Software vulnerabilities& Malicious software

EECS 195

Spring 2019

Zhou Li

Objectives

- Learn about memory organization, buffer overflows, and relevant countermeasures
- Common programming bugs, such as off-by-one errors, race conditions, and incomplete mediation
- Survey of past malware and malware capabilities
- Virus detection
- ~~Tips for programmers on writing code for security~~

Software security?

Leo, doctor

The system is hacked. Find out why?

OMG. Let me try.

Jim, programmer

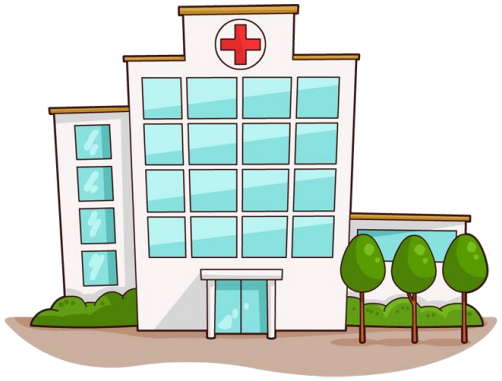
Security tools

Authentication

Access Control

Cryptography

vulnerability



Hospital Information System

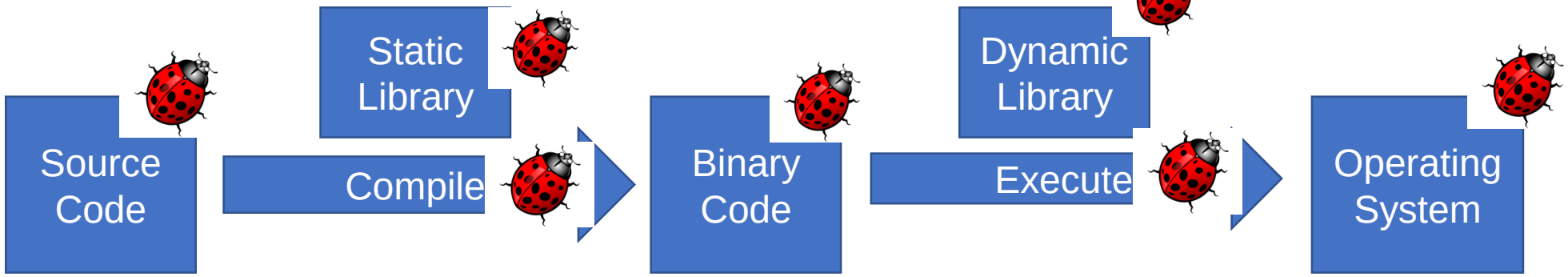
malware (threat)

vulnerability

Shared workstation

Programs and unintentional oversights

- Program
  - Implementation of algorithms/specifications/functionalities
  - Source code (C, C++, Java, …)
  - Binary code (after compilation)
- Unintentional oversights
  - Human error => software flaw (vulnerabilities) => exploitation



Types of software vulnerabilities

- Buffer overflows
- TOCTTOU
- Undocumented access points (backdoors)
- ~~Off-by-one errors~~
- Integer overflows
- ~~Unterminated null-terminated string~~
- ~~Parameter length, type, or number errors~~
- Unsafe utility libraries
- Race Condition
- …

Buffer overflows

- Oversights to document or check excessive data
- Attacker's inputs are expected to go into regions of memory allocated for data, but those inputs are instead allowed to overwrite memory holding executable code
- The trick for an attacker is finding buffer overflow opportunities that lead to overwritten memory being executed, and finding the right code to input
- Break access control on code execution and lead to privilege escalation

