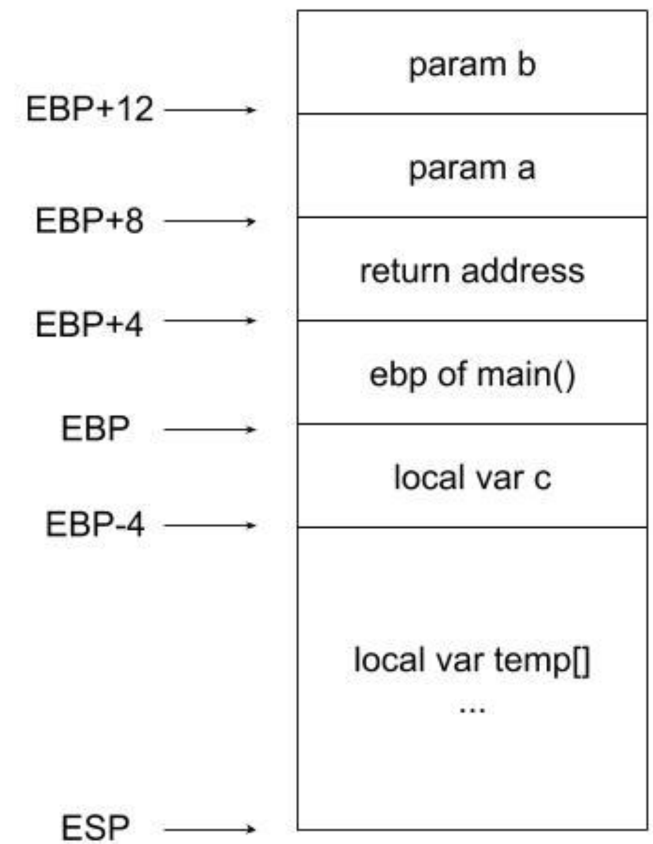
# EBP and ESP

- Use the sample code in lecture notes
-
```c
#include <stdio.h>
int add(int a, int b) {
  int c, temp[512];
  c = a + b;
  return c;      }

int main(int argc, char * argv) {
  int one, two, three;
  one = 1; two = 2;
  three = add(one, two);
  return three;
}
```

# EBP and ESP

- For ***add*** function called by main()
- (assume we do not store registers in stack)
    - for simplicity
    - also not consider stack alignment here
- (assume we push parameters from right to left in source code)
    - another question → why? how?
    - explain in later slides
- [EBP + offset] to locate element inside current frame.

