Buffer Overflow Attack

Lecture 2

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

MARSHALL

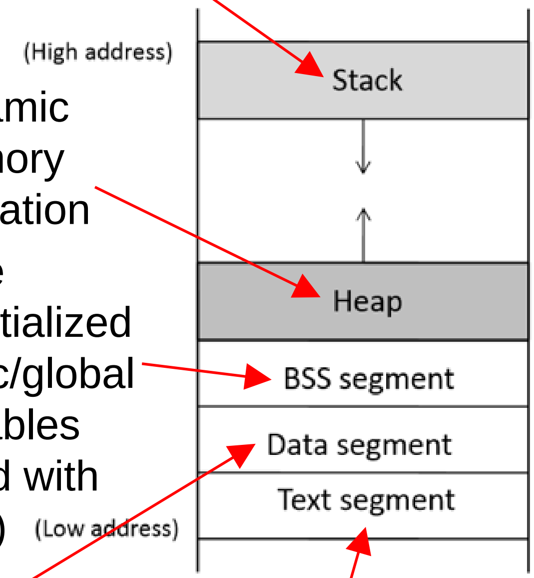
Introduction

- famous buffer overflow attacks
  - Morris worn (1988)
    - buffer overflow in the fingerd network service
  - Code Red worm (2001)
    - execute arbitrary code and infect the machine with the worm
  - SQL Slammer (2003)
    - generate random IP addresses and send itself out to those addresses
  - Stagefright attack against Android (2015)
    - allows adversary to perform arbitrary operations on the victim's device
  - more…

Program Memory Layout

- prerequisite of understanding buffer overflow attack:
  - understanding how the data memory is arranged inside a process
- when program running, needs memory space to store data
  - for C program, its memory is divided into five segments
    - text segment
    - data segment
    - BSS segment
    - heap
    - stack

store local variables defined inside functions, and function-related data (return address)



dynamic memory allocation

store uninitialized static/global variables (filled with zero)

store static/global variables

store executable code of program (read-only)

Program Memory Layout (cont.)

allocates size bytes of uninitialized storage

1 arg: number of bytes to allocate

ref: https://en.cppreference.com/w/c/memory/malloc

```
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;
}
```

pointer variable

int pointer

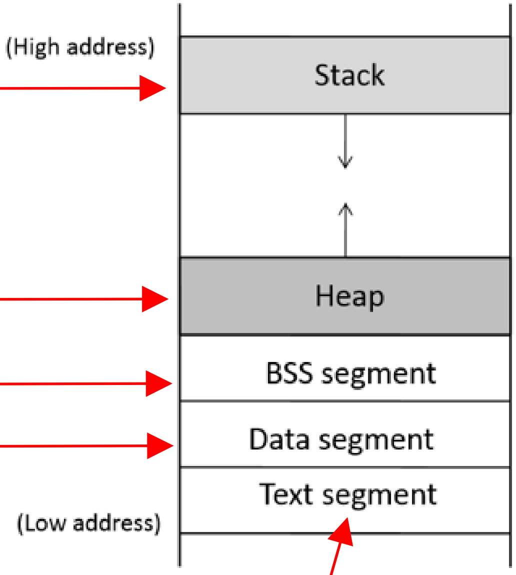
return the size of data type

a, b ,ptr

ptr points to the memory here

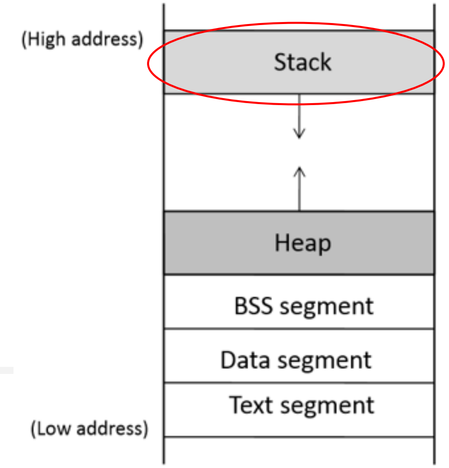
y

x



store executable code of program (read-only)

Stack Memory Layout



- stack: store data used in function invocations
- a program executes as a series of function calls (execution)
  - when function is called, space is allocated for it on the stack
  - e.g.,