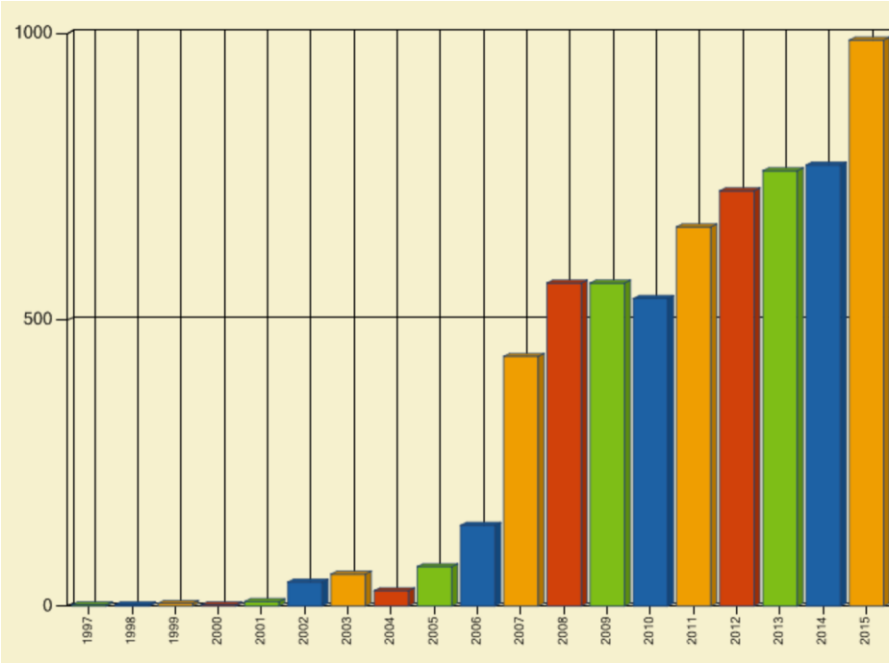
Buffer overflows (cond.)

Extremely common bug in C/C++ programs.

- First major exploit: 1988 Internet Worm. Fingerd.

whenever possible avoid C/C++

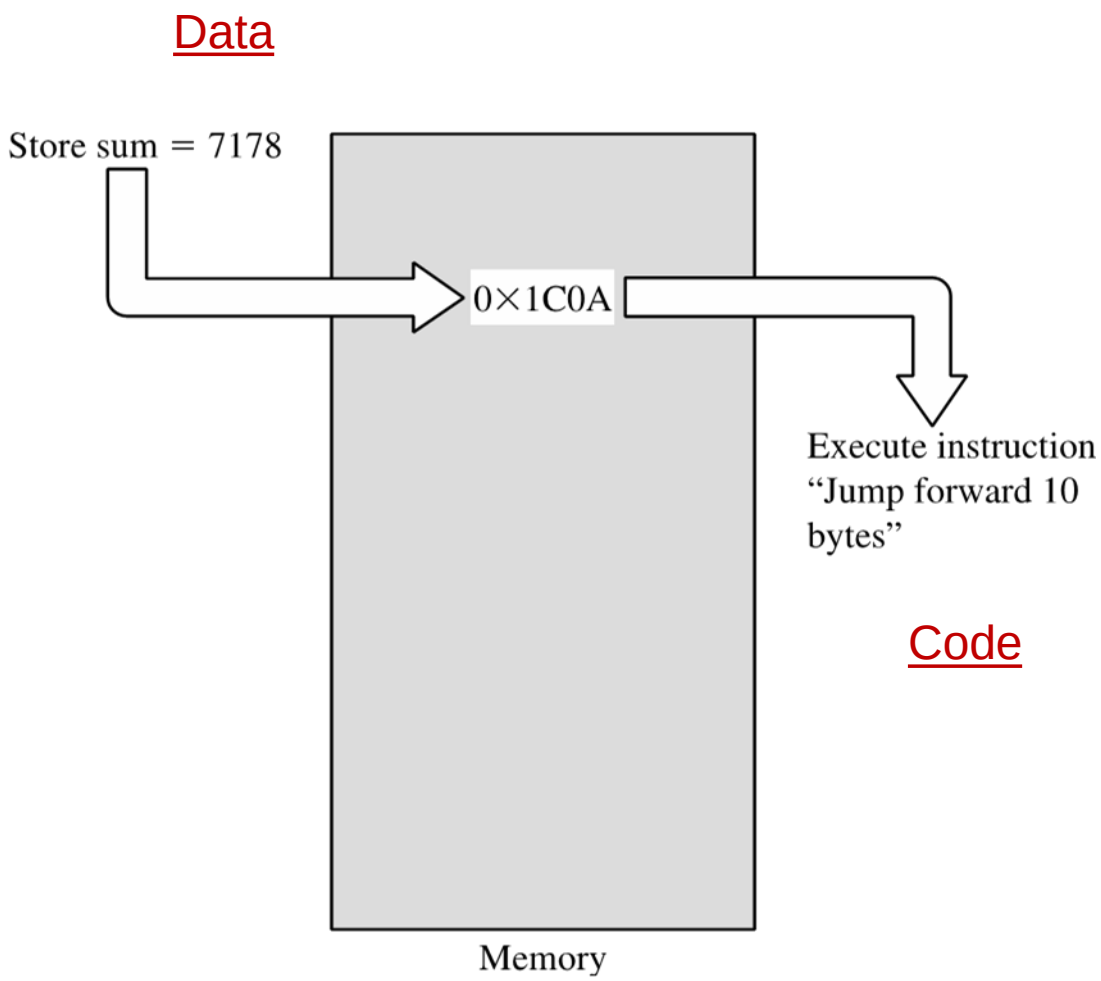
… but often cannot avoid C/C++
Need to understand
attacks and defenses