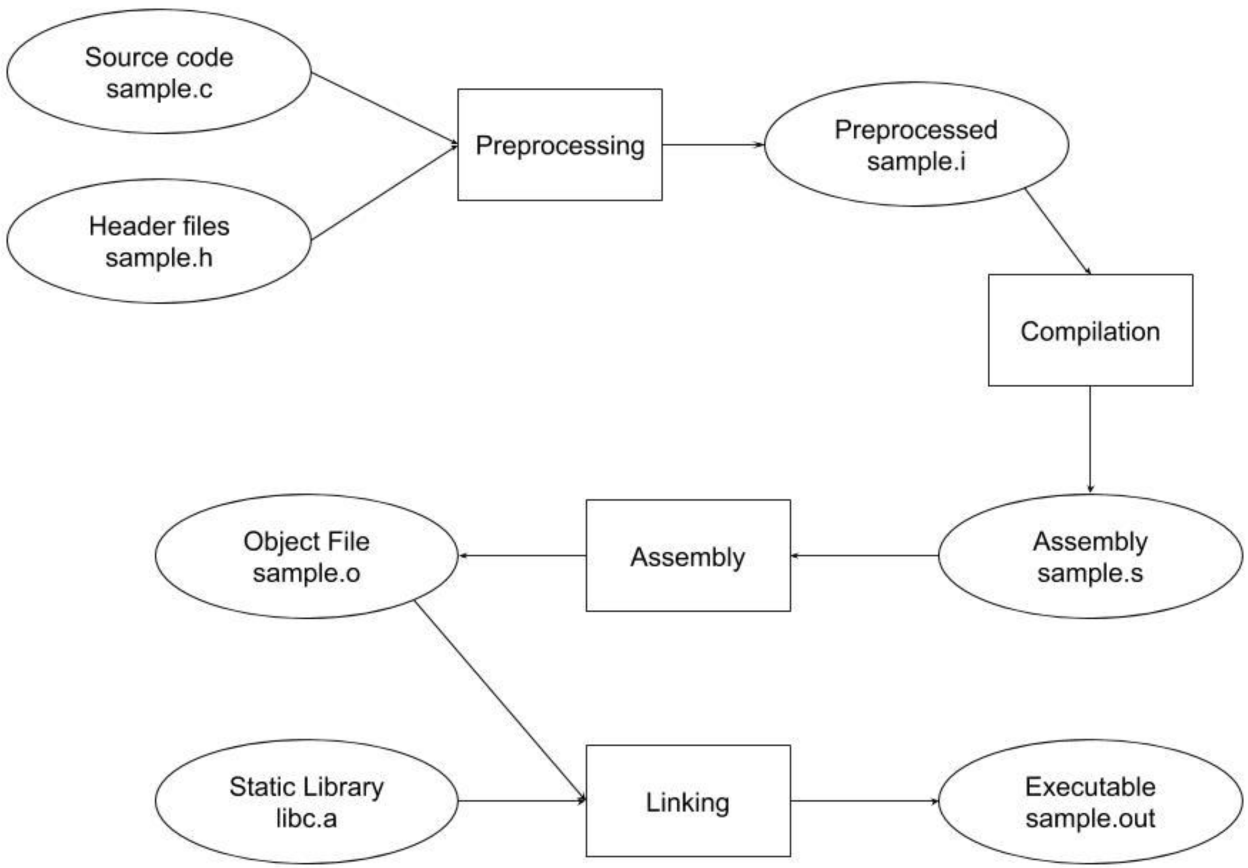


# Stack Frame

- Old EBP & Return Address
  - related to function calls
- Caller & Callee
  - calling function → caller; called function → callee
  - Stack Frame (also known as Activity Record) is function-level
  - Each function has its own stack frame
- When funcA calls funcB, funcB calls funcC …
  - A chain
  - When funcC returns, we need trace back to funcB, and so on
- old EBP: to recover caller’s stack frame
- return address: to continue the exec of caller
  - the next instruction to exec after the callee’s return

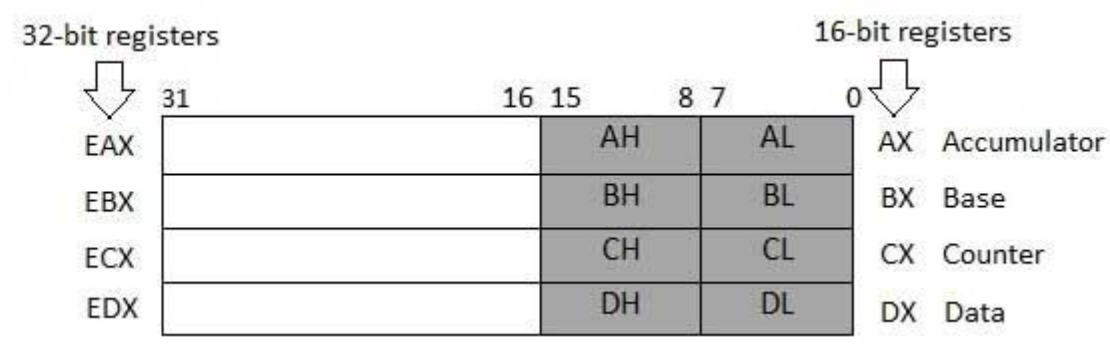
# Assembly basics

- Think about how you get an executable from C source code.



