Software Security 1: Buffer Overflow

Kehuan Zhang

IERG4130 2022

Stack Memory and Function Calls

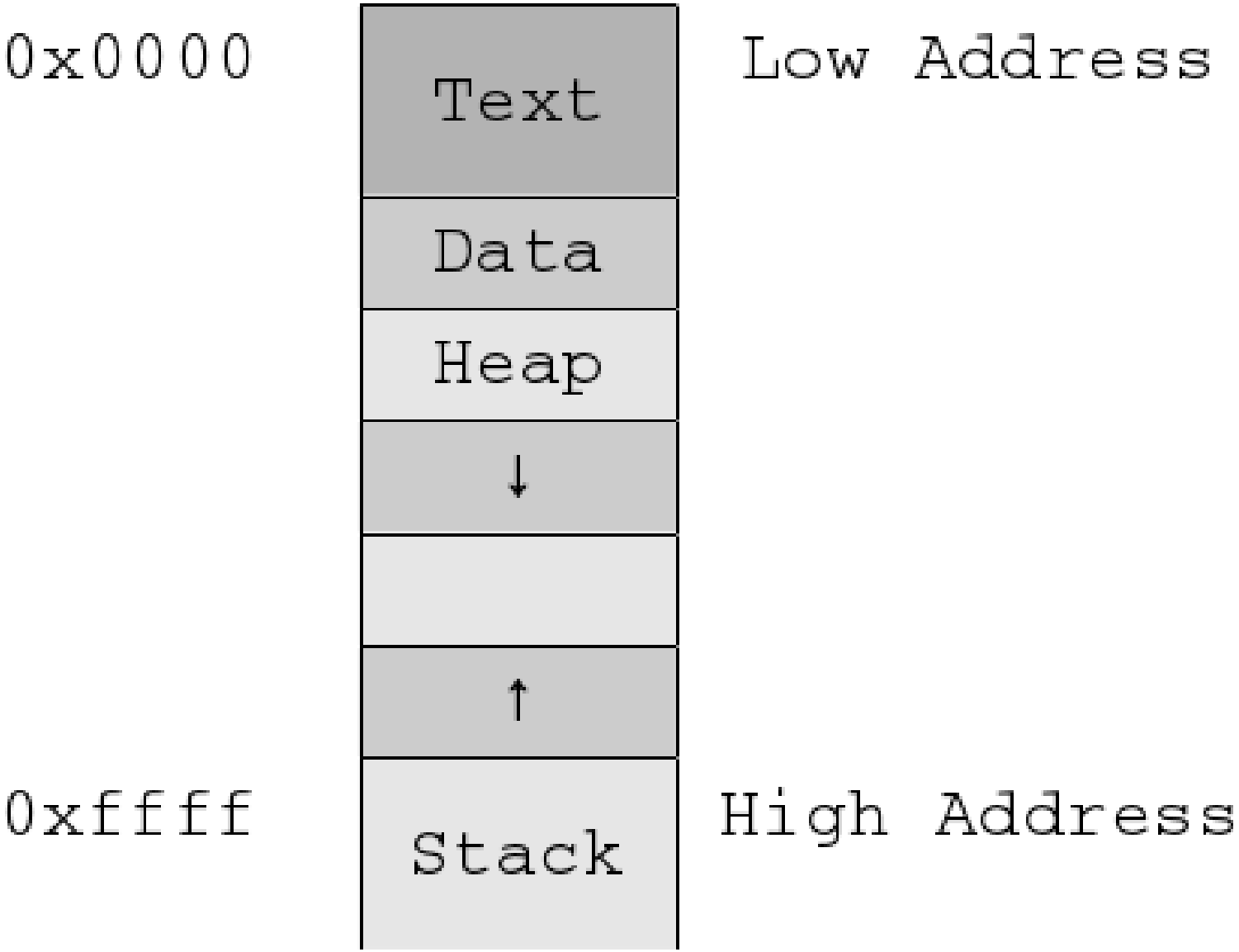
What is Buffer Overflow

- To understand the problem, we need to understand:
  - What is Buffer?
  - Why can Buffer be overflowed?
  - What could happen if a buffer was overflowed?

Memory Structure of a Process

- Text Segment: Program code
  - Begin at lower end address
  - Normally read-only (except self-modifying code)
- Data Segment: Global and static variables, constants
- Heap
  - Dynamic storage spaces allocated via malloc(), new()
- Stack: also for dynamic variables
  - e.g., local variables, passing arguments, etc.
  - Begin at higher-end address and going down-wards
  - Dynamically share memory spaces with Heap
- Heap vs. Stack
  - Heap and stack share the same space
  - They are growing towards to each other
    - ⋆ Why is it designed in this way?
  - Stack Point (stack top) grow to LOWER address

A visualized view of process memory



Stack Basics

- Two basic operations: push and pop, both on the stack top
- One important use of stack is to support function calls
  - function local variables
  - function arguments
- Besides, stack itself can be used as a data source for instructions
  - Stack-oriented instructions
    - ★ Which will assume and take operands from stack top
  - E.g., ADD
    - ★ No operand was given after the 'ADD' opcode
    - ★ Actual operation: pop the top two numbers in the stack, add them, and push the sum back to stack top

