Software vulnerabilities: memory corruption

Classifying vulnerabilities

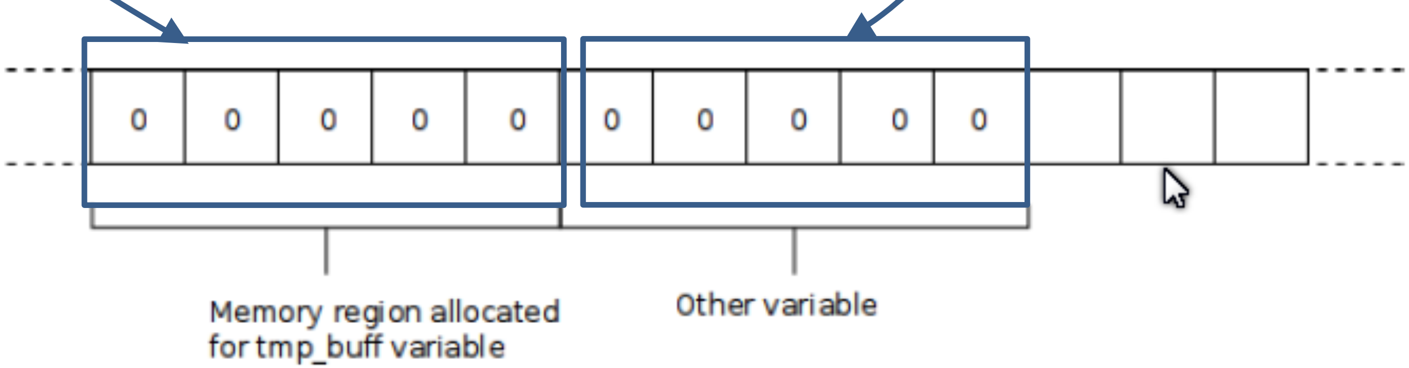
- Memory corruption (safety)
  - Buffer Overflow
  - Integer Overflow
  - Stack Overflow
  - Use after free/ Double free
  - Null pointer dereference
- Exploiting memory corruption to control program
  - Format string attacks
  - Stack buffer overflow (or stack-based buffer overflow)
  - Heap buffer overflow (or heap-based buffer overflow)
  - Overflowing on non “control-data” memory
- Resulting attacks
  - Code injection on the stack (e.g., shellcodes)
  - Code reuse techniques (Return-to-libc, Return-Oriented Programming, Jump-Oriented Programming)

Buffer Overflows

- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the adjacent memory locations
- Originates from mistakes done while writing code
  - unfamiliarity with language
  - Boundary or arithmetic errors
- Mostly C / C++ programs
- Addressed by languages with automatic memory management: dynamic bounds checks (e.g., Java) or automatic resizing of buffers (e.g., Perl)
  - Still native libraries that are written in C (e.g., JNI)

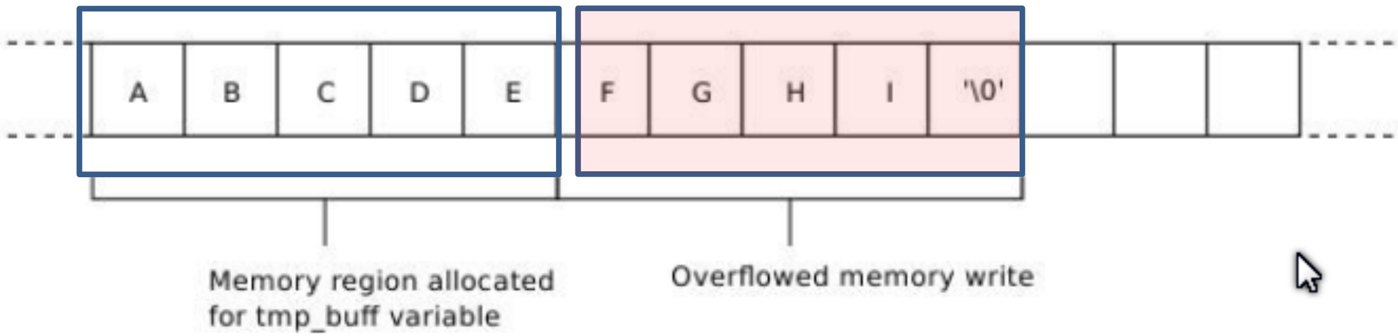
Buffer Overflows: Basics

```
char src[]="ABCDEFGHI";
char tmp_buff[5];
int password_checked;

strcpy(tmp_buff, src);
```