Source: web.nvd.nist.gov

Background

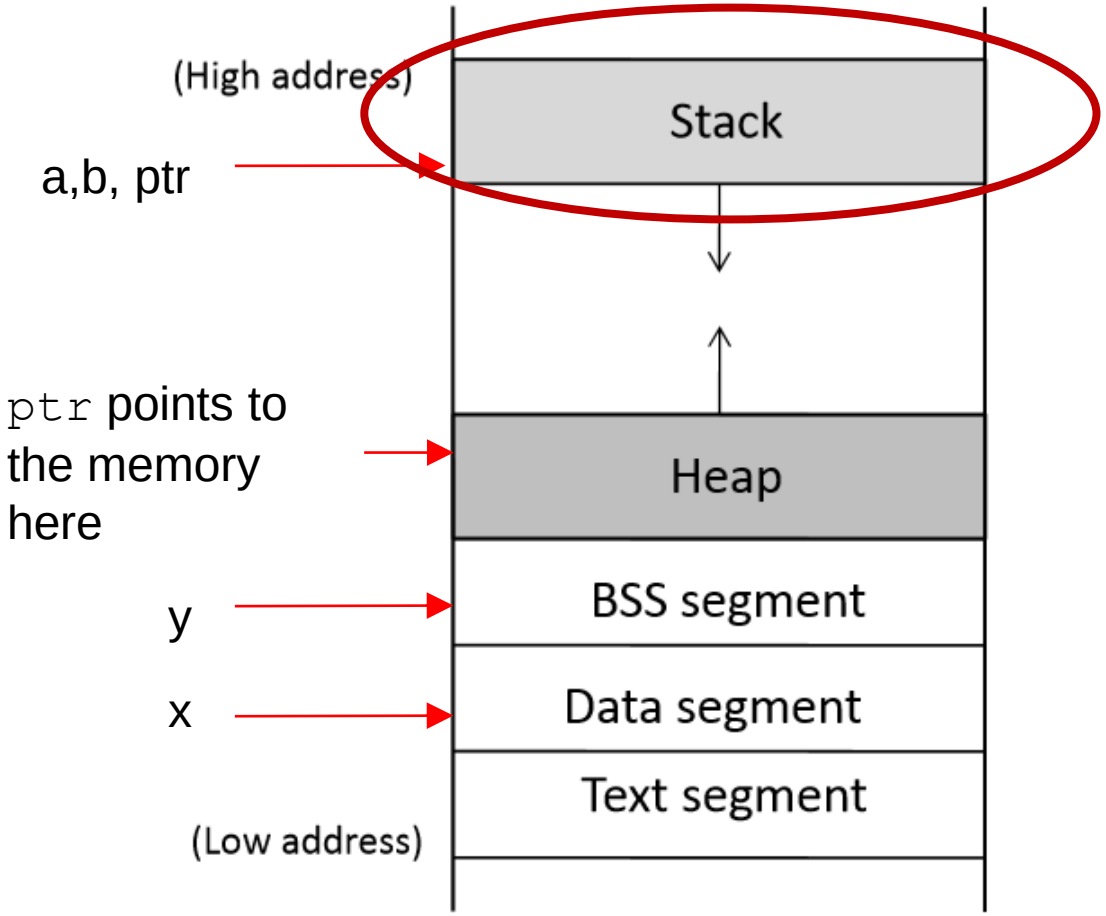
- Memory
  - Holding code & data
  - Code is indistinguishable from data in memory representation
  - Code & data can be referenced through address or CPU register
  - Both OS and user applications co-exist in memory (different space)
  - Isolation & access control (e.g., page table) at hardware/OS level to prevent unauthorized access