Simple Facts about Function Call

- Function call is everywhere
  - Why do we need function call? (code reuse, modular, etc.)
- What do we need for calling a function?
  - Arguments (to be passed from caller to callee function)
  - Return address (address of next Instruction after returning)
  - Local variables (need for callee's internal calculation)
  - How about *Return Value*? (it's optional)
    - ⋆ Sometimes function without returning value is called *procedure*
- On x86 architecture, all needed values will go through Stack
  - Stacks are organized into **frame**s
  - Each function has its own frame
  - Support recursive function calls (but sometimes is problematic)
- Each stack frame has similar structure and elements
  - Arguments, local variables, returning address
  - As well as stack (top) pointer (ESP, Extended Stack Pointer)
  - And Base frame pointer (EBP, or Frame Pointer, Extended Base Pointer)
  - EBP points to previous frame (caller's stack frame)

What Happens during a Function Call?

- By caller function
  - Push arguments (follow certain orders, e.g., right to left for C)
  - Push return address (done by CALL instr automatically)
  - Load the addr of the 1st instruction of callee function to EIP register (also by CALL instr)
- By callee function
  - PUSH EBP (to save caller function's stack Base Pointer)
  - Setup callee function's new stack base pointer (what is it?)
  - Save context (i.e., registers to be used in callee function)
  - Adjust ESP (to allocate space for local variables)
  - Execute desired instructions in this callee function
  - Restore previous stack frame (depends on call convention)
  - Finally, RET instruction will load return address from stack to PC
- Two conventions to adjust stack pointer (SP) – balancing the stack
  - Why do this problem exist?
  - __stdcall: callee function will do stack balancing
  - __cdecl: C default calling convention -> caller to adjust

A Demo on Simple C code

- What will the assembly code be looked like?

```c
#include <stdio.h>
int add(int a, int b) {
  int temp[512];
  return a + b;
}

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

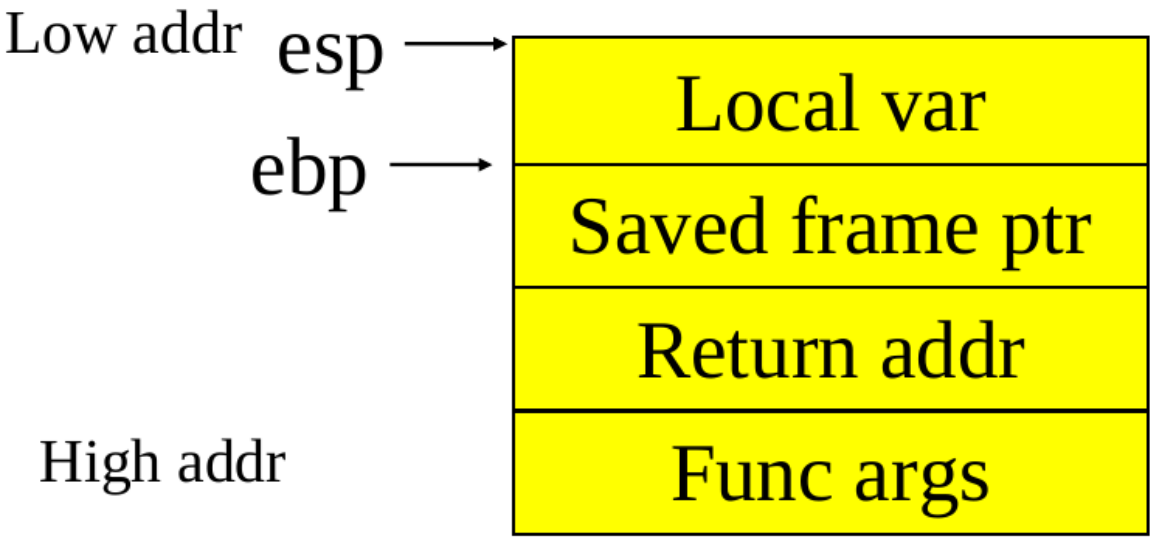
The Assembly Code for Demo Program

- Full code can be generated with command (or Download here):
  - compile command: `gcc -S -masm=intel -m32 -ggdb a.c -o a.S`
    - ★ -S: assembly only, -masm: intel syntax, -m32: 32-bit CPU, -ggdb: with debug info