Buffer Overflows: After strcpy()

```
char src[]="ABCDEFGHI";
char tmp_buff[5];
int password_checked;

strcpy(tmp_buff, src);
```



Buffer Overflows: usual suspects

- String manipulation functions that don't properly check string length
  - gets(), strcpy(), …
- Copy with incorrect parameters
  - memcpy() …
- Incorrect computation of required memory length
  - zero-sized malloc()s …

Preventing Buffer Overflows

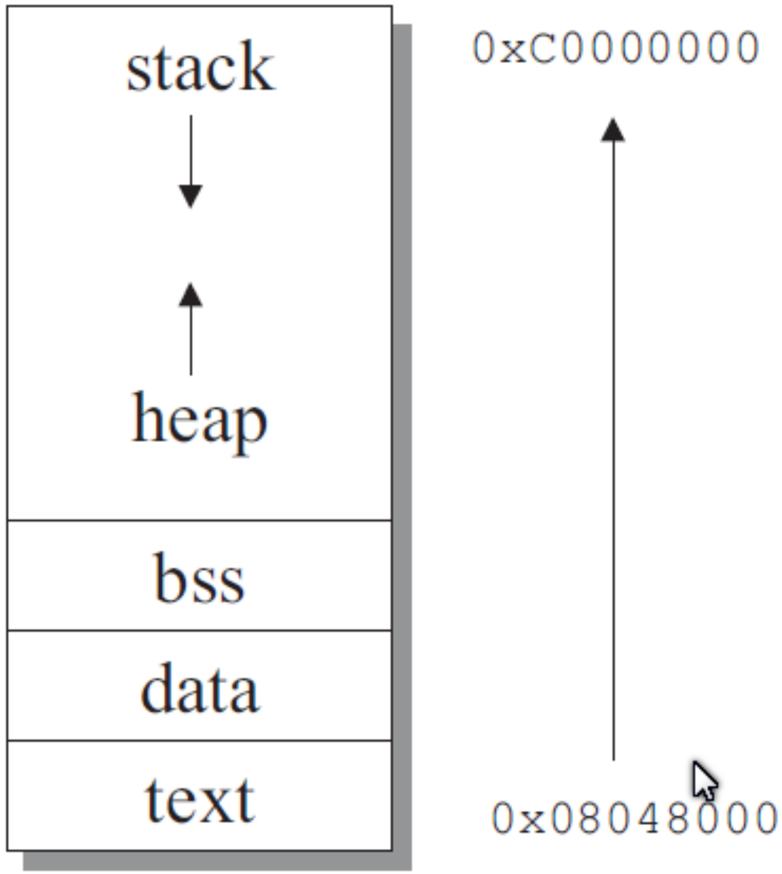
- Safe usage of copy primitives using libc methods
  - For instance, limit amount of data to copy
  - strncpy(tmp_buff, src, sizeof(tmp_buf))
- Compiler extension if array length can be computed
- Annotations (e.g. Deputy, CCured…)

Integer overflows

- Unsigned ints
  - 32 bits: range from 0 to 2^32-1 (4,294,967,295)
  - Overflow: 4,294,967,295 + 1 = 0
- Signed ints
  - 32 bits: range from − (2^31) to 2^31 − 1 (2,147,483,647)
  - Overflow: 2,147,483,647+2 = -2,147,483,648
- Some languages (Java, Ada) throw exceptions, many don't
- Attacker can supply large values used in :
  - Computation of the size of a buffer allocation (malloc)
  - Array access (in particular the bound checks)
- Solution: always check overflow before critical operation!

