IERG 4130 Tutorial 4

The Chinese University of Hong Kong

SONG Zirui

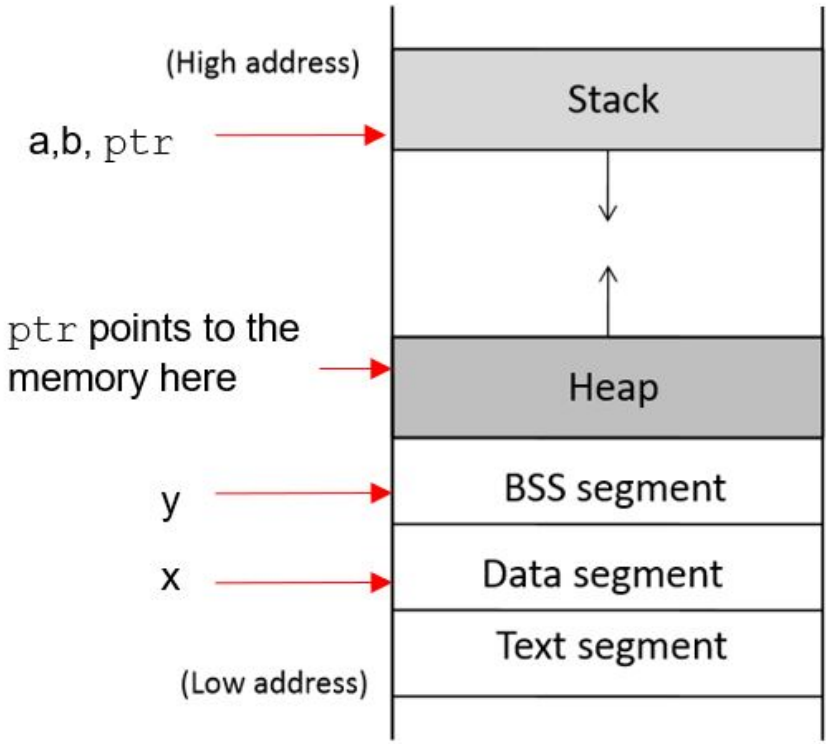
October 6, 2022

Outline

Overview: Program Memory Stack

- Different variables are stored in different locations

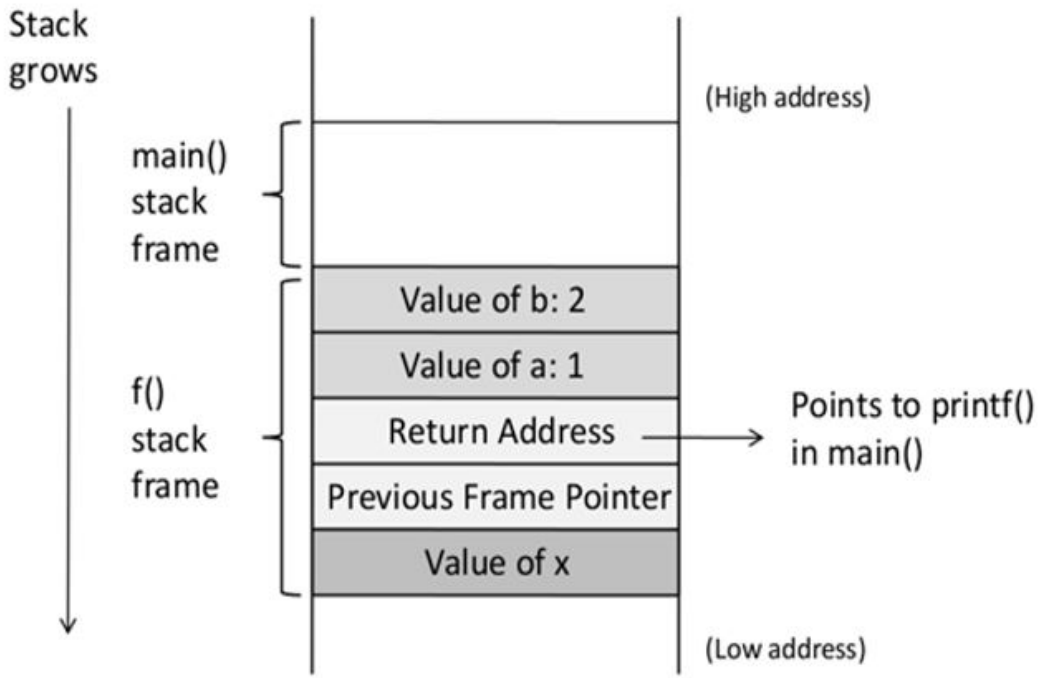
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // 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;
}
```



Overview: Function Call Stack

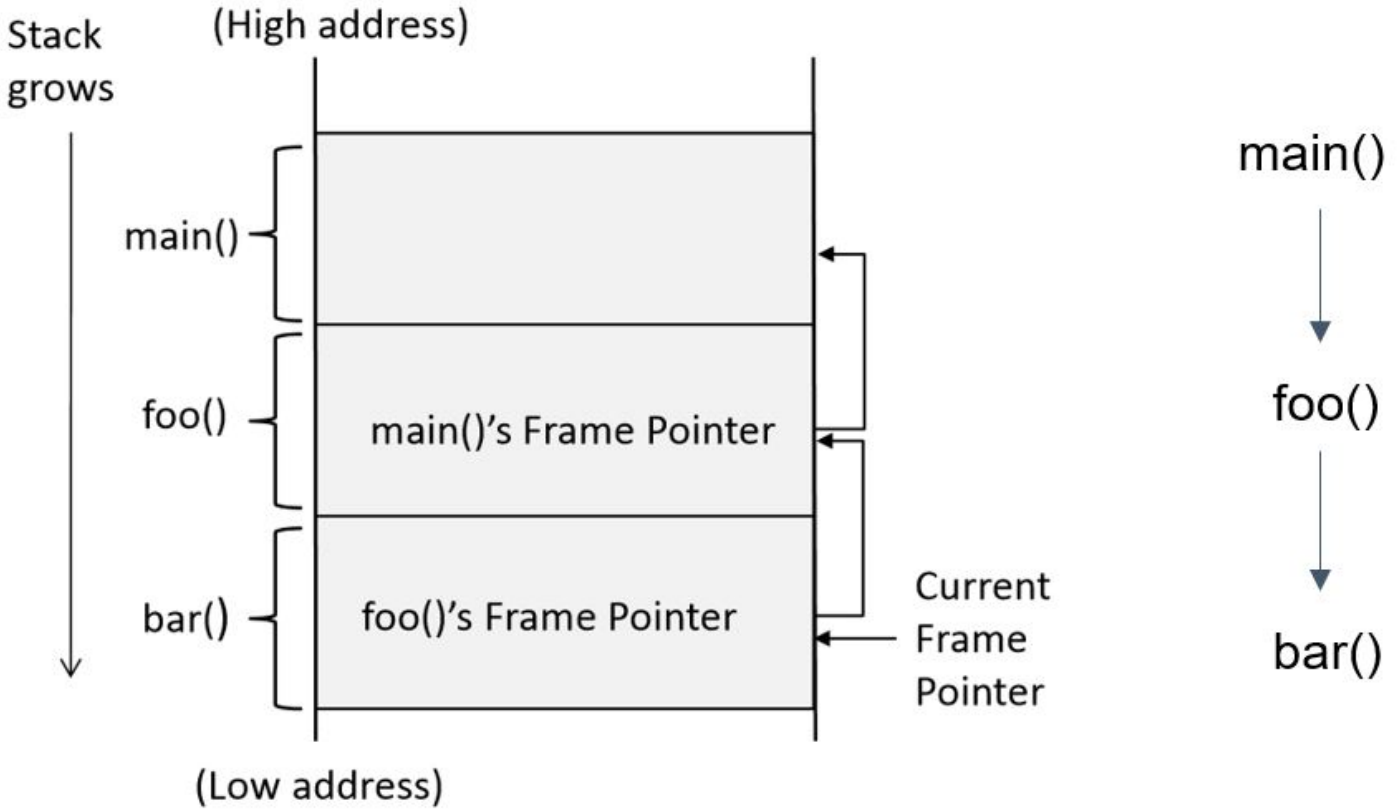
- If we consider the function call...

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



Overview: Stack Layout for Function Call Chain

- If we consider more function calls in a function call chain...



Overview: Shellcode

- **Aim of the malicious code**: Allow to run some commands to gain access of the system
- **Solution**: Shell program

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

- Assembly code (machine instructions) for launching a shell
- Goal: Use `execve("bin/sh", argv, 0)` to run shell

Overview: Shellcode

- The details of the given shellcode
- Assembly version of `execve("bin/sh", argv, 0)`