```
add: ; Note: dest register will go first, then src reg
    push ebp        ; save the frame pointer of main (caller)
    mov ebp, esp    ; set up new frame (callee) on stack top
    sub esp, 2048 ; space for local variables. Why 2048?
    mov edx, DWORD PTR 8[ebp]    ; load 'a'
    mov eax, DWORD PTR 12[ebp]   ; 'b': why 'a' is shallower?
    add eax, edx                 ; pass return value through r
    leave ; equiv. to "mov esp, ebp  &  pop ebp": faster
    ret   ; load returning address from stack to EIP/PC reg
```

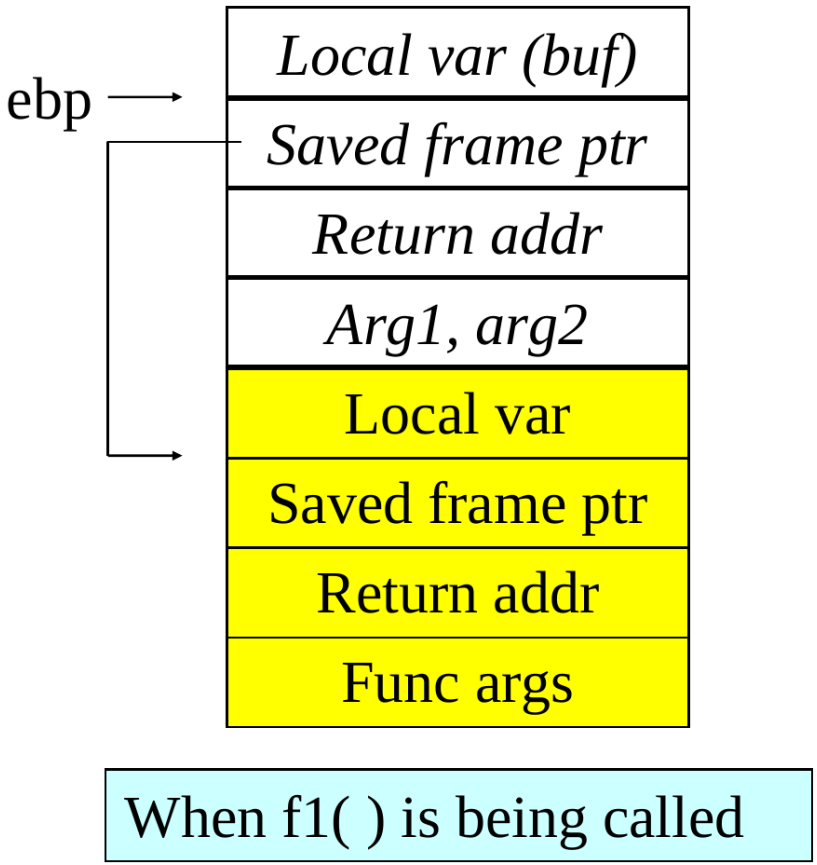
A Graphic View of Stack – Before Function Call

- NOTE
  1. Pay attention to the position of lower and higer address
     - ★ Push a value will make Stack Pointer move toward lower addr
  2. For a given piece of code, you should be able to draw its stack



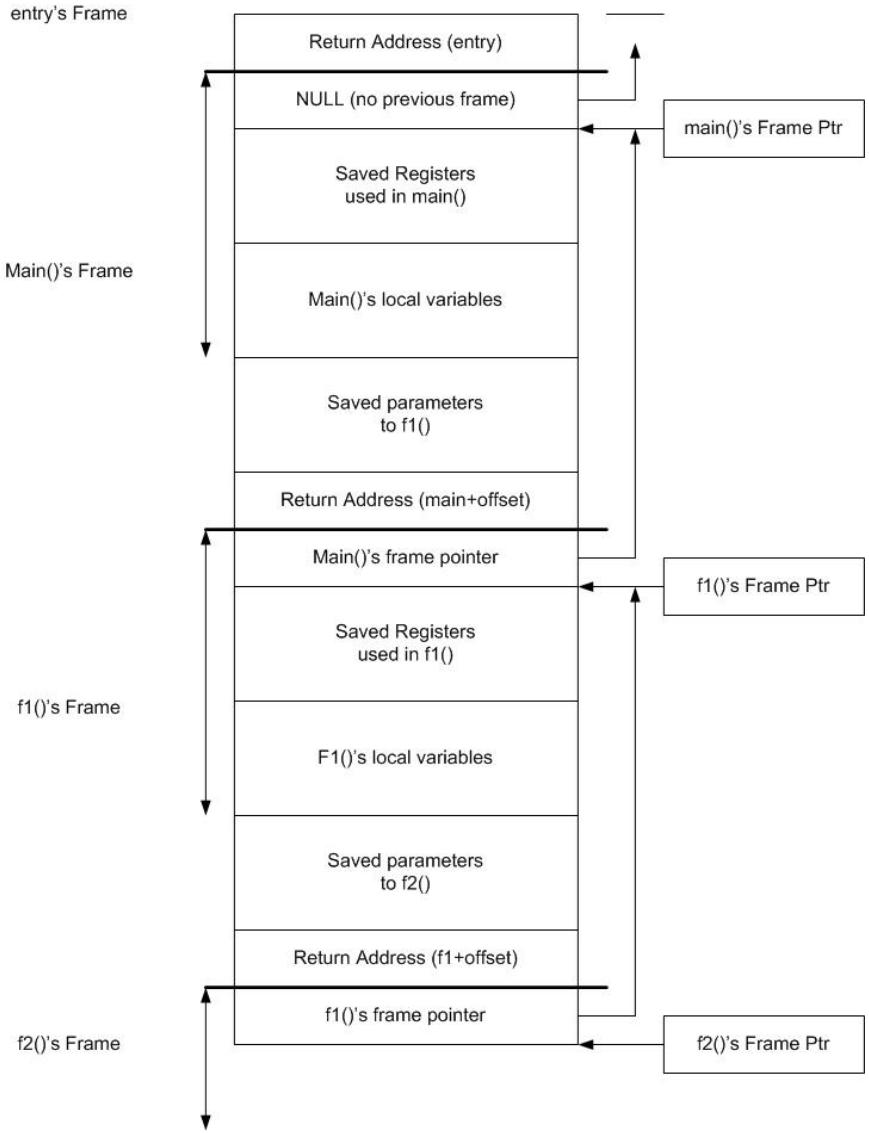
Stack layout before f1( ) is called

A Graphic View of Stack – After Function Call



Stack Structure for Nested Function Calls

- NOTE: Higher address is on the top in this figure!!



Launch of Buffer Overflow Attack

Why Buffer Can Be Overflowed?

- No boundary check for writing data on stack
  - E.g., array in C/C++
  - So it is possible to write beyond the data structure boundary
  - Some languages do have the feature of bounds checking
    - E.g., Python, Java
    - Which in general is safer
    - *Rust* is a new programming language, aming to achieve memory safety (Lastest
    - Reading material: Implications of Rewriting a Browser Component in Rust `https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/`

What Will Happen if Buffer is Overflowed?

- Overwrite data on stack $\rightarrow$ stack structure destroyed
- Consequences
  - Minor one: ruin the data $\rightarrow$ data make nosense
  - Severe one: change critical variable and program behaviour
    - ★ Boolean flags from False to True
  - Most dangerous one: code injection and execution
    - ★ E.g., overwrite the returning address
    - ★ Thus could hijack the program control flow (i.e., compromise Control Flow Integrity CFI)
  - What could attack do with buffer overflow attack?
    - ★ Arbitrary code execution (code injection attack)
    - ★ Denial of Service (e.g., corrupt the stack structure)

Two Conditions for Successful Attacks

1. Inject attack code into buffer of vulnerable process
   - The attack code normally called "shell code", because the injected code will often open a remote shell controlled by attackers
2. Change the execution path of the vulnerable process to run the injected code
   - Normaly overwirte the returning address on stack to replace it with the entry address of injected code

Sample Vulnerable Code 1