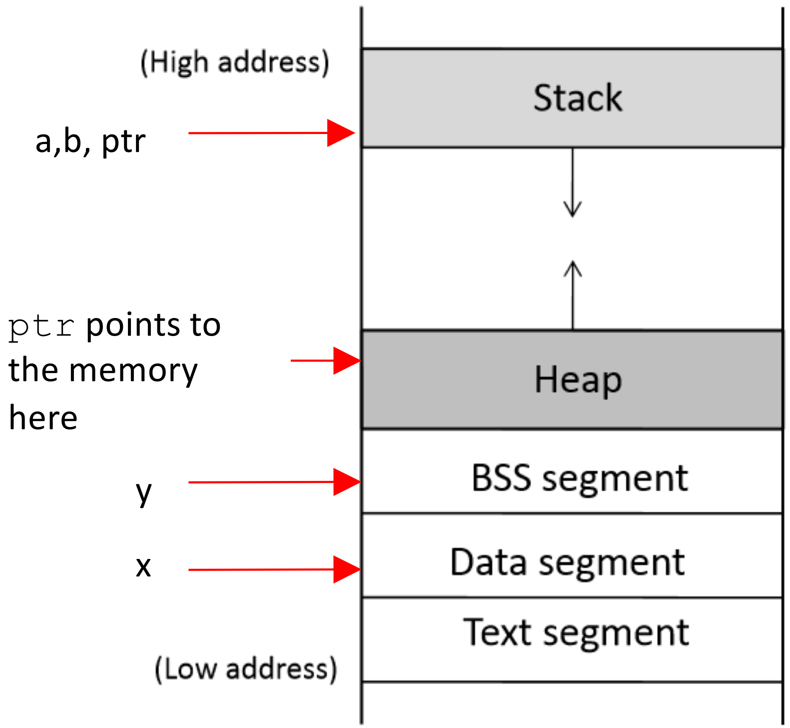
More about Memory Layout

- Text
  - Also called code segment
- Data
  - Global initialized data
- BSS
  - Global uninitialized data
- Stack
  - Local variables
  - Also used to store function environments and parameters during calls (stack frames)
  - LIFO
  - Multi-threading : multiple stacks
- Heap
  - Dynamically allocated variables
  - Reserved through calloc() and malloc()