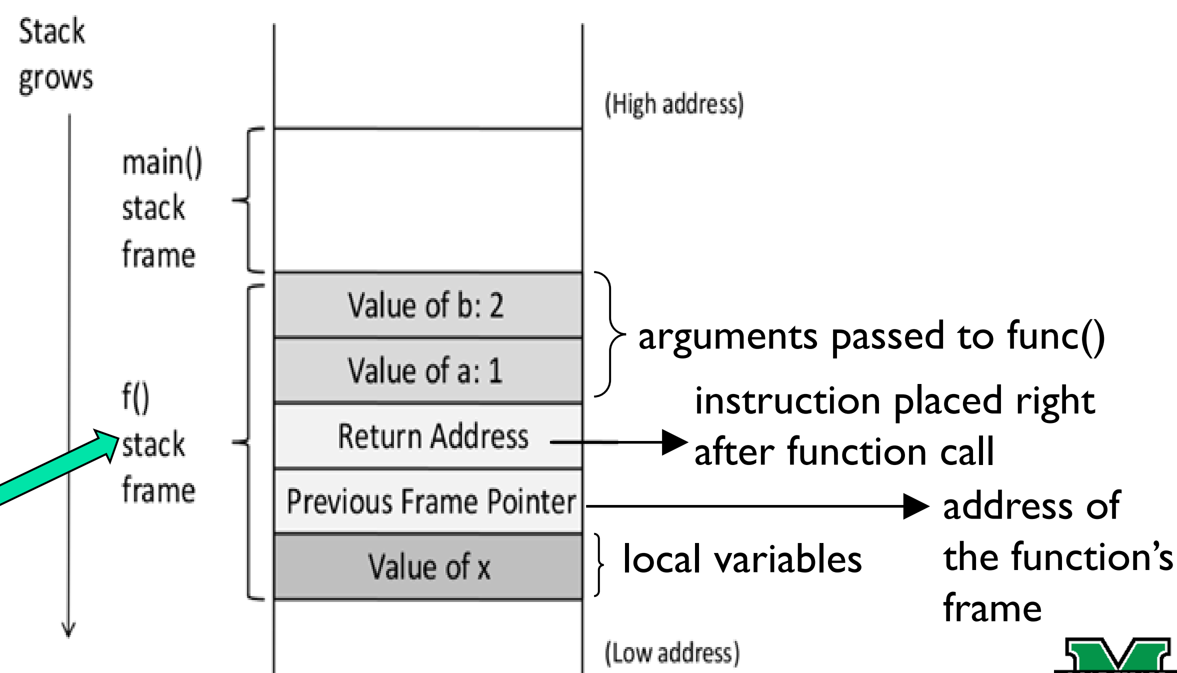
```
void func(int a, int b)
{
   int x, y;

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

two integer arguments: a and b

two integer local variables: x and y

when func() is called, stack frame is allocated



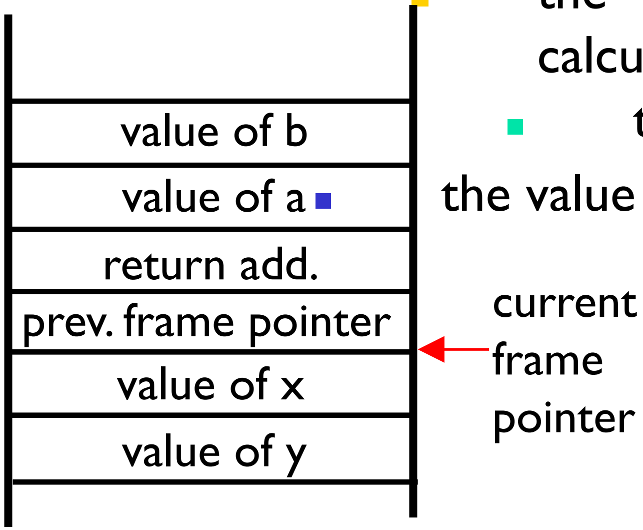
Frame Pointer

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

store result get value

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

- inside func(), how to access arguments and local variables?
  - only way: knowing their memory add.
  - issue: add. cannot be determined during compilation (compilers cannot predict run-time status of stack)
  - solution: frame pointer, a special register in CPU
    - points to a fixed location in stack frame
      - the add. of each argument and local variable can be calculated using frame pointer and add. offset
        - the value of offset can be decided during compilation
    - the value of frame pointer can change during run time

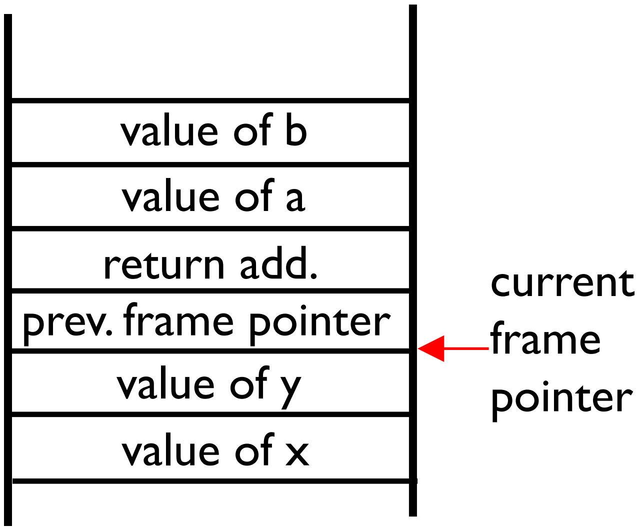