# Assembly basics

- Registers (in CPU)
  - General registers
    - Data registers
    - Pointer registers
    - Index registers
  - Control registers
  - Segment registers




- Data registers
  - EAX, EBX, ECX, EDX
- Pointer registers
  - EIP (Instruction Pointer): stores the offset address of the next instruction to be executed
  - ESP (Stack Pointer) - the top element of stack
  - EBP (Base Pointer) - the stack base

# Assembly basics

Note the following illustrations are simplified to let you familiar with their functions, for a detailed reference please check official manuals.

- Syntax: **Intel** v.s. AT&T
  - MOV EAX, EBX; transfer data in EBX to EAX
  - ADD EAX, EBX; add data in EAX and data in EBX and store the result to EAX
  - PUSH EAX;
    - from register (CPU) to memory
    - equal to ESP=ESP-4; MOV [ESP], EAX
  - POP EAX;
    - from memory to register (CPU).
    - equal to MOV EAX, [ESP]; ESP=ESP+4
  - CALL func;
    - equal to PUSH EIP; JMP func
  - RET ; equal to POP EIP

# Sample

- Use the following sample code
-
```
#include <stdio.h>
int add(int a, int b) {
  int c, temp[512];
  c = a + b;
  return c;       }

int main(int argc, char * argv) {
  int one, two, three;
  one = 1; two = 2;
  three = add(one, two);
  return three;
}
```

add an instruction:
temp[0] = 10;

# What happens in **caller:** main()

```
main:
.LFB1:
        .loc 1 9 0
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 16
        .loc 1 11 0
        mov     DWORD PTR [ebp-4], 1
        mov     DWORD PTR [ebp-8], 2
        .loc 1 12 0
        push    DWORD PTR [ebp-8]
        push    DWORD PTR [ebp-4]
        call    add
        add     esp, 8
        mov     DWORD PTR [ebp-12], eax
        .loc 1 13 0
        mov     eax, DWORD PTR [ebp-12]
        .loc 1 14 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Every function starts with
*push ebp;*
*mov ebp, esp*

allocate space for its stack

for main(), 1 and 2 are **local variables**

push 1 and 2 to the stack
push 1=> ESP=ESP-4; [ESP]=1

remove all pushed parameters, so that we can restore those saved registers

# What happens in **callee:** add()

```
add:
.LFB0:
        .file 1 "sample.c"
        .loc 1 2 0
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 2064
        .loc 1 4 0
        mov     DWORD PTR [ebp-2052], 10
        .loc 1 5 0
        mov     edx, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [ebp+12]
        add     eax, edx
        mov     DWORD PTR [ebp-4], eax
        .loc 1 6 0
        mov     eax, DWORD PTR [ebp-4]
        .loc 1 7 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Every function starts with *push ebp; mov ebp, esp*

I add “temp[0] = 10;” in source code

parameter a

parameter b

local variable c

put return value in eax

restore the old ebp
equals to:
mov esp,ebp
pop ebp

ret equals to
pop eip

# Calling Conventions

- Caller and callee agree on a convention
    - the order of pushing parameters (left to right? right to left?)
    - how to balance the stack
        - callee or caller?
    - name mangling
        - for linking stage
- __cdecl
    - C default calling convention → from right to left; caller to balance
- __stdcall
    - win32 API → from right to left; callee to balance

# cdecl and stdcall

```
_cdecl int MyFunction1(int a, int b)
{
  return a + b;
}
```

and the following function call:

```
x = MyFunction1(2, 3);
```

```
_MyFunction1:
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret
```

and

```
push 3
push 2
call _MyFunction1
add esp, 8
```

```
_stdcall int MyFunction2(int a, int b)
{
  return a + b;
}
```

and the calling instruction:

```
x = MyFunction2(2, 3);
```

```
:_MyFunction2@8
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret 8
```

and

```
push 3
push 2
call _MyFunction2@8
```

# Buffer Overflow

- Why?
  - for continuous memory writing
  - no boundary check for data on stack
- Severe?
  - Overwrite data on stack
  - hijack the control flow → arbitrary (malicious) code execution
- How to launch?
  - Step 1. Find those standard string function calls without a compulsory boundary check
    - A lot of C standard functions
    - strcmp(), strcpy(), ...