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;
}
```



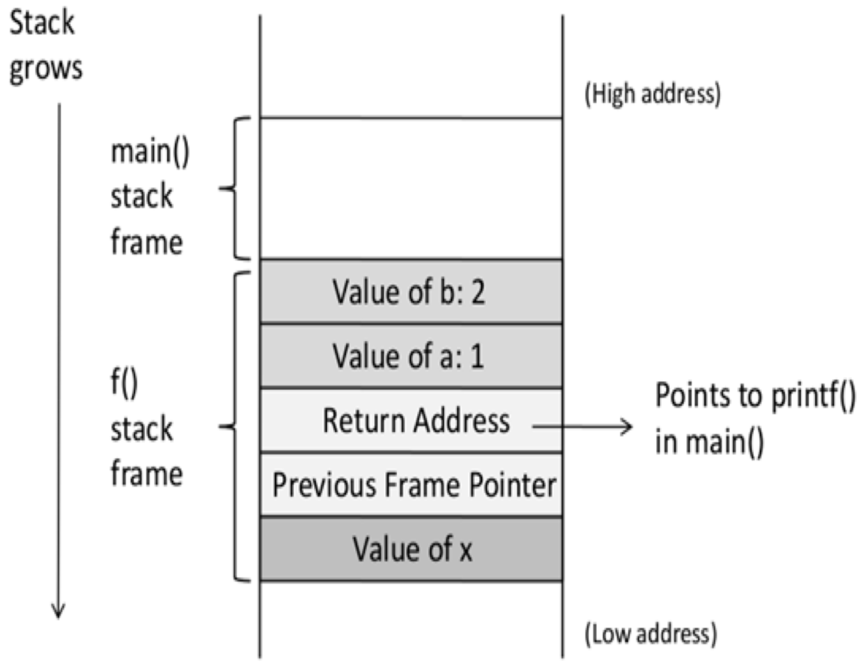
Order of the function arguments in stack

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

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

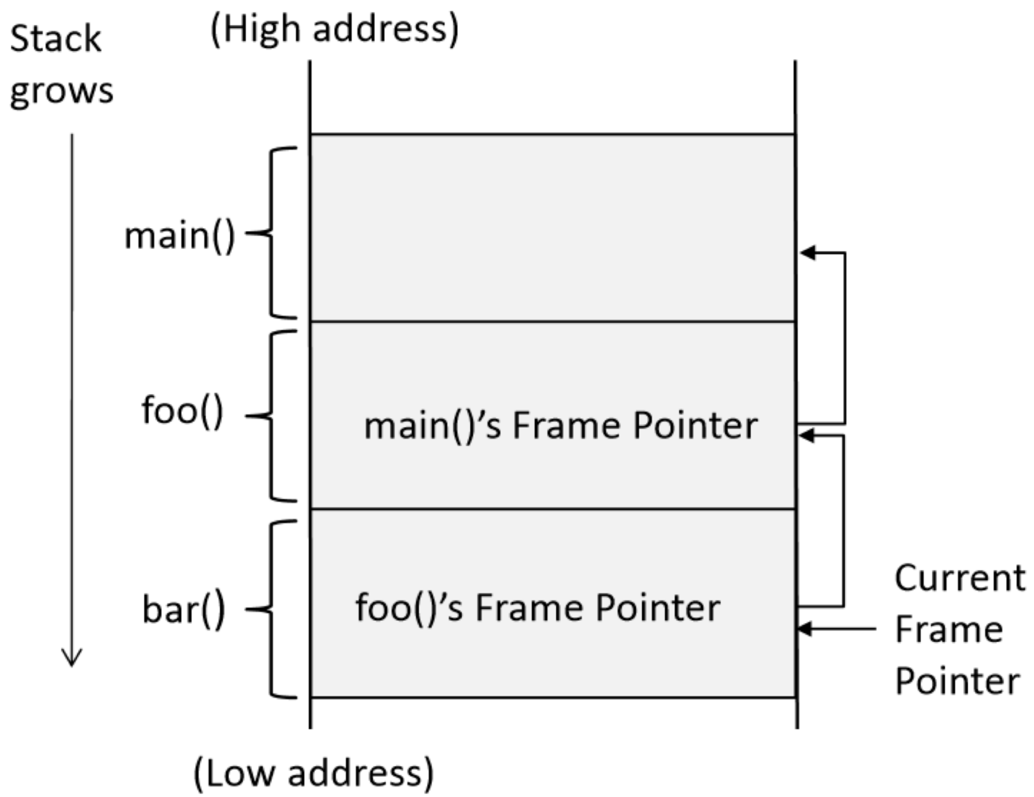
```
movl    12(%ebp), %eax     ; b is stored in %ebp + 12
movl    8(%ebp), %edx      ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)     ; x is stored in %ebp - 8
```

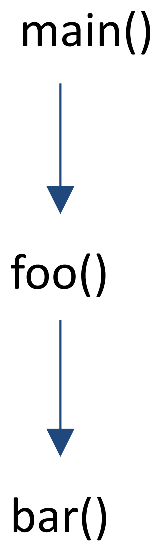
Function Call Stack

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



Stack Layout for Function Call Chain





Stack Overflow

- Stack Overflows can be caused by:
  - Recursive calls
  - Reentrant interrupts
  - Allocations on the stack
    - Large
    - Controlled by the attacker
    - Alloca(), char array[function_parameter]

Stack Overflow

- This should never happen ?
- Detection : guard page (Unix)
