Lab 2: Buffer Overflows

Fengwei Zhang

Buffer Overflows

- One of the most common vulnerabilities in software

- Programming languages commonly associated with buffer overflows including C and C++

- Operating systems including Windows, Linux and Mac OS X are written in C or C++

How It Works

- Applications define buffers in the memory
  - *Unsigned char c [10]*

- Applications use adjacent memory to store variables, arguments, and return address of a function.

- Buffer Overflows occurs when data written to a buffer exceeds its size.

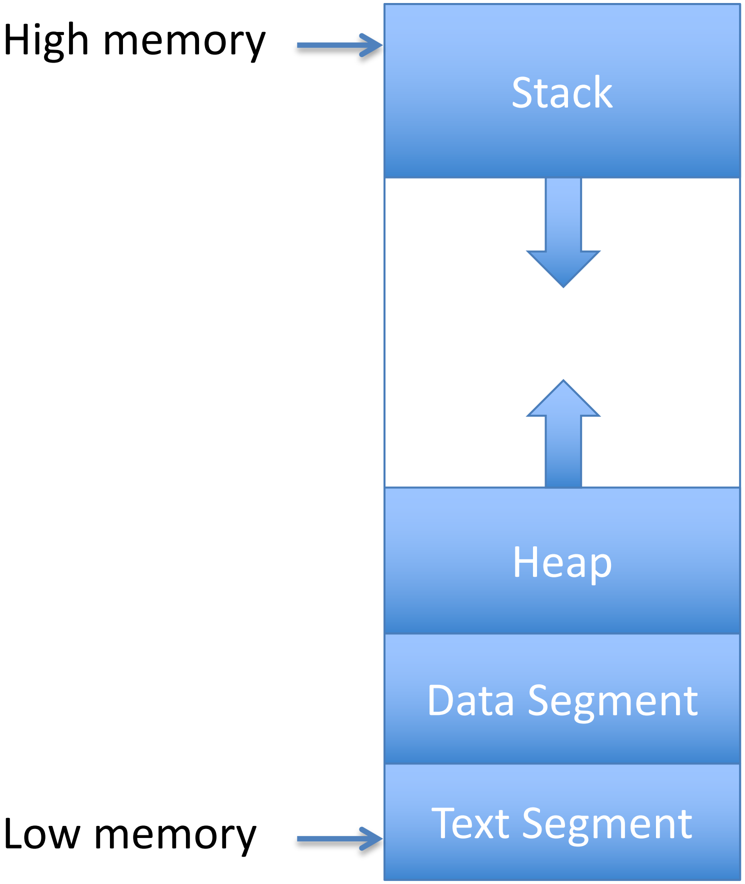
Overflowing A Buffer

- Defining a buffer in C
  - char buf [10];
- Overflowing the buffer
  - Char buf [10] = ‘x’;
  - strcpy(buf, “AAAAAAAAAAAAAAAAAAAAAAA”)