```
#include <stdio.h>

void main(int argc, char *argv[]) {
    char buffer[512];
    if (argc > 1)
        strcpy (buffer, argv[1]);
}
```

- Where is the vulnerable point? Why?
- What is the possible consequence?
- Can you draw and analyze the stack structure to confirm your answers?

Sample Vulnerable Code 2

```
int main(int argc, char *argv[]) {
    char passwd_ok = 0;
    char passwd[8];
    strcpy(passwd, argv[1]);
    if (strcmp(passwd, "niklas")==0)
        passwd_ok = 1;
    if (passwd_ok) { ... }
}
```

- Where is the vulnerable point?
- What is the possible consequence?
- Please draw and analyze the stack structure by yourselves

Sample Shell Code

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

- But real world shell code is more complex
  - Need careful calculation on the layout of stack data
  - And other auxiliary code before the shell code
  - C source code -> compile into binary code -> hex string
  - Lab 1
- Reading Material:
  - Shellcode Walkthrough: `http://security.cs.pub.ro/hexcellents/wiki/kb/exploiting/shellcode-walkthrough`

Assembly code of compiled sample shell code

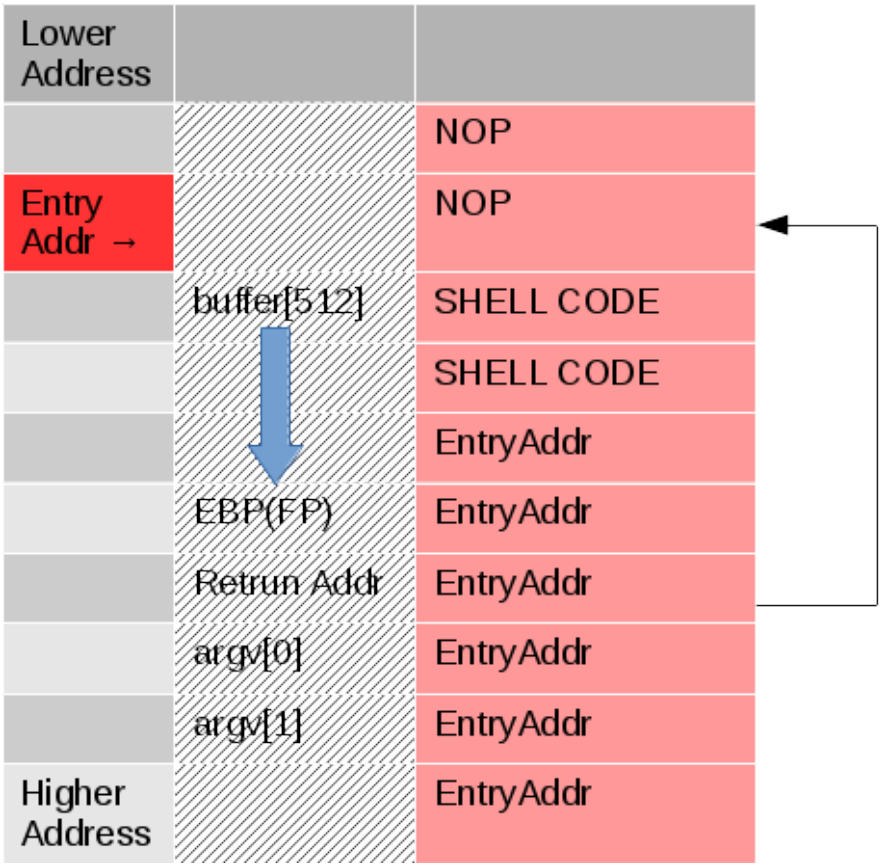
```
      nop
      nop                   //end of nop sled
      jmp find              //jump to end of code
cont: pop %esi              //pop address of sh off stack into %esi
      xor %eax, %eax        //zero contents of EAX
      mov %al, 0x7(%esi)    //copy zero byte to end of string sh (%esi)
      lea (%esi), %ebx      //load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi)    //save address of sh in args [0] (%esi+8)
      mov %eax,0xc(%esi)    //copy zero to args[1] (%esi+c)
      mov $0xb,%al          //copy execve syscall number (11) to AL
      mov %esi,%ebx         //copy address of sh (%esi) into %ebx
      lea 0x8(%esi),%ecx    //copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx    //copy address of args[1] (%esi+c) to %edx
      int $0x80             //software interrupt to execute syscall
find: call cont             //call cont which saves next address on stack
sh:   .string "/bin/sh "    //string constant
args: .long 0               //space used for args array
      .long 0               //args[1] and also NULL for env array
```