  - Problem when allocation is too large
    - the stack pointer can "jump over" the guard page
- Prevention: Not so easy
  - Abstract interpretation [*Regehr05*]
    - Works as long as control flow can be determined statically
  - Forbid / bound recursion
  - Forbid / limit allocation on the stack
- Problem with micro-controllers
  - No MMU (Memory Management Unit)

Stack Buffer Overflow / Overrun

- A special case of buffer overflow
  - Exploitation of existing vulnerability in order to modify control flow
- The overflowed buffer is allocated on the stack, typically corrupts return address
- First wide-scale exploitation: Morris' worm (1989) in Unix's fingerd

Vulnerable Program

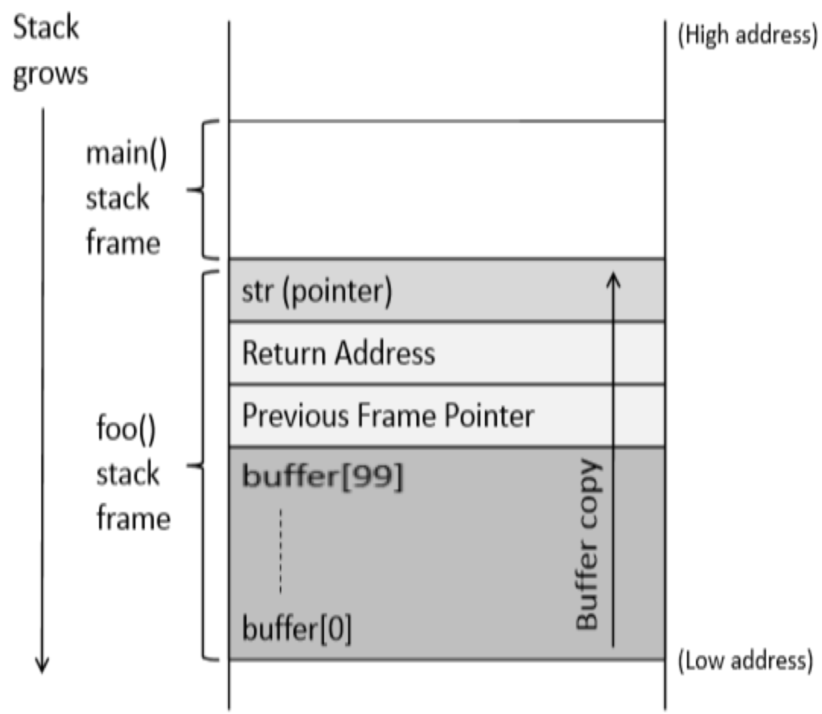
```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    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

```c
/* 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;
}
```