```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Frame Pointer

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

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



frame pointer register (x86 architecture)

```
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
```

eax and edx: general-purpose registers storing temporary values

12(%ebp): %ebp + 12

movl array_base(%esi), %eax
add the address of memory location array_base to the contents of number register %esi to determine an address in memory. Then move the contents of this address into number register %eax.

addl %edx, %eax
adds together its two operands (%edx and %eax), storing the result in its second operand (%eax)

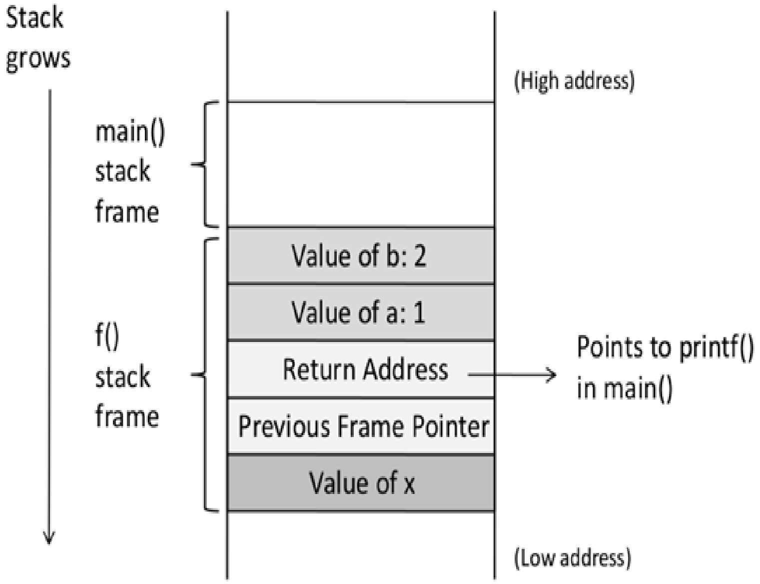
-8(%ebp): %ebp - 8

on 32-bit architecture, return address and frame pointer both occupy 4 bytes. So,
a is at ebp + 8
b is at ebp + 12

Function Call Chain

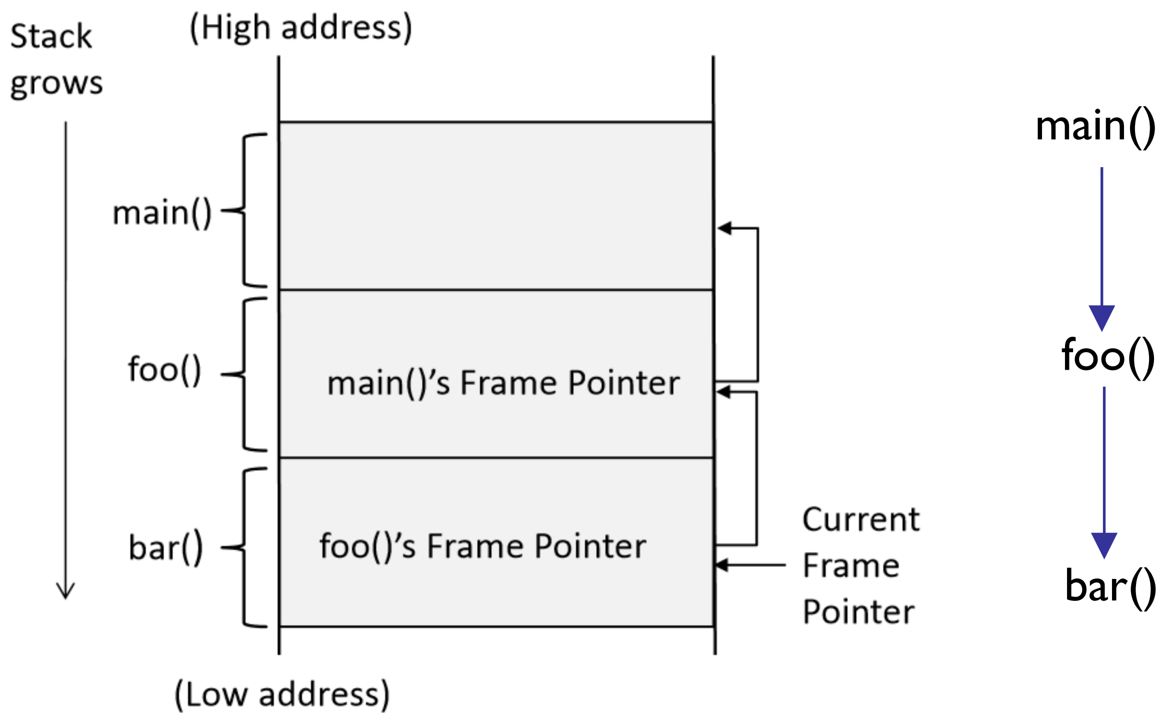
- call another function from inside a function
  - every time function is called, a stack frame is allocated on the top of stack
  - when function is returned (completed), the stack frame allocated for it is released
  - e.g.,