```
const char code[] =
  "\x31\xc0"          /* xorl    %eax,%eax     */   ← %eax = 0 (avoid 0 in code)
  "\x50"              /* pushl   %eax          */   ← set end of string "/bin/sh"
  "\x68""//sh"        /* pushl   $0x68732f2f   */
  "\x68""/bin"        /* pushl   $0x6e69622f   */
  "\x89\xe3"          /* movl    %esp,%ebx     */    ← set %ebx
  "\x50"              /* pushl   %eax          */
  "\x53"              /* pushl   %ebx          */
  "\x89\xe1"          /* movl    %esp,%ecx     */    ← set %ecx
  "\x99"              /* cdq                   */    ← set %edx
  "\xb0\x0b"          /* movb    $0x0b,%al     */    ← set %eax
  "\xcd\x80"          /* int     $0x80         */    ← invoke execve()
;
```

Buffer-Overflow: The Vulnerable Program

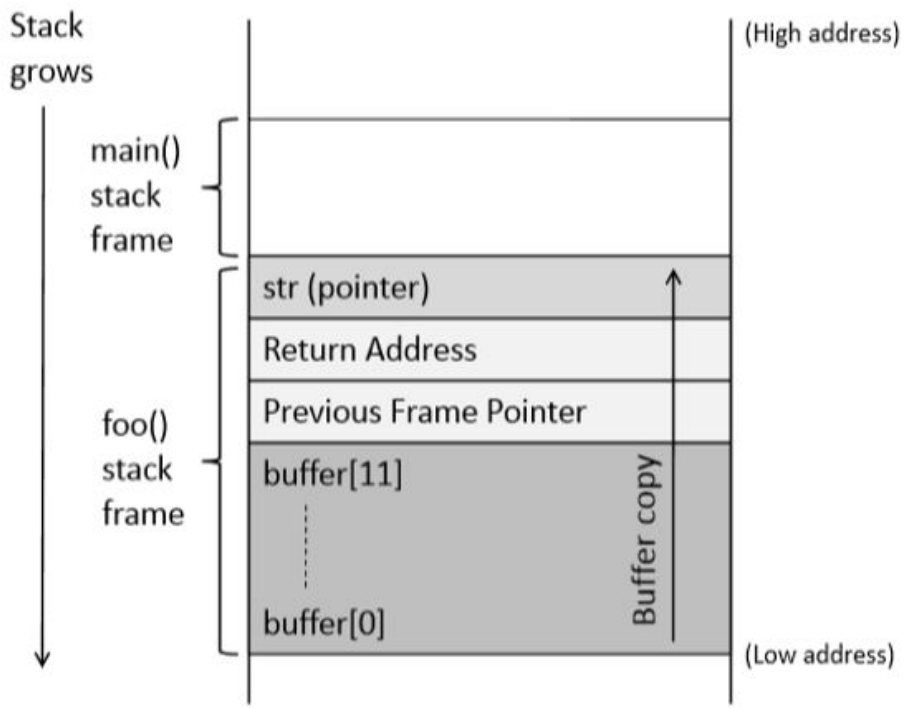
- Read data from badfile
- Storing the file contents into a str
- Calling bof function with str as an argument
- **Note**: Badfile is created by the user and hence the contents are in control of the user