```
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
```

Stacks Before the Attack (Vulnerable Code 1)

Lower Address	
	buffer[512]
	EBP(FP)
	Retrun Addr
	argv[0]
	argv[1]
Higher Address	

Stacks After the Attack (Vulnerable Code 1)



Other memory object can also be overflowed

- For example,
  - Heap
  - Integer
  - Format String (e.g., those used in printf())
- Any criteria to decide whether something can be overflowed?
  - Can affect control flow? Boundary protection?
  - Your thoughts?

Heap Overflow

- Heap contains data object created with malloc() or new()
- It has the same property as buffer objects on stack
- Although it may not be able to overwrite Return addr directly
  - It can still take over the control flow
  - E.g., function pointers
  - Question: how can we exploit the feature of function pointers to launch heap overflow attack?

Integer Overflow

- Every object has limited range depends on lower hardware
- For example, on 32-bit CPU:
  - int $\rightarrow$ signed 32-bit by default, $-2^{31} \sim 2^{31}$-1
  - unsigned short $\rightarrow$ 16-bit by default, $0 \sim 2^{16}$-1
  - short $\rightarrow$ signed 16-bit, $-2^{15} \sim 2^{15} - 1$
- What is the value of a after executing following code?

```
unsigned short a = 65535; // which is 2^16-1, 0xFFFF
a = a+1; // 0x0000, together with overflow-flag?

short b = 32767; // which is 2^15-1 0x7FFF
b = b+1; // 0x8000 --> what value is it?
```

An Example of Integer Overflow Attack

- Where is the vulnerability? Why is it vulnerable? How can we fix it?

```
void func(int a, int v) {
    int buf[4];
    // init(buf);
    buf[a] = v;
}
```

```
func:
    push    ebp
    mov ebp, esp
    sub esp, 16  ; allocate space for "int buf[4]"
    mov eax, DWORD PTR [ebp+8] ; get index a and save to eax
    mov edx, DWORD PTR [ebp+12] ; get v and save it to edx
    mov DWORD PTR [ebp-16+eax*4], edx ; save v to buff[a]
    nop  ; in above instr, ebp-16 is starting addr of buff
    leave
    ret
```

Format String Overflow

```
int func(char *user) {
    fprintf(stdout, user);
}
```

- Where is dangerous?
  - "char *user" is used as a format string
- What if user = "%s%s%s%s%s%s%s" ??
  - It may either crash and cause "Segmentation Fault"
  - Or leakout content accidently and cause privacy problems
- Correct Form