Program Memory Stack

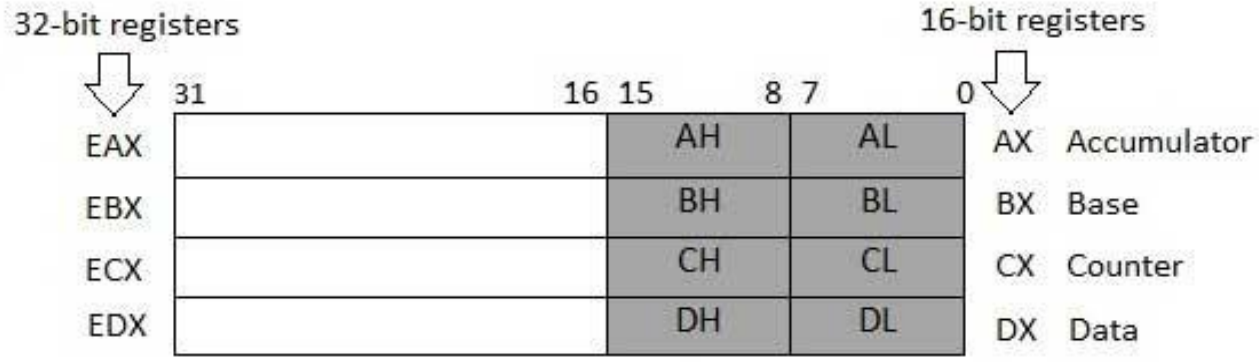
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```

Buffer overflows



Registers

- Data registers
  - EAX, EBX, ECX, EDX
  - Many are used for function parameters
- Pointer registers
  - EIP (Instruction Pointer): stores the offset address of the next instruction to be executed
  - ESP (Stack Pointer)
  - EBP (Base Pointer)
- Index registers
- Control registers
- Segment registers



Stack Frame

- EBP: Base Pointer
  - Points to previous frame pointer
  - EBP+offset: to locate variables
  - The return address will always be at EBP+4, the first parameter will always be at EBP+8, and the first local variable will be at EBP-4 (or EBP-8).

- ESP: Stack Pointer, pointing to the stack top (low address)
  - shifted when POP&PUSH