```
int main(int argc, char **argv)
{
   char str[404];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 404, badfile);
   bof(str);
   printf("Returned Properly\n");
   return 1;
}
```

Buffer-Overflow: The Vulnerable Program

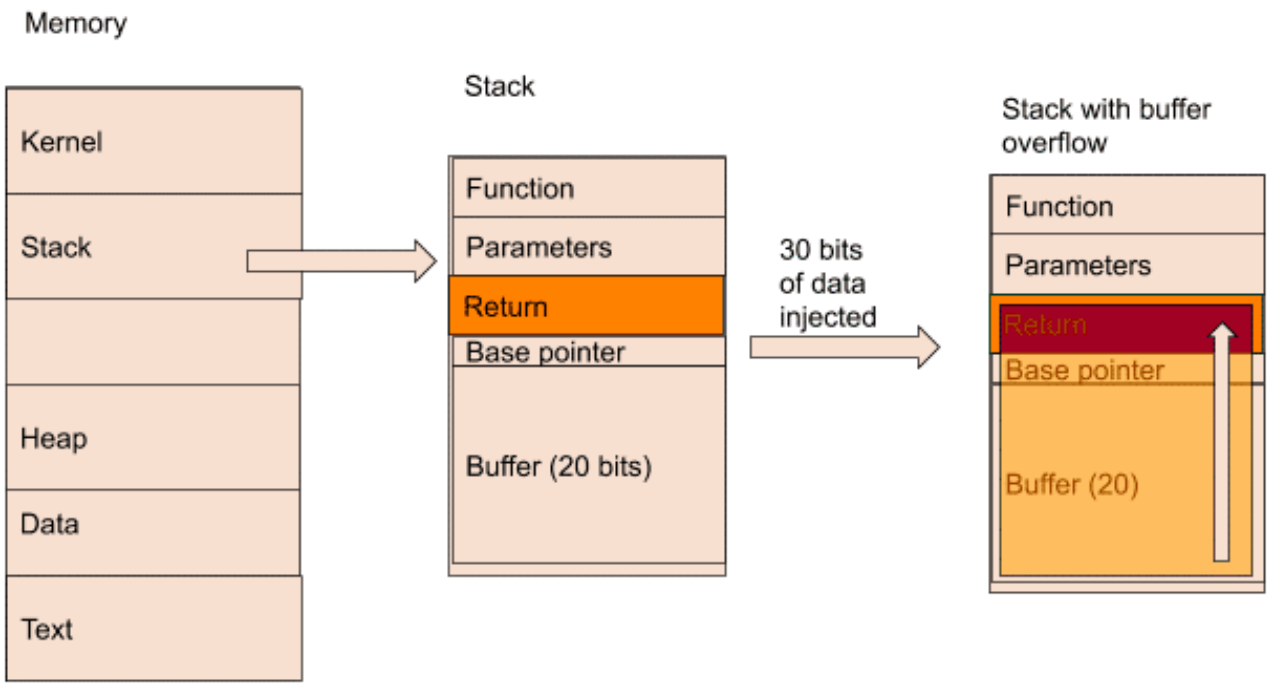
- bof() have 128 bytes buffer
- But write 404 bytes in to...

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[128];

    /* The following statement has a buffer overflow
    strcpy(buffer, str); ①

    return 42;
}
```



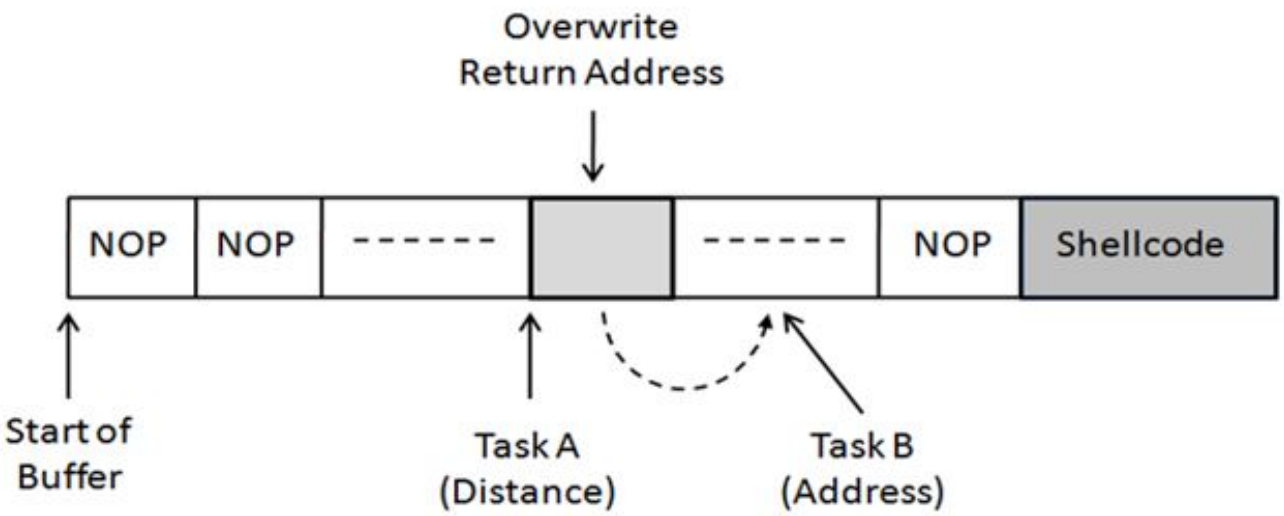
Buffer-Overflow: How to attack

- Core idea: Make the program executes the shellcode
- Task A: replace the return address
- Task B: append shellcode into the buffer (for further execution)



