Buffer Overflow

Kameswari Chebrolu

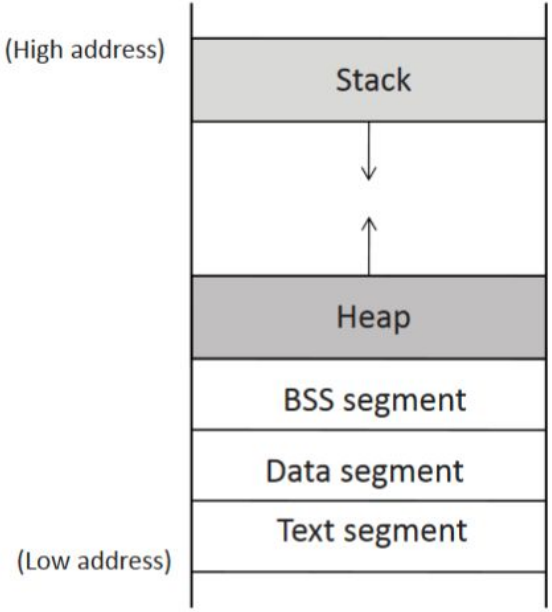
Buffer Overflow Attack

- Basis of many prior famous attacks
  - Morris worm in 1988, Code Red worm in 2001, SQL Slammer in 2003, Stagefright attack against Android phones in 2015

Process Address Space

Virtual memory

- Text segment: stores the executable code; usually read-only
- Data segment: stores static/global variables that are initialized
- BSS segment: stores uninitialized static/global variables
  - Filled with zeros by the operating system
- Heap: provides space for dynamic memory allocation
  - Managed by malloc, calloc, realloc, free, etc.
- Stack: storing data related to function calls
  - Local variable, return address, arguments etc

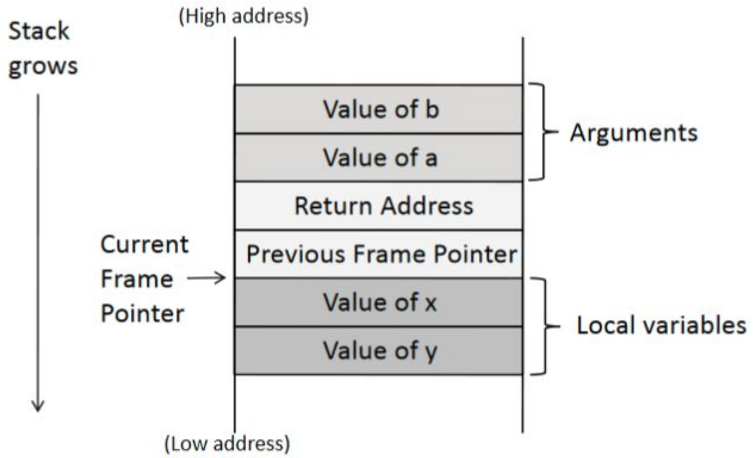


```c
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // 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;
}
```

- variable `x` is a global variable → Data segment
- Variable `y` is an uninitialized static → BSS segment
- Variables `a` and `b` are local variables → program's stack
- Variable `ptr` a local variable → stack
- `ptr` is a pointer, pointing to a block of memory, which is dynamically allocated using `malloc()`; therefore, when the values `5` and `6` are assigned to `ptr[1]` and `ptr[2]`, they are stored in the heap segment