```
int func(char *user) {
    fprintf(stdout, "%s", user);
}
```

- See `http://julianor.tripod.com/bc/formatstring-1.2.pdf` for more details

Defense Against Buffer Overflow

Buffer Overflow Attack is Very Popular

- Buffer Overflow has a long history
  - Can be traced back to 1988, the first Internet worm by Morris
- Many known defense technologies are available
- But it is still very popular and a major problem. Why?
  - Legacy and widely deployed code that is hard to upgrade
  - Inexperienced and careless developers
  - Lack of code checking step in development flow
  - New types of attacks, e.g., Return-orienged-programming (ROP)

Methods to Defense Agaisnt Buffer Overflow Attack

- Fundamentally, we need to perform bounds checking
  - But the question is who, when and how.
- Who will do the checking?
  - Human beings: generally it means "code review"
  - Software: e.g., compiler, integrity verfication code, etc.
  - Hardware: e.g., CPU features
- When to perform the check?
  - Static: at compile time or a separate step before code running
  - Dynamic: at runtime
- How to perform the check?
  - Safer version of functions: e.g., strcpy() vs. strncpy()
  - By hardware: no execution bit/flag or shadow stack
  - By software: stack integrity checking (canary)

Strategies And Stages

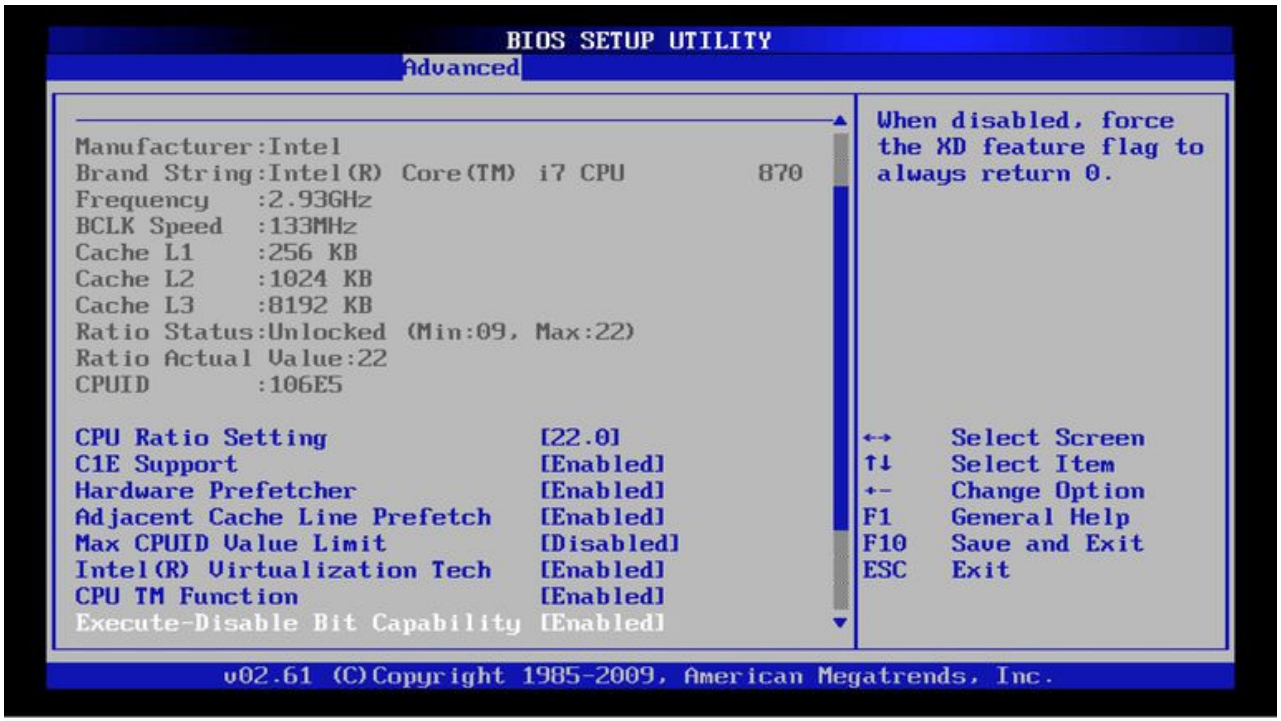
- Strategies
  - Detect and elliminate such vulnerabilities (ideally)
  - Prevent code injection
  - Detect code injection
  - Prevent code execution
- Stages
  - At source code level
  - At code linking and loading time
  - At code execution time
- Try to analyze those defending methods by yourselves

Using Hardware Features from CPU

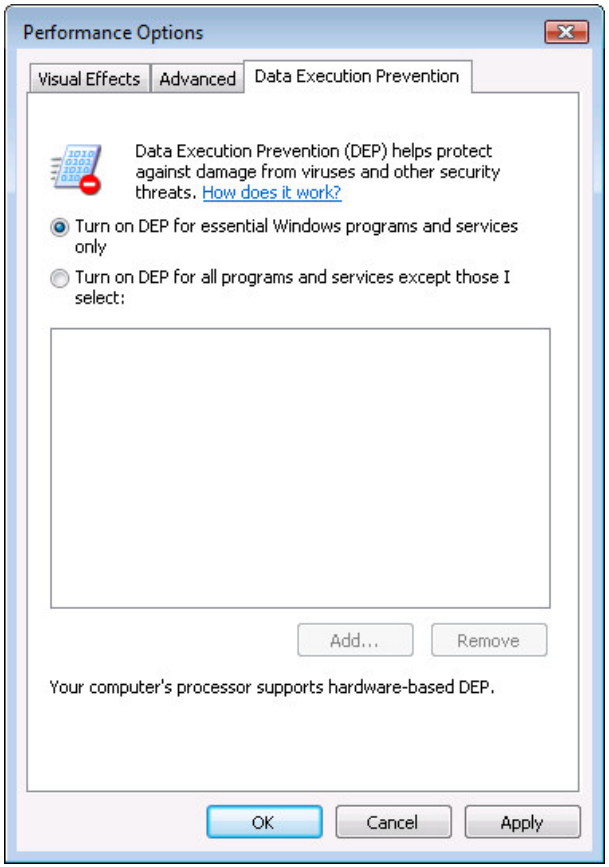
- Who: hardware, When: dynamic
- How?
  - No Execution Bit for Stack Memory (or others when needed)
  - Shadow Stack
  - Also require support from Operating Systems

No Execution Bit

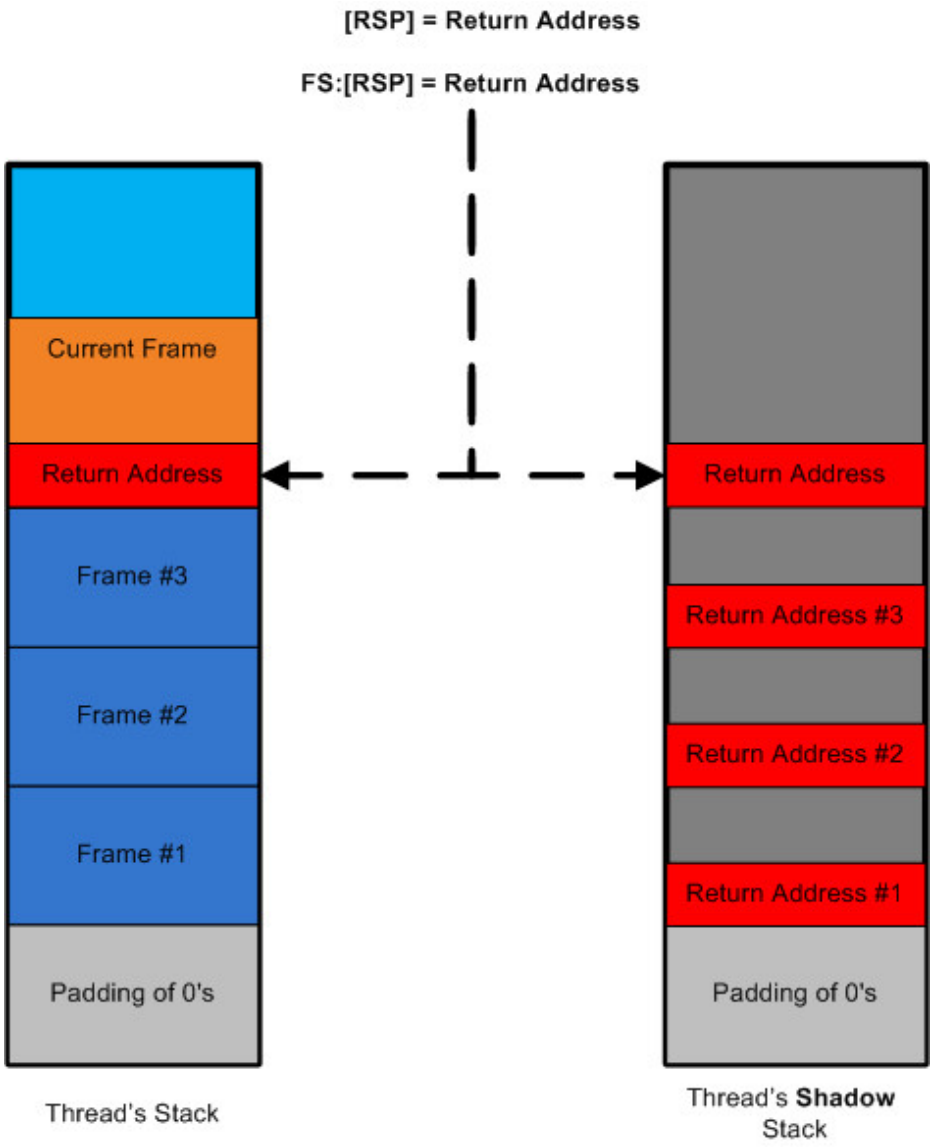
- Most OS will a default rule of W^X for memory protection
  - which means: if write then no execution