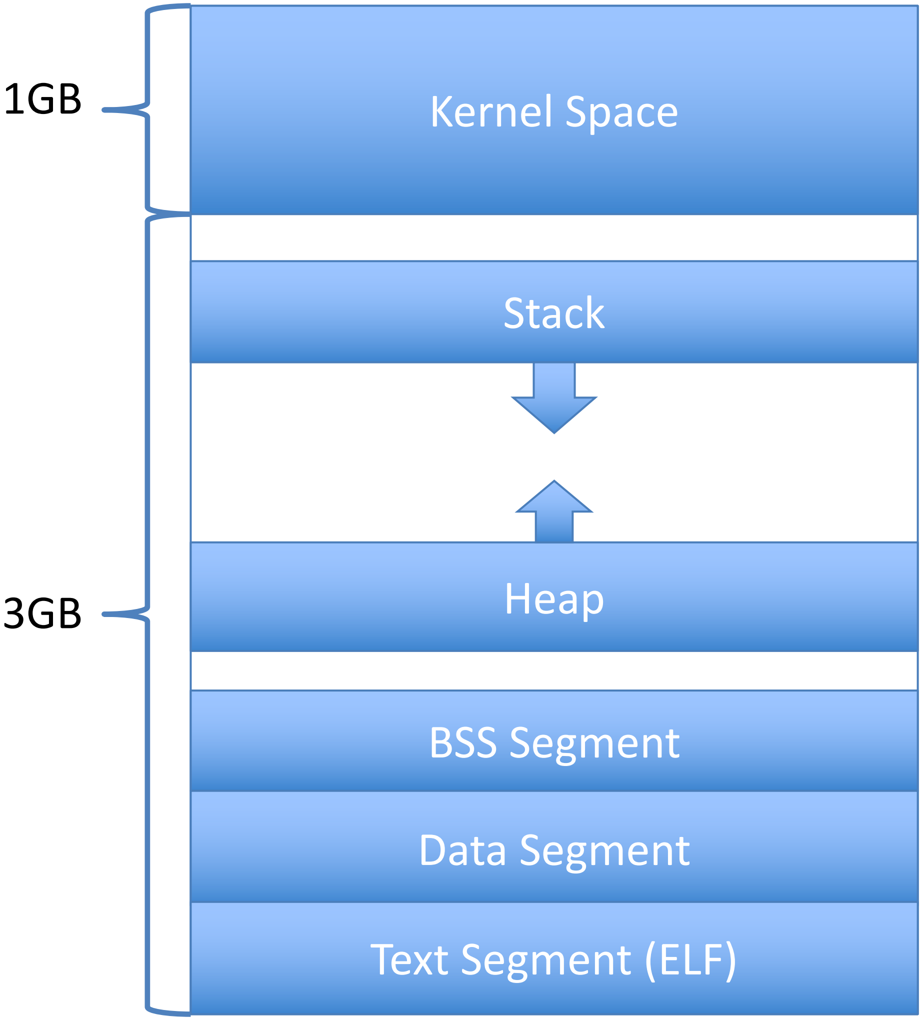
Why We Care

- Because adjacent memory stores program variables, parameters, and arguments
- Attackers can change these values through overflowing a buffer
- Attackers can gain control over the program flow to execute arbitrary code

Process Memory Layout



Memory Layout for 32-bit Linux



Local variable: int a

Function malloc()

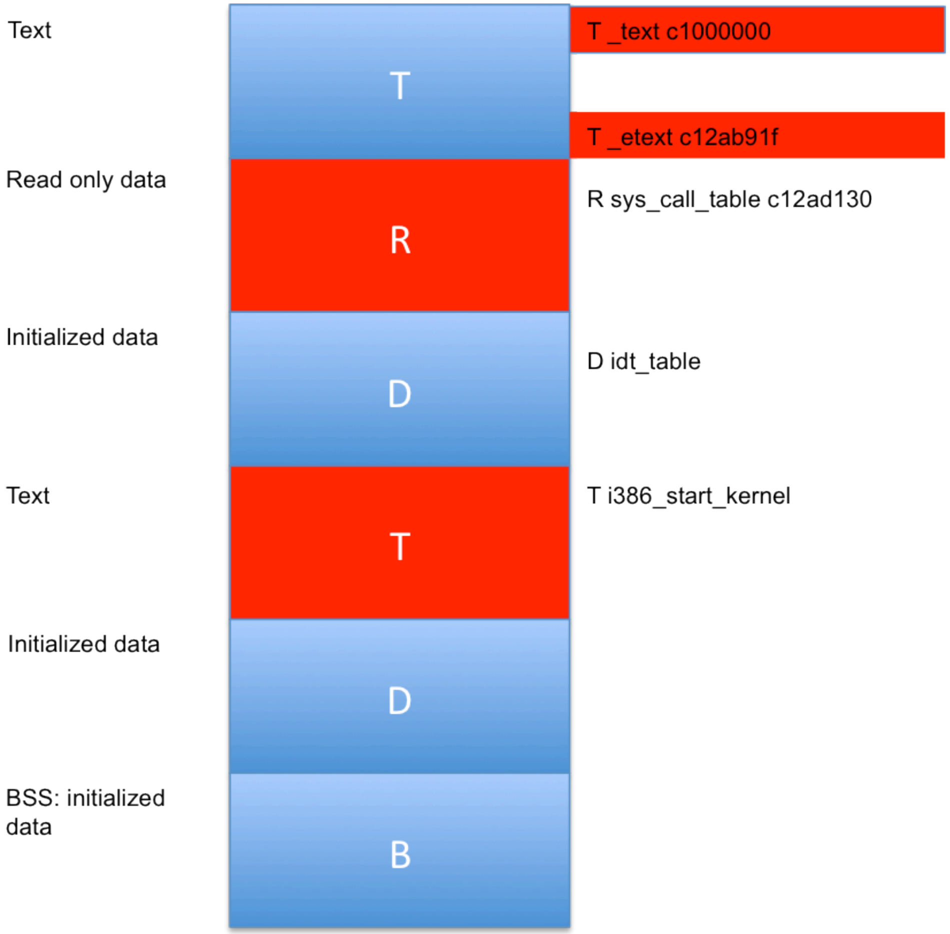
Uninitialized static variables: static char *u

static char *s = “Hello world”