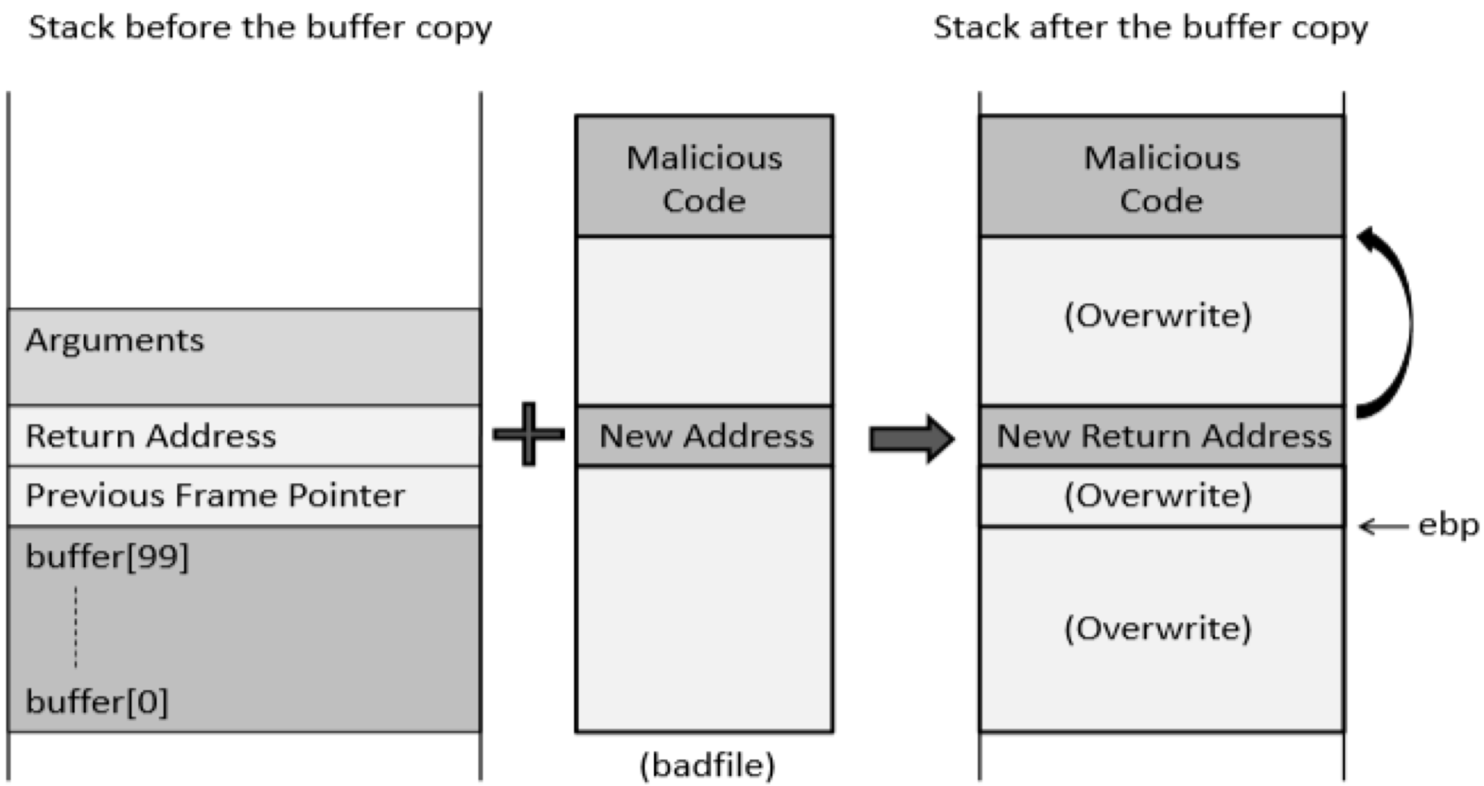


Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

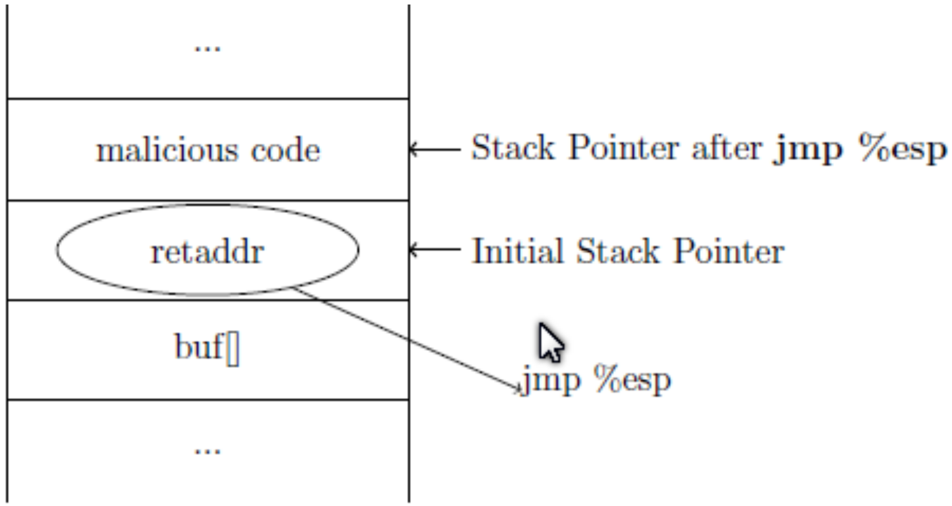
- Invalid instruction
- Non-existing address
- Access violation
- Attacker’s code → Malicious code to gain access

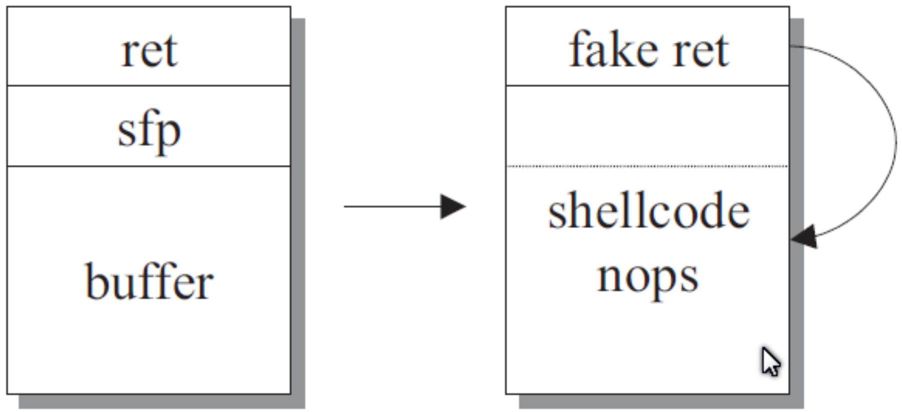
How to Run Malicious Code



More on Modifying return address

- Not always easy to know the exact address
- Trampolines :
  - jmp %esp at a fixed address
- NOP sled
  - A long sequence of NOP
  - Followed by the shellcode
  - Jumping anywhere in there leads to the shellcode