Stack Memory Layout

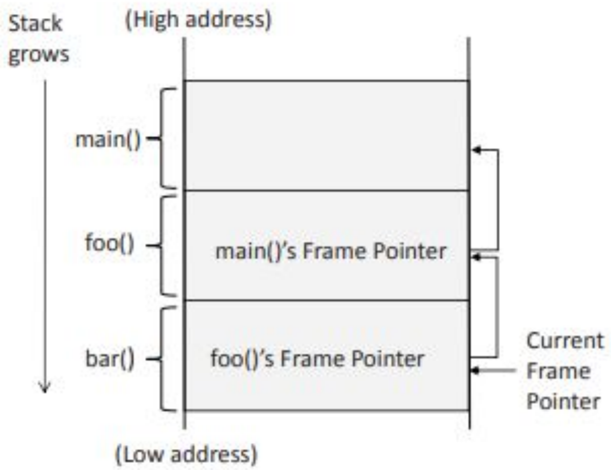
```
void func(int a, int b)
{
    int x, y;

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



- Stack frame: memory allocated to fu
- Return address: address of next instruction in the program after the function
- Frame Pointer: see next slide

Frame Pointer

- Inside func(), we need to access the arguments and local variables
- How? need to know their memory addresses
  - Addresses cannot be determined during compilation time
- A special register is introduced in CPU called frame pointer
- Points to a fixed location in the stack frame
  - See Current frame pointer in the figure
  - Address of each argument and local variable can be calculated using this register and an offset



Previous frame pointer?

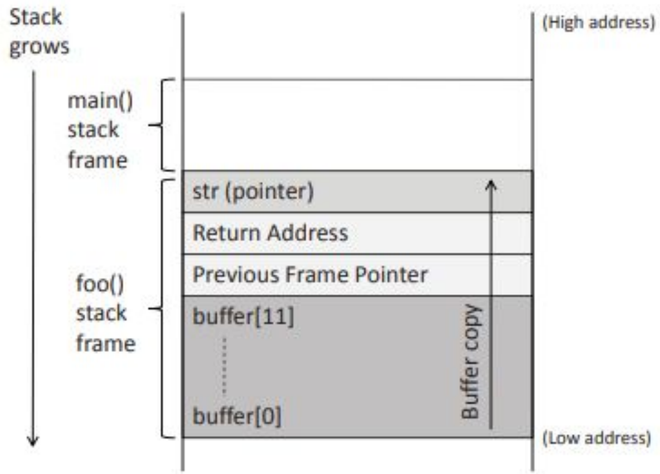
- We often call a function from inside a function
  - See fig: Three stack frames are on the stack
- Frame pointer register always points to the stack frame of the current function
- Once we return from bar(), where is function foo()'s stack frame?
- Before entering the callee function, the caller's frame pointer value is stored in the "previous frame pointer" field
  - When the callee returns, the value in this field will be used to set the frame pointer register

Buffer Overflow Vulnerability

- Arises due to improper memory copying from src to dest
- Before copying, a program needs to allocate adequate memory space for the destination
  - Programming error: fail to allocate sufficient memory for the destination $\longrightarrow$ more data will be copied
  - Results in a buffer overflow
  - Programming languages, such as Java, automatically detect the problem but other languages such as C and C++ don't
- What is the big deal? Program will crash!
- Attacker can gain a complete control of a program, rather than simply crashing it
  - If program runs with privileges (e.g. root), attackers will be able to gain those privileges

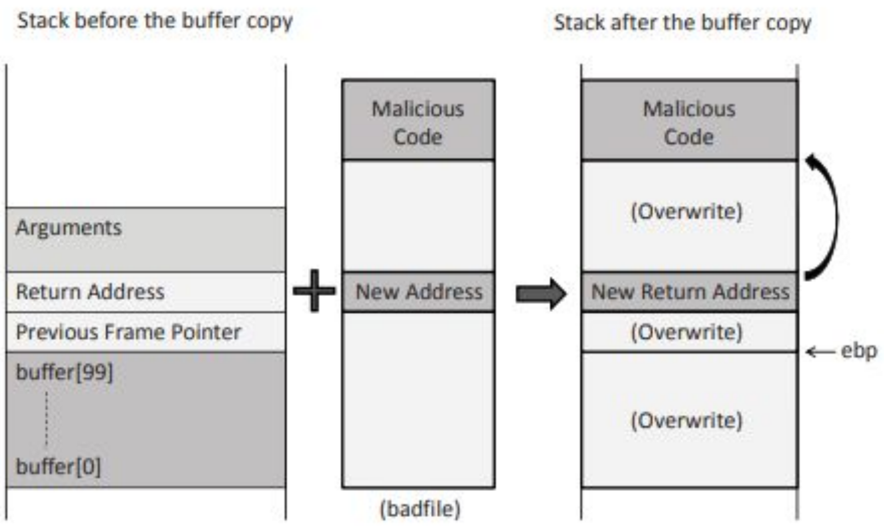
```c
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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