Buffer-Overflow: How to attack

- Task A: replace the return address
  - Challenge: Find the offset distance between the base of the buffer and return address
- Task B: append shellcode into the buffer (for further execution)
  - Challenge: Find the address to place the shellcode



Buffer-Overflow: How to attack

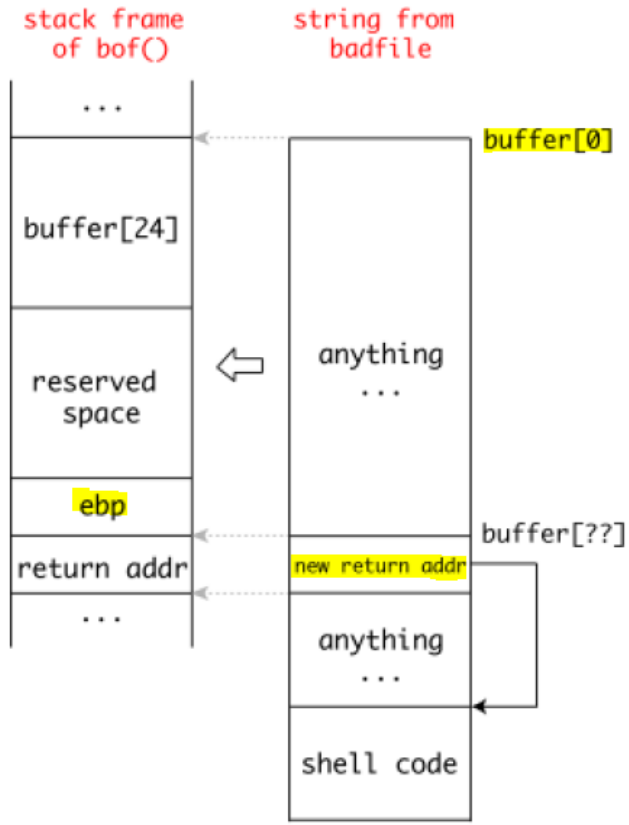
- Task A: replace the return address
  - **Hint**: use gdb to get the addresses
  - You need to exploit the vulnerability in stack.c (not exploit.c)