  - Turn on NX feature of CPU in BIOS (may already be on by default)
  - Also need support from OS (may already on by default)



No Execute Bit in OS (Demo: memory maps on Linux)



Shadow Stack



Static Defenses

- For developers: educating them to remember to perform bounds check
  - Use safe versions of common string functions
  - strcpy() vs. strncpy(), sprintf() vs. snprintf(), strncat, etc.
  - Code Auditing: manual inspect the code to indentify potential problems
- Verification Tools that can help
  - E.g., Coverity from Synopsys
  - Perform static analysis on source code
  - source code -> AST -> search for dagenours functions and patterns

Runtime Defense

- Why we need runtime defenses?
  - Principle: multi-layer defense (i.e., defense in-depth)
  - Sometimes above techniques may fail, e.g., return-to-libc attack
- Three typical methods
  - ASLR: Address Space Layout Randomization
  - StackGuard
  - Libsafe

Return-to-libc Attack

- No-Execution-Bit is not enough: vulnerable to return-to-libc attack
  - NX methods does not prevent code (or data) injection. It only prevents execution of injected code
  - But attackers
    can inject arguments in stack but execute existing code in TEXT segment

Original Vulnerable	When Overflowed	What exec() will see
Callee's Local Buffers		
Saved EBP(caller)		
Return Address	Addr of exec() in libc	Saved EBP (caller)
Arg 1 for callee()	Fake Return Addr for exec()	Fake Return address when finish
Arg 2 for callee()	Argument 1 for exec()	Argument 1 for me
... ...	Argument 2 for exec()	Argument 2 for me

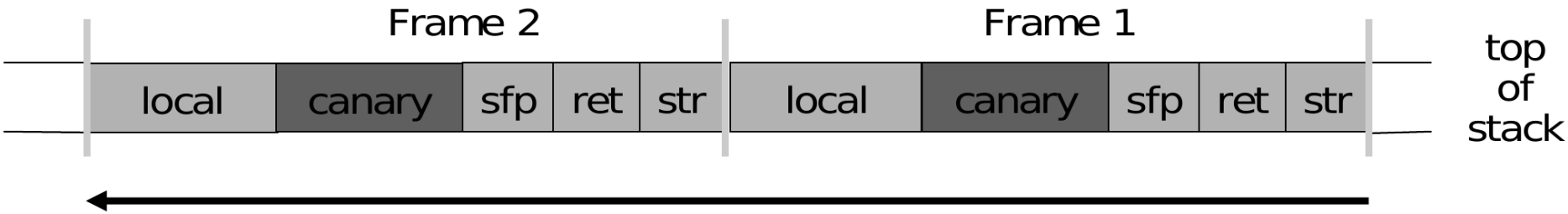
- Why we need a fake r address?
- Exec() will assume it i reached via a CALL instruction, so there is retum address in stac
- But in fact it is reache through RET instructi (which will not push a instruction but will pop retum address from s instead instead

Solution? Randomize the Address Space

- Return-to-libc attack depends on the knowledge of entry address of specific functions (e.g., exec())