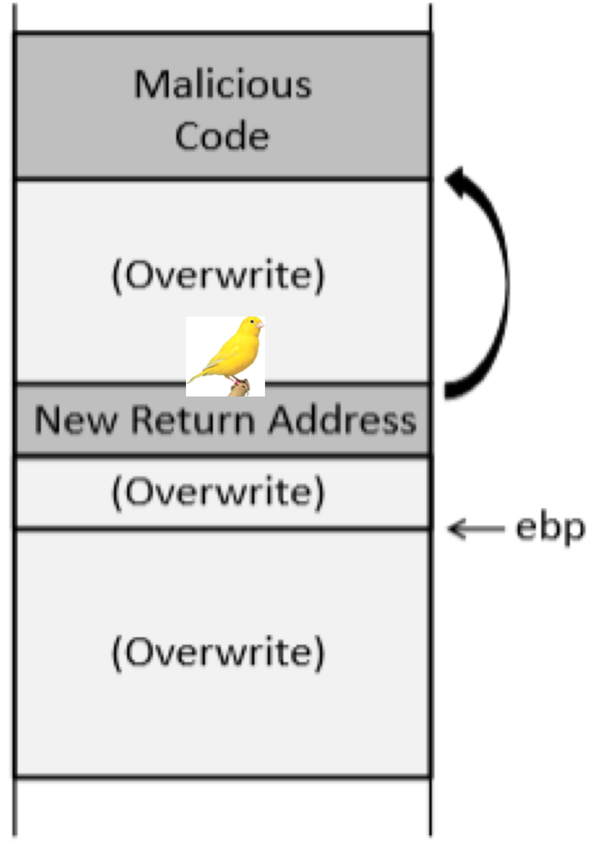
Exploiting Stack Buffer Overflows

- Corrupting the Control Flow
  - Return address in a stack frame
  - Function pointer
  - Exception handler
  - Global Offset Table (GOT)
  - C++ objects

- Corrupting the Program State (other stack frames)
  - Variable that stores authentication status
  - Modify stored password
  - Pointer value

Protections: Canaries

- Objective : Detect unexpected modifications of values on the stack (e.g., return address)
- Inserting a known value on the stack
  - the “canary”
- Compiler extension - inserts code that:
  - Adds a random value after the return address
  - Checks canary value before using return address
- Limitations
  - Canaries can be guessed or obtained through memory leaks
  - Canary copy needs to be stored in secure place that might also be corrupted



Protections: Non Executable Memories (Writable XOR Executable / NX)

- Executable and writable memories regions
  - Allows to directly write instructions (in a buffer) and execute them, e.g., from the stack.
  - Makes stack-based buffer overflows easy to exploit

- W XOR X (A.k.a NX / XN / DEP / PAX)
  - OS splits memory regions – no region is both writable and executable
- Most systems now support a form of W XOR X:
  - Was complex on x86 (segmentation)
    - HW support for it since ~2004
- Many names for the same feature :
  - AMD : Enhanced Virus Protection,
  - ARM : XN eXecute Never
- Defeated by Return-to-libc attacks / Return Oriented programming

Protections: ASLR (Address Space Layout Randomization)

- Randomizes base addresses of memory segments
  - Addresses change at every execution
- Randomize start or base address of:
  - Program code
  - Libraries code
  - Heap/Stack/Data regions
- Objectives:
  - Difficult to guess the stack address in the memory.
  - Difficult to guess %ebp address and address of the malicious code
- Many programs have problems with that
  - Sometimes rewriting part of them is necessary, notably when they contain specially crafted assembly code
  - Code needs to be position independent for program randomization
- Limitations
  - Memory leaks used to learn memory layout
  - Address space / system limitations (e.g., page boundaries) may allow brute-force probes

Address Space Layout Randomization

```
#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);
}
```

Address Space Layout Randomization : Working

```
$ 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

3

```
$ 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
```