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
......
(gdb) b foo          ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
......
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
```

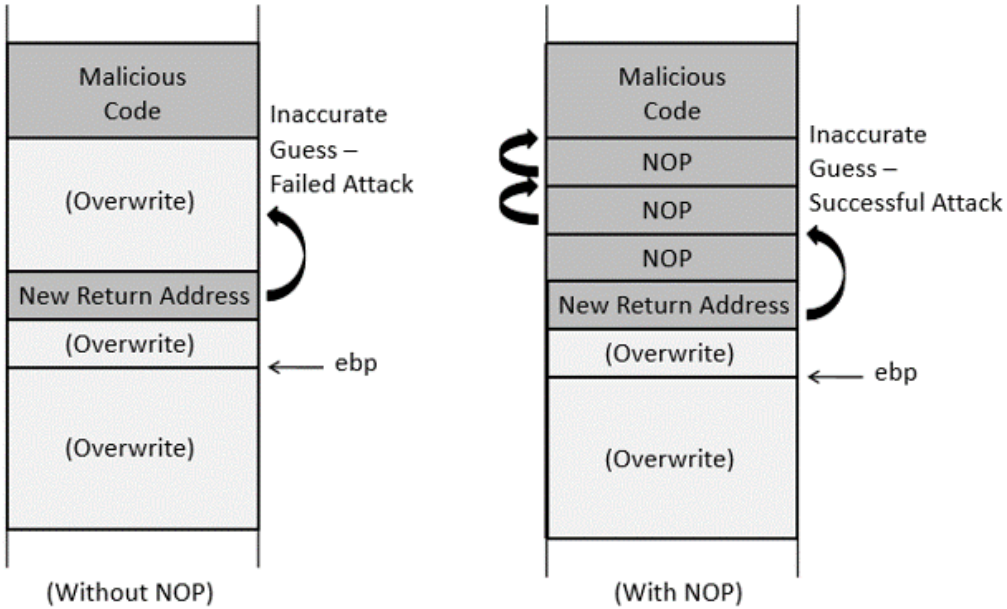
Buffer-Overflow: How to attack

- Task A: replace the return address
  - Address of buffer[0]: X
  - Address of ebp: Y
  - ebp: 4 bytes
  - buffer[??] = buffer[Y - X + 4]



Buffer-Overflow: How to attack

- Task B: append shellcode into the buffer (for further execution)
  - **Hint**: use memcpy(dest, src, N)
  - To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer.



Buffer-Overflow: Countermeasure

- Developer approaches:
  - Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying
- OS approaches:
  - ASLR (Address Space Layout Randomization)
- Compiler approaches:
  - Stack-Guard
- Hardware approaches:
  - Non-Executable Stack

Buffer-Overflow: Countermeasure

- ASLR (Address Space Layout Randomization)
  - To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes

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

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

Buffer-Overflow: Countermeasure

- ASLR (Address Space Layout Randomization)

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

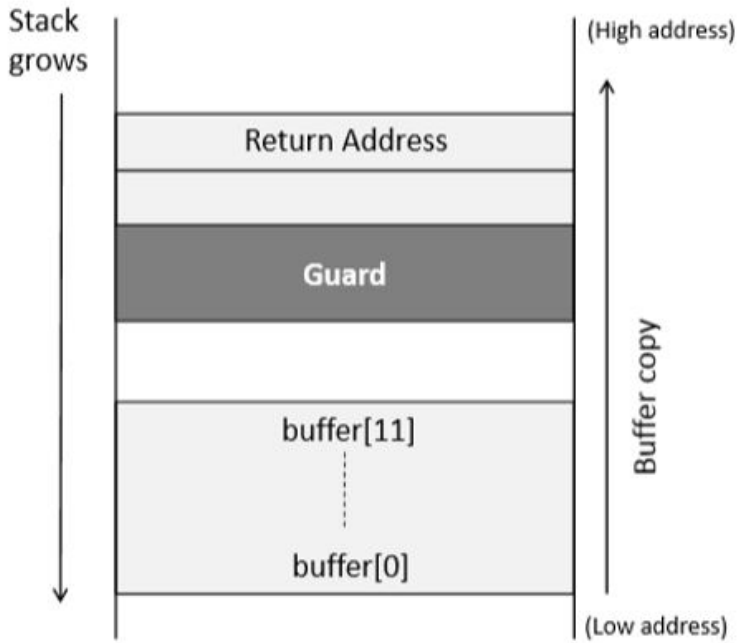
```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

3

Buffer-Overflow: Countermeasure

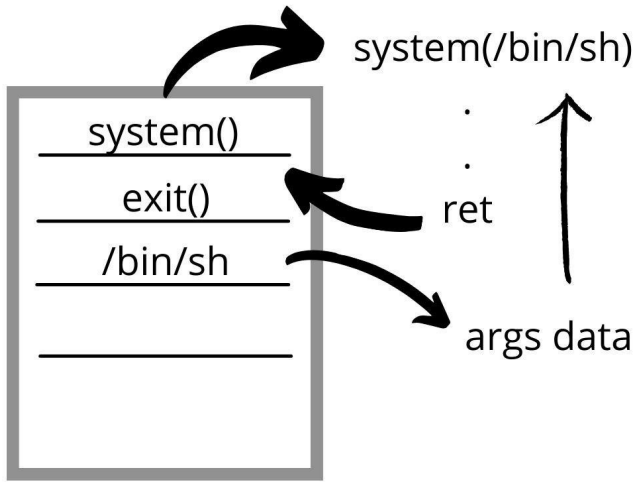
- Stack-Guard
  - A random secret value is generated and stored in memory
  - Then be assigned to a local variable which gets stored in the stack
  - When calling copy function, this secret value will be checked

```
void foo (char *str)
{
   int guard;
   guard = secret;

   char buffer[12];
   strcpy (buffer, str);

   if (guard == secret)
      return;
   else
      exit(1);
}
```



Buffer-Overflow: Countermeasure

- Non-executable stack
  - NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable
  - This countermeasure can be defeated using a technique called Return-to-libc attack



$ shell spawned

Overview: Format String

- `printf()` - To print out a string according to a format

```
int printf(const char *format, ...);
```