Binary of the program

Virtual Memory Layout

System.map-2.6.32



Text

T

T _text c1000000

T _etext c12ab91f

Read only data

R

R sys_call_table c12ad130

Initialized data

D

D idt_table

Text

T

T i386_start_kernel

Initialized data

D

BSS: initialized data

B

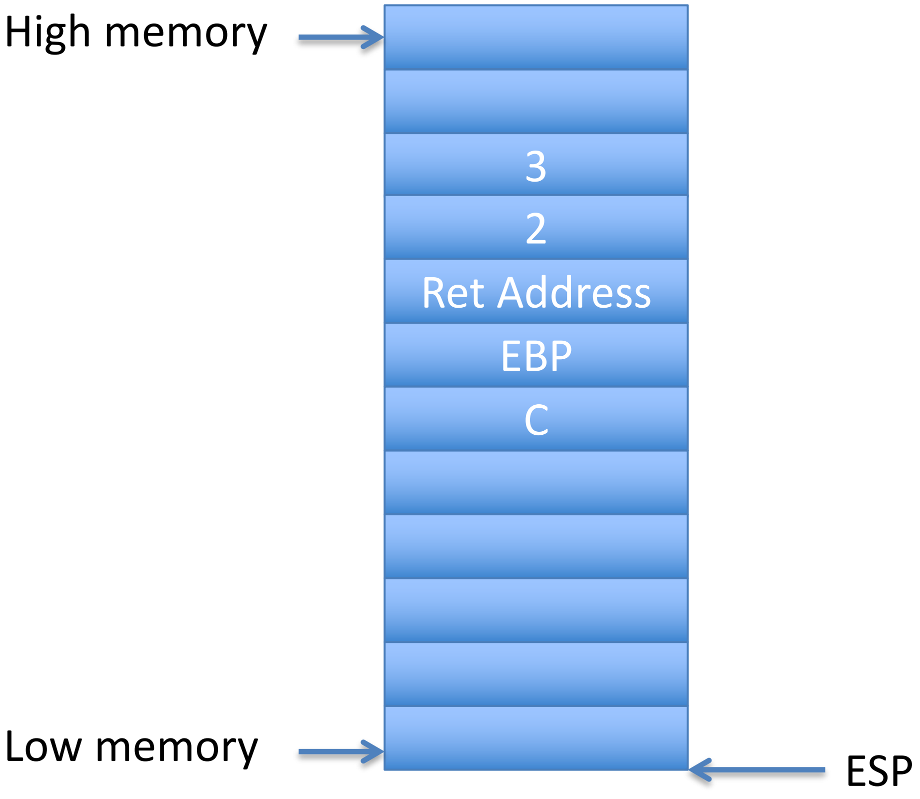
Stack Frame

- The stack contains activation frames including local variables, function parameters, and return address
- Starting at the highest memory address and growing downwards
- Last in first out

A Simple Program

Add (2,3)

```
int add (int a, int b)
{
        int c;
        c = 1+b;
        return c;
}
```



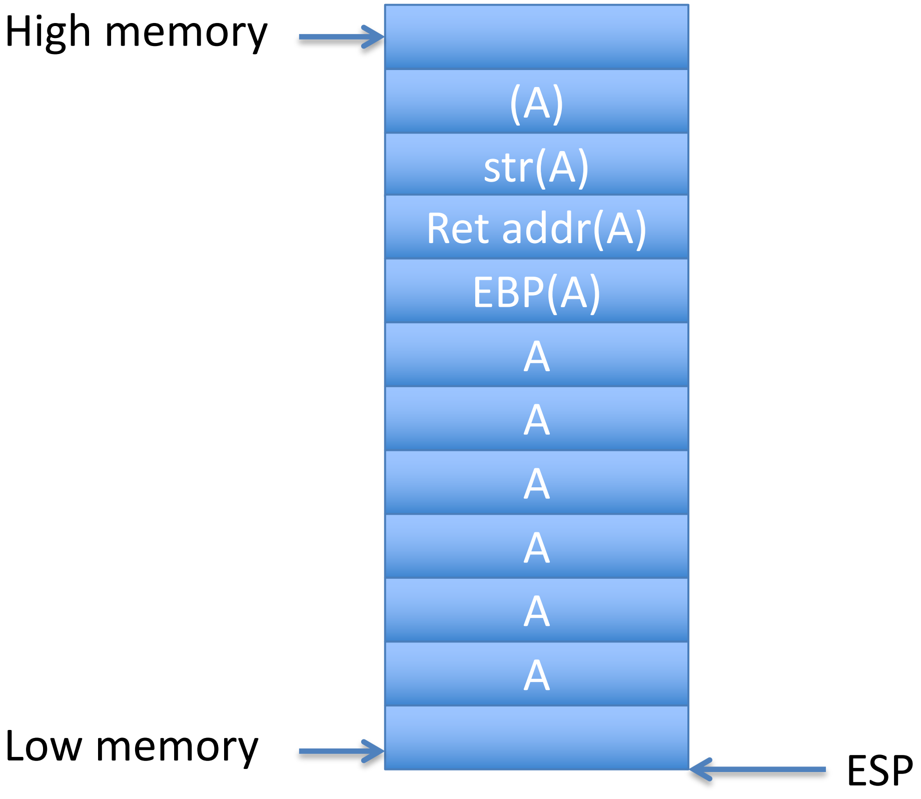
Another Program

```
int func (char * str)
{
      char mybuff[512];
      strcpy(myBuff, str);
      return 1;
}

int main (int argc, char ** argv)
{
      func (argv[1]);
      return 1;
}
```