https://practicalmalwareanalysis.com/2012/04/03/all-about-ebp/

Order of the function arguments in stack

```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

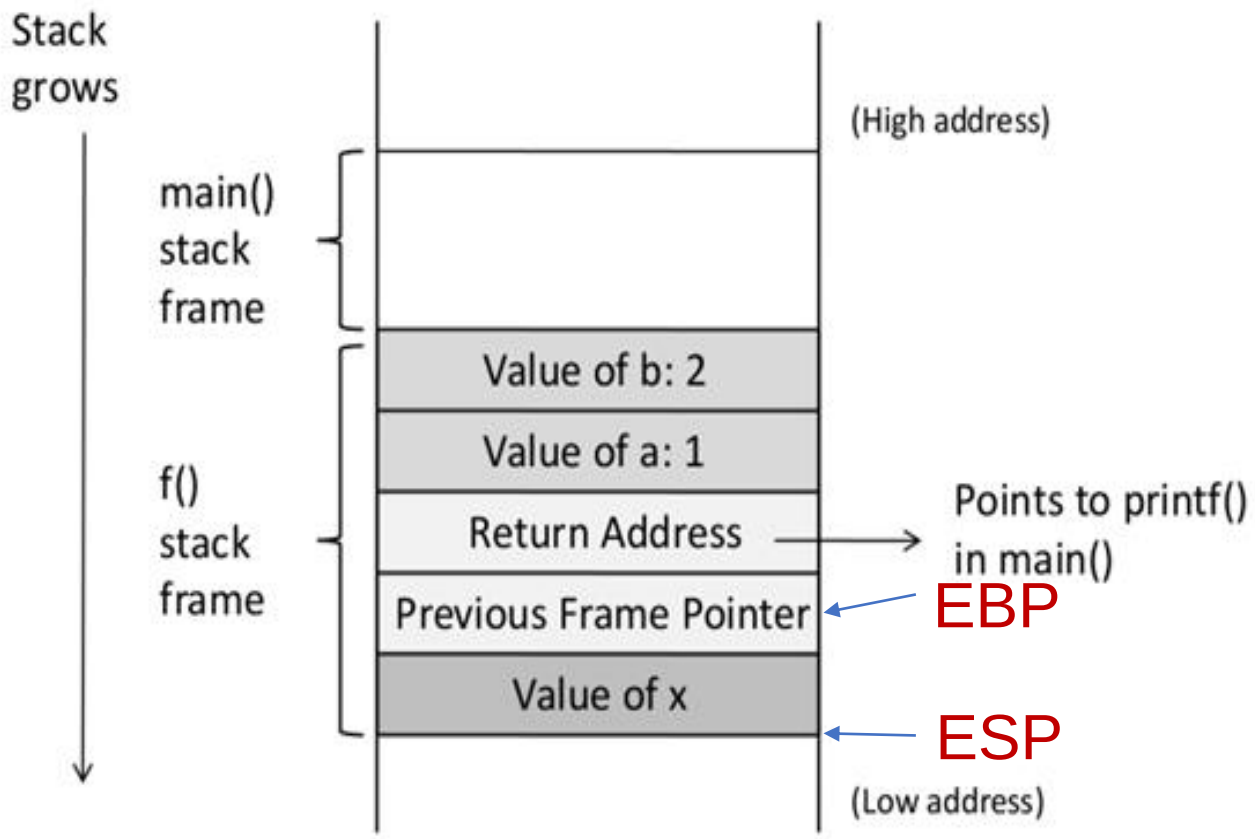
gcc -S <filename>: c to assembly

x = a + b

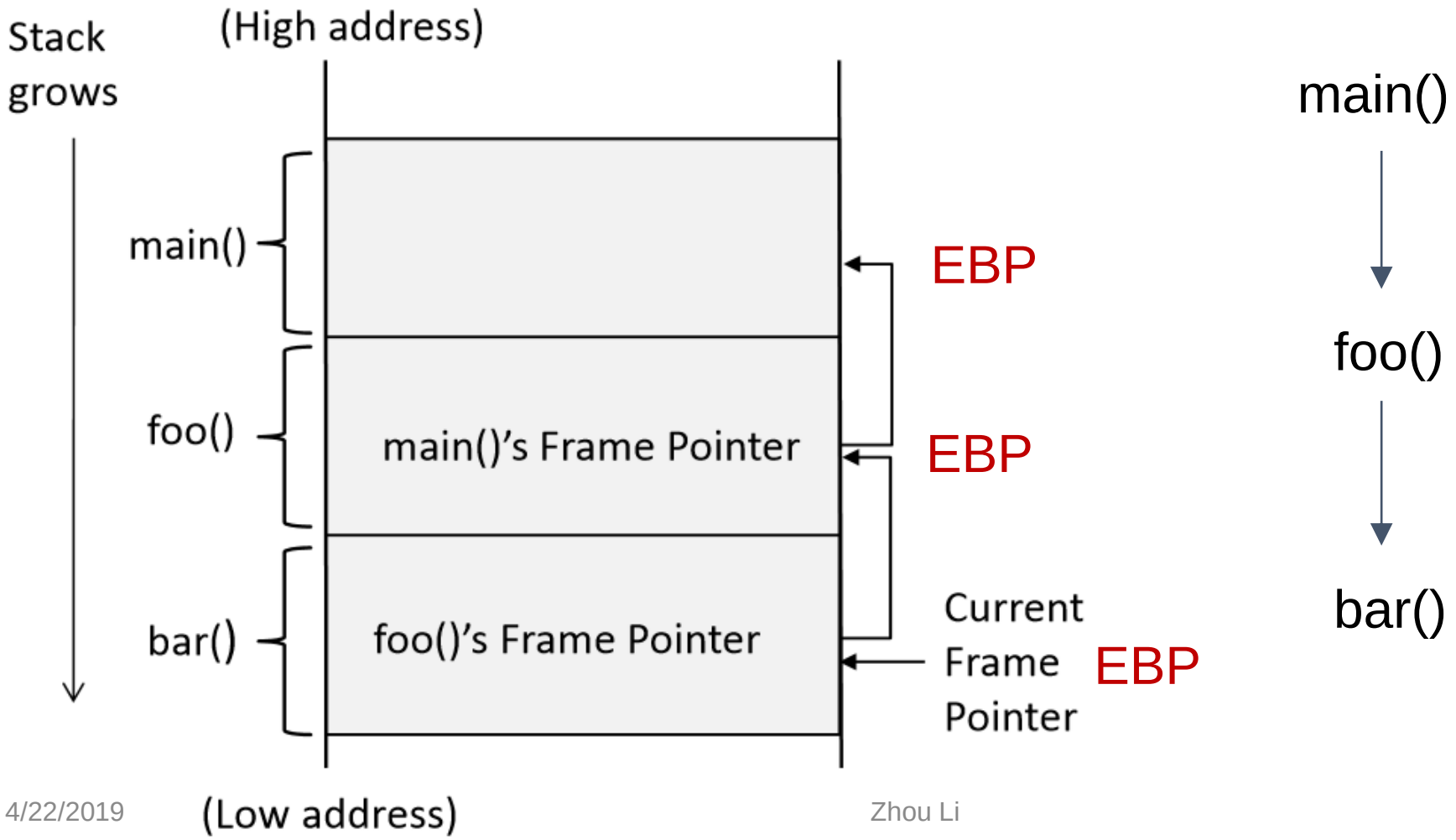
```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