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



Function Call Chain (cont.)

- only one frame pointer register: always pointing to the frame of current function
- question: how the functions were called?



Stack Buffer-Overflow Attack

- memory copying: copying data from one place to another place
  - before copying, a program allocates memory space
  - issue: programmer fails to allocate *sufficient* amount of memory
    - consequence: more data is copied to the des. buffer than the amount of allocated space → buffer overflow
      - program crash (corruption of data beyond buffer)
      - gain control of program (attacker)
- some languages (e.g., Java) automatically detect the problem (buffer over-run), while many others (e.g., C) do not

Copying Data Causes Buffer Overflow

- strcpy()

char* strcpy(char* destination, const char* source):

- copies the string pointed by the source (including the null character) to the destination.
- when making copy, it stops when meets \0 (the end of string)

```
#include <string.h>
#include <stdio.h>

void main()
{
  char src[40] = "hello world \0 extra string";
  char dest[40];

  // copy to dest (destination) from src (source)
  strcpy(dest, src);
}
```

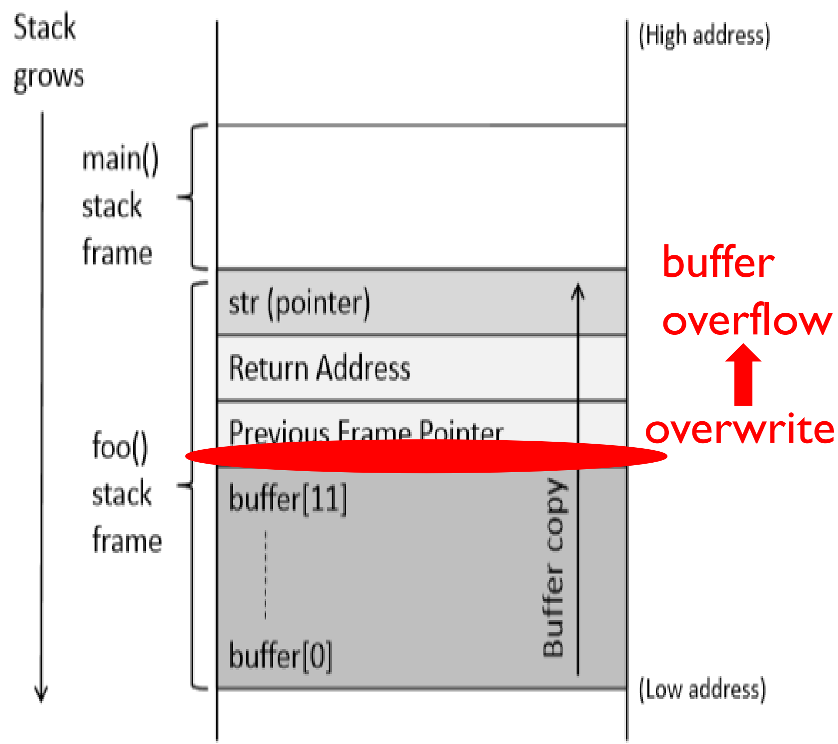
strcpy(dest, src); → only copy "hello world" to dest. why???