Draw the Stack Frame!

Overflowing “myBuff”



Buffer Overflow Defenses

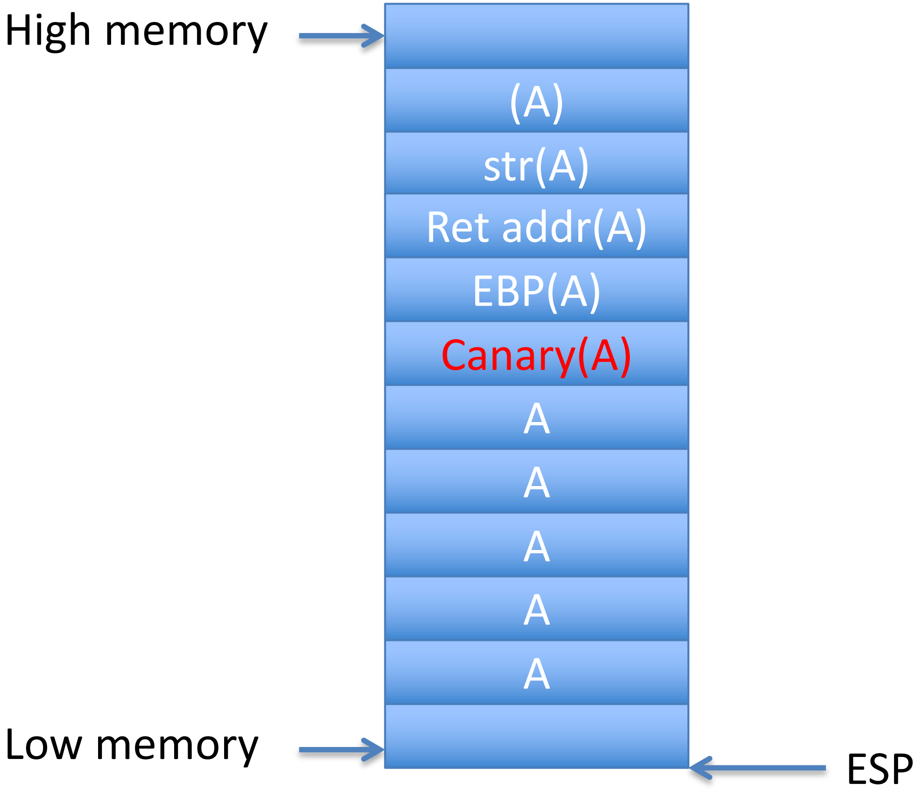
- The attack described is a classical stack smashing attack which execute the code on the stack
- It does not work today
  - NX – non-executable stack. Most compilers now default to a non-executable stack. Meaning a segmentation fault occurs if running code from the stack (i.e., Data Execution Prevention - DEP)
    - Disable it with –zexecstack option
    - Check it with readelf –e <PROGRAM> | grep STACK
  - StackGuard: Canaries
    - Disable it with –fno-stack-protector option
    - Enable it with –fstack-protector option

Stack Canaries

- Stack smashing attacks do two things
  - Overwrite the return address
  - Wait for algorithm to complete and call RET

- Stack Canaries: Stack Smashing Protector (SSP)
  - Placing a integer value to stack just before the return address
  - To overwrite the return address, the canary value would also be modified
  - Checking this value before the function returns

Stack Canaries (cont’d)



Bypassing NX and Canaries

- NX - non-executable stack
  - Executing code in the heap
  - Data Execution Prevention (DEP)
  - Return Oriented Programming (ROP)

- Stack Canaries
  - Overwriting the Canary with the same value
  - Brute force attack (e.g., DynaGuard in ACSAC'15)

Reminders