  - If that address is unknown or wrong, then the attack will be reduced to DoS (due to corruption of execution flow)
- How to hide the entry address information?
  - Randomize the memory space layout (incl. shared library, stack)
  - The entry address of each object is different for each loading
- Demo
  - The memory layout information is in file /proc/pid/maps
  - Open two instance of the same program, say VIM, and observe their memory mapping information
- Question: Any patterns have you observed from above memory space layout?
  - Aligned to memory page size (normally 4KB)
  - Randomize within certain ranges (Windows 8 bits, Linux: 16 bit)
  - ASLR only increases the attacking complexity (Shacham et al. On the Effectiveness of Address-Space Randomization. CCS 2004)

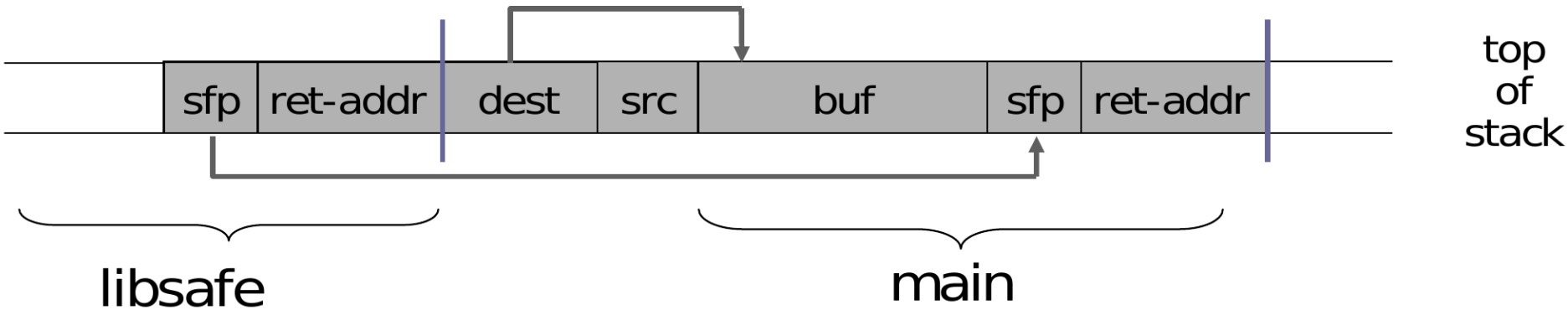
Runtime Checking: StackGuard

- Strategy: detect injections (by verify stack integrity)
  - Embed "canaries" in stack frames between BUFFER and RETURN address
  - Verify integrity of embedded canaries prior to function return
  - If canary integrity compromised, then terminate
  - Compiler Support GCC -fstack-protector option, MS Visual Studio: /GS



Runtime Checking: Libsafe

- Strategy: dynamically load a safer library (with bounds checking)
  - Need to recompile the source code
  - Intercept calls to dangerous functions (e.g., strcpy())
  - Validates if sufficient spaces are available in current stack frame
  - Continue if yes, otherwise will terminate the application



Conclusions

- Key Expectations on Learning Outcome
  - Be able to draw stack structure for a give piece of code
  - Understand why buffer overflow attacks are possible
  - Be able to analyze and idenfity potential overflow risks
  - Understand different strategies to defense buffer overflow attack
  - Be familary with some common defense techniques, e.g., NX, ASLR
- Next Lecture: Software Security 2 $\rightarrow$ Malicious Software
  - To understand different types of malicious code
  - How it works, what are the potential harms, and how to defense