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

    return 1;
}

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;
}
```

Normal Consequences



- New address may not be mapped to any physical address → program will crash
- Address mapped to protected physical address (kernel) → program will crash.
- Address mapped to a physical address which holds data (not instructions) → program will crash.
- Address contains a valid machine instruction but logic altered!

Attack Goals

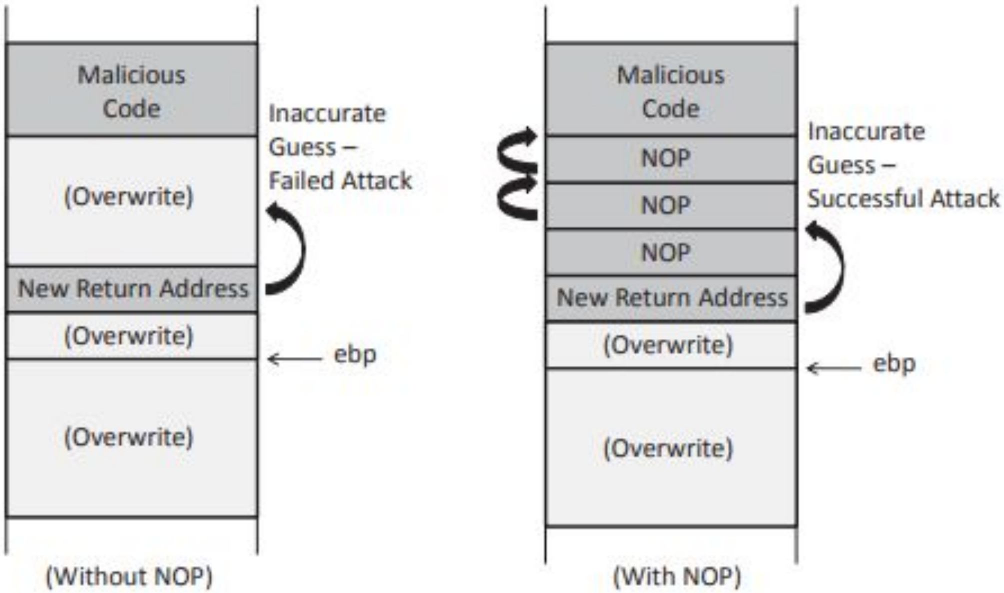


Lot more complicated to realize in practice!

- How to finding the address of the Injected Code?
- How to construct the input file?
  - Malicious code should ideally be a shell code. Why?

Address of Injected Code

- To be able to jump to malicious code, we need to know the memory address of the malicious code
  - We know offset of the malicious code but we don’t know the address of the function foo’s stack frame
  - Target program unlikely to print out the value of its frame pointer or the address of any variable inside the frame
- Need to guess?
  - 32 bit machine → 2^32 guesses?
  - No. Space is much smaller
- Before countermeasures, most OS placed the stack at a fixed starting address (virtual)
- Most programs do not have a deep stacks
- Can improve guess via NOP Sledding!


Malicious Code
(Overwrite)
New Return Address
(Overwrite)
(Overwrite)
Inaccurate Guess – Failed Attack
ebp
(Without NOP)
Malicious Code
NOP
NOP
NOP
New Return Address
(Overwrite)
(Overwrite)
Inaccurate Guess – Successful Attack
ebp
(With NOP)

Malicious code

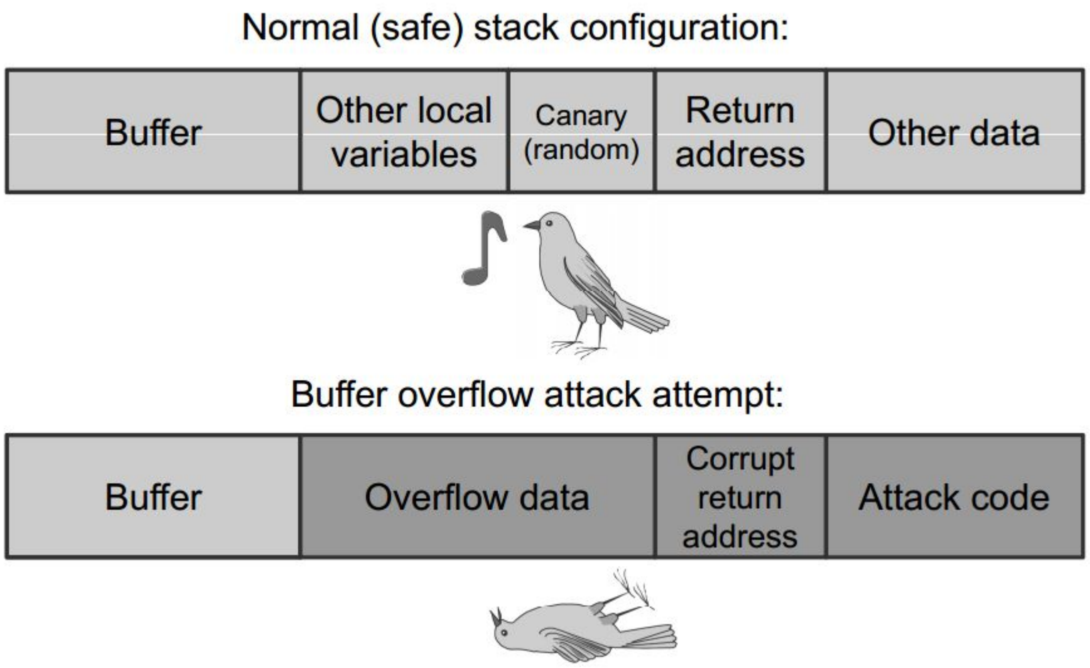
- Often shell code to launch more commands
- Written in Assembly language (op codes)
- Challenge: No 0x00 (buff will terminate)
- Solution: Encode shell code
  - Needs to decode itself first