# How to launch buffer overflow attack

- step **2**. Write a shellcode
  - Payload
  - Why this name?
- Let’s see an example
  - gcc -fno-stack-protector -ggdb bufferoverflow.c -o bufferoverflow
  - gdb ./bufferoverflow
  - str=’A’*24

```c
#include <string.h>

void foo(char *str){
        char buffer[12];
        strcpy(buffer, str);
}

int main()
{
        char *str = "AAAAAAAAAAAAAAAAAAAAAAAA";
        foo(str);

        return 1;
}
```

# GDB your program

- GNU Project Debugger
- *b 5*
  - add a breakpoint at strcpy() in foo()
- *b 10*
  - add a breakpoint at char *str = “AAAA…”;
- *r*
  - run to the breakpoint 2
- *info frame*

```
gdb-peda$ b 5
Breakpoint 1 at 0x8048411: file bufferoverflow.c, line 5.
gdb-peda$ b 10
Breakpoint 2 at 0x8048437: file bufferoverflow.c, line 10.
```

```
gdb-peda$ info frame
Stack level 0, frame at 0xbfffec90:
 eip = 0x8048437 in main (bufferoverflow.c:10); saved eip = 0xb7d82637
 called by frame at 0xbfffed00
 source language c.
 Arglist at 0xbfffec78, args:
 Locals at 0xbfffec78, Previous frame's sp is 0xbfffec90
 Saved registers:
  ebp at 0xbfffec78, eip at 0xbfffec8c
```

# GDB your program

- *info registers*
    - check register values

```
gdb-peda$ info register
eax            0xb7f1ddbc       0xb7f1ddbc
ecx            0xbfffec90       0xbfffec90
edx            0xbfffecb4       0xbfffecb4
ebx            0x0      0x0
esp            0xbfffec60       0xbfffec60
ebp            0xbfffec78       0xbfffec78
esi            0xb7f1c000       0xb7f1c000
edi            0xb7f1c000       0xb7f1c000
eip            0x8048437        0x8048437 <main+17>
```

- *x/16x 0xbfffece8*
    - See stack layout

previous eip

```
gdb-peda$ x/16x 0xbfffec78
0xbfffec78:     0x00000000      0xb7d82637      0xb7f1c000      0xb7f1c000
0xbfffec88:     0x00000000      0xb7d82637      0x00000001      0xbfffed24
0xbfffec98:     0xbfffed2c      0x00000000      0x00000000      0x00000000
0xbfffeca8:     0xb7f1c000      0xb7fffc04      0xb7fff000      0x00000000
```

- *c*
    - continues to run to next breakpoint (breakpoint 1)
- *info frame*

```
gdb-peda$ info frame
Stack level 0, frame at 0xbfffec50:
 eip = 0x8048411 in foo (bufferoverflow.c:5); saved eip = 0x8048449
 called by frame at 0xbfffec90
 source language c.
 Arglist at 0xbfffec48, args: str=0x80484e0 'A' <repeats 24 times>
 Locals at 0xbfffec48, Previous frame's sp is 0xbfffec50
 Saved registers:
  ebp at 0xbfffec48, eip at 0xbfffec4c
```

# GDB your program

- *x/16x 0xbfffec48*

previous eip

```
gdb-peda$ x/16x 0xbfffec48
0xbfffec48:     0xbfffec78      0x08048449      0x080484e0      0x00000000
0xbfffec58:     0xb7d98a50      0x080484ab      0x00000001      0xbfffed24
0xbfffec68:     0xbfffed2c      0x080484e0      0xb7f1c3dc      0xbfffec90
0xbfffec78:     0x00000000      0xb7d82637      0xb7f1c000      0xb7f1c000
```