Copying Data Causes Buffer Overflow

- when copying data, what will happen if the string is longer than the size of buffer?

```
#include <string.h>

void foo(char *str)
{
  char buffer[12];
  strcpy(buffer, str);
}
void main()
{
  char *str = "This is definitely longer than 12";
  foo(str);
}
```



Exploiting Buffer Overflow Vulnerability

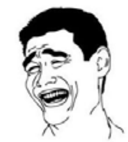
- overflowing buffer:
  - cause program crash
  - run some other code (more interesting to attacker)
    - if attackers control what code to run, they can hijack the execution of programs
      - privilege escalation for attackers

Exploiting Buffer Overflow Vulnerability (cont.)

```
#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;
}

void 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");
}
```

do you know what inside?



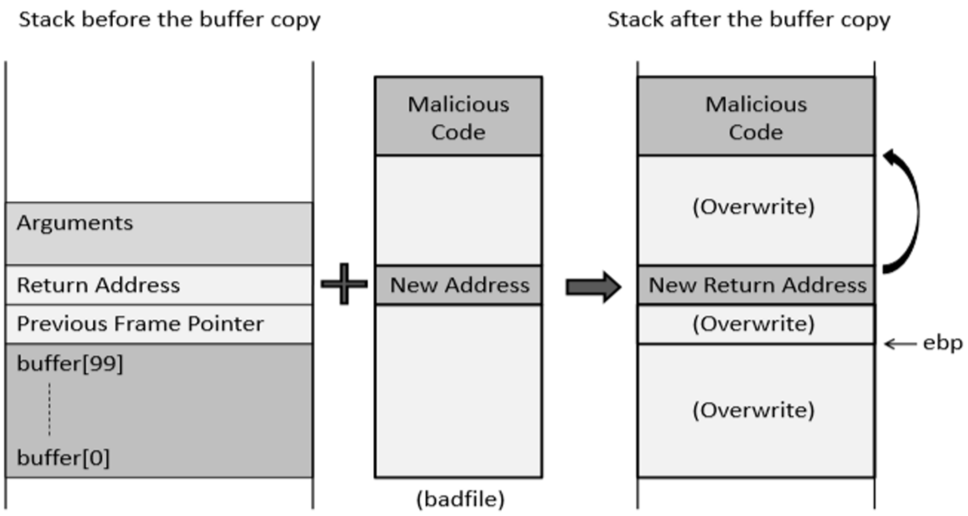
open file "badfile" to read.

read 300 bytes and copy data to 100 bytes buffer

the content is copied to buffer from "badfile"

Exploiting Buffer Overflow Vulnerability (cont.)

get code into memory of running program:
- not difficult:
  - place code in "badfile"
  - let program read "badfile"
    - program copies code to buffer



force program to jump to our code (already in memory)
- using buffer overflow
  - overwrite return add. field
    - use add. of malicious code to overwrite
  - when foo() returns, it jumps to new add. (add. of malicious code)

Setup for Environment

- attack environment: Ubuntu
- buffer overflow has a long history, so many OS have countermeasures against it
- to simplify environment
  - turn off countermeasures
  - later on, turn them back on to show
    - countermeasures only make buffer overflow more difficult, not impossible

Disable Address Randomization

- address space layout randomization (ASLR): countermeasure to buffer overflow
  - randomizing the memory space of key data areas in process
    - the base of executable
    - the positions of stack, heap, and libraries
  - making it difficult for attackers to guess the add. of injected malicious code

Disable Address Randomization

- turning countermeasure off

```
% sudo sysctl -w kernel.randomize_va_space=0
```

- goal: exploit buffer overflow vulnerability in Set-UID root program
  - a Set-UID root program runs with root privilege when executed by normal user
    - assigning normal user extra privileges
  - if buffer overflow vulnerability is exploited in privileged Set-UID root program
    - consequence: the injected malicious code can run with root's privilege

Vulnerable Program: stack.c

- compile set-uid root version of program

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

make stack executable (-z execstack)

turn off Stack-Guard (countermeasure) (-fno-stack-protector)

- 1st command: compiles stack.c program
- 2nd and 3rd commands: turn executable stack into root-owned set-uid program
  - the order of 2nd and 3rd commands cannot be reversed

sudo chown root stack: change ownership

sudo chmod 4755 stack: sets permissions (read, write, execute)

```
#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;
}

void 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");
}
```

Vulnerable Program: stack.c

- badfile: contains random contents
  - when the size of file is less than 100 bytes, the program runs properly
  - when the size of file is larger than 100 bytes, the program crashes
    - buffer overflow happens

```
$ echo "aaaa" > badfile
$ ./stack
Returned Properly
$
$ echo "aaa … (100 characters omitted) … aaa" > badfile
$ ./stack
Segmentation fault (core dumped)
```