Defenses

- Safer Functions: Memory copy functions like strcpy, sprintf, strcat, and gets; use safer versions strncpy, snprintf, strncat, fgets, respectively.
  - Safer functions are only relatively safer
  - If a developer specifies length larger than actual size of buffer, there will be buffer overflow vulnerability
- Safer Dynamic Link Library: Requires changes to the program
  - Can build a safer library and get a program to dynamically link to the functions in this library
- Program Static Analyzer: Warns developers of patterns in code that may potentially lead to buffer overflow vulnerabilities

- Use languages such as Java, Python that provide better automatic boundary checking
- Compiler: Allows compilers to insert instructions into the binary that can verify the integrity of a stack
  - Stackshield: store return address before function call and check after call is it is changed!
  - Stackguard: Uses canary, see next slide
- Operating System: Address Space Layout Randomization or ASLR
  - Targets the fact that attackers must guess the address of the injected code
  - Randomizes the layout of the program memory
  - Most OS implement this feature
- Hardware Architecture: Avoid execution of code on the stack
  - Modern CPUs support a feature called NX bit (No-eXecute)
  - Separates code from data
  - If stack is marked as non-executable, OS will not execute code on it
    - Can be defeated using a different technique called return-to-libc attack

Canary



Normal (safe) stack configuration:
Buffer
Other local variables
Canary (random)
Return address
Other data
Buffer overflow attack attempt:
Buffer
Overflow data
Corrupt return address
Attack code

Reference

- See attached notes
  - This is a sample chapter, with incomplete portions
  - But the crux of the idea is there