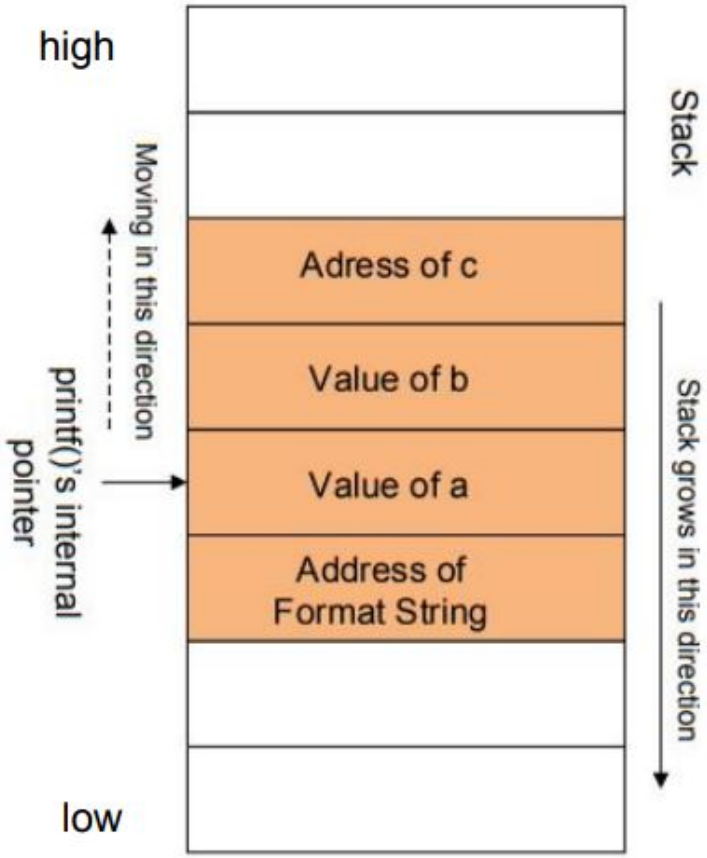
- The argument list of `printf()` consists of
  - One concrete argument format
  - Zero or more optional arguments
- Compilers don't complain if less arguments are passed to `printf()` during invocation

Format String: Vulnerable Program's Stack Example

- If we have a case like:

```
printf ("a value %d,
        b value %d,
        c address %08x",
        a, b, &c);
```

- What if miss the a, b, &c?
  - Check data on the stack
  - Overwrite memory data



Format String: How to attack

- What can we achieve by using this?
  - Attack 1 : Crash program
  - Attack 2 : Print out data on the stack
  - Attack 3 : Change the program's data in the memory
  - Attack 4 : Change the program's data to specific value
  - Attack 5 : Inject Malicious Code
- **Hint: try to use some conversion specifiers**

d, i	signed int type, decimal
o, u, x, X	unsigned int, octal(o), unsigned decimal(u), hexadecimal notation(x or X)(uppercase for X, lowercase for x)
f, F	double
n	the number of characters successfully written so far
s	a pointer to the initial element of an array

Figure: Most-used conversion specifiers

Format String: How to attack

- Attack 1 : Crash program
  - **Hint:** `printf()` will treat %s as address
  - i.e., `printf()` will fetch data from "this" address
- Attack 2 : Print out data on the stack
  - **Hint:** `printf()` treats %s as hexadecimal notation
  - i.e., `printf()` will fetch "this" data from the stack
- Attack 3, 4, 5: Try to find solutions by yourself!

d, i	signed int type, decimal
o, u, x, X	unsigned int, octal(o), unsigned decimal(u), hexadecimal notation(x or X)(uppercase for X, lowercase for x)
f, F	double
n	the number of characters successfully written so far
s	a pointer to the initial element of an array

Figure: Most-used conversion specifiers

Format String: Countermeasure

- Developer approach: IT WORKS!
  - Avoid using untrusted user inputs for format strings in functions like `printf`, `sprintf`, `scanf`, ...
- Address randomization (ASLR): IT WORKS!
  - Makes it difficult for the attackers to guess the address of the target memory (return address, address of the malicious code)
- Stack-Guard: NOT WORK!
  - We can ensure that only the target memory is modified; no other memory is affected.
- Non-executable Stack/Heap (NX-bit): NOT WORK!
  - Attackers can use the return-to-libc technique to defeat the countermeasure

Lab Tasks Q&A