- *b 6*
- *c*
- *x/16x 0xbfffec48*
-

```
[-----------------------------------code-----------
   0x804841a <foo+15>:  push   eax
   0x804841b <foo+16>:  call   0x80482e0 <strcpy@plt>
   0x8048420 <foo+21>:  add    esp,0x10
=> 0x8048423 <foo+24>:  nop
   0x8048424 <foo+25>:  leave
   0x8048425 <foo+26>:  ret
   0x8048426 <main>:    lea    ecx,[esp+0x4]
   0x804842a <main+4>:  and    esp,0xfffffff0
```

we have executed strcpy()

```
gdb-peda$ x/16x 0xbfffec48
0xbfffec48:     0x41414141      0x08048400      0x080484e0      0x00000000
0xbfffec58:     0xb7d98a50      0x080484ab      0x00000001      0xbfffed24
0xbfffec68:     0xbfffed2c      0x080484e0      0xb7f1c3dc      0xbfffec90
0xbfffec78:     0x00000000      0xb7d82637      0xb7f1c000      0xb7f1c000
```

previous EBP has been mangled!

# GDB your program

- We have successfully mangled saved old EBP
- How to mangle return address?
  - return to the function you want to execute
  - (by carefully calculation and overflow)
- Solve it in our lab 1. (will release soon)

# How to defend buffer overflow

- Lack checking? Do checking!
  - strcpy → strcpy_s in MS
  - *https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/strcpy-s-wcscpy-s-mbscpy-s?view=vs-2017*
- Address Randomization - ASLR
  - Attacker needs know where the stack is.
  - Make “guess” much harder
- StackGuard
  - Canary

# Appendix

- Stack Guard Protection Scheme
  - Turned on by default
  - To turn it off: *gcc -fno-stack-protector example.c*
- Stack Overflow

| | | |
|---|---|---|
| … local variable ... | old ebp | return addr |
| ...AAAAAA... | AAAA | AAAA |

- Use “Canaries” to detect
  - A canary is a value inserted between *local variable* and *return address*
  - If you want to overwrite *return address*, the canary will be first overwritten
  - Detected !

# -fno-stack-protector

- *gcc -fno-stack-protector -S -masm=intel -m32 -ggdb sample.c -o sample2.S*

```
add:
.LFB0:
        .file 1 "sample.c"
        .loc 1 2 0
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 2064
        .loc 1 4 0
        mov     edx, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [ebp+12]
        add     eax, edx
        mov     DWORD PTR [ebp-4], eax
        .loc 1 5 0
        mov     eax, DWORD PTR [ebp-4]
        .loc 1 6 0
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

16B+512*4 = 2064B, not know why 16B?
→ Stack Alignment (default 16B)

Parameter a

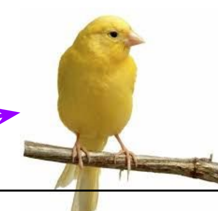
Parameter b

# -fno-stack-protector

- *gcc -S -masm=intel -m32 -ggdb sample.c -o sample2.S*

```
add:
.LFB0:
        .file 1 "sample.c"
        .loc 1 2 0
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 2072
        .loc 1 2 0
        mov     eax, DWORD PTR gs:20
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        .loc 1 4 0
        mov     edx, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [ebp+12]
        add     eax, edx
        mov     DWORD PTR [ebp-2064], eax
        .loc 1 5 0
        mov     eax, DWORD PTR [ebp-2064]
        .loc 1 6 0
        mov     ecx, DWORD PTR [ebp-12]
        xor     ecx, DWORD PTR gs:20
        je      .L3
        call    __stack_chk_fail
```

16B+512*4 = 2064B, why 2072B?!
8B for Canary !



canary is from ebp-12 to ebp-4
DWORD = 8bytes

local variable c comes here
from ebp-2064 to ebp-2060 (4B)

buffer temp[512] is from
ebp-2060 to ebp-12 (2048B)

XOR to check canary