Stack Layout for Function Call Chain



Vulnerable Program

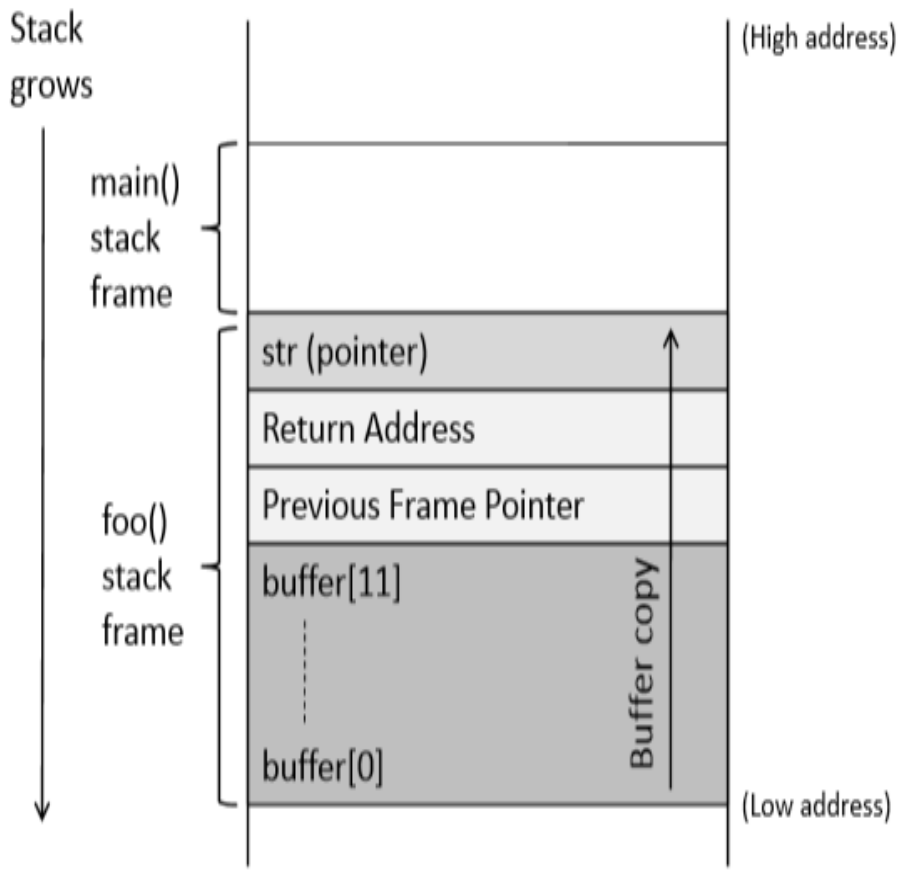
```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

- Reading 300 bytes of data from badfile.
- Storing the file contents into a str variable of size 400 bytes.
- Calling foo function with str as an argument.

Note : Badfile is created by the user and hence the contents are in control of the user.

Vulnerable Program

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```



Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- Attacker's code ⟶ Malicious code to gain access

How to Run Malicious Code