CIS 551 / TCOM 401

Computer and Network Security

Spring 2009

Lecture 2

Buffer Overflow Attacks

- > 50% of security incidents reported at CERT are related to buffer overflow attacks
- Problem is access control but at a very fine level of granularity
- C and C++ programming languages don't do array bounds checks

Buffer overflows in library code

- Basic problem is that the library routines look like this:

```
void strcopy(char *src, char *dst) {
  int i = 0;
  while (src[i] != "\0") {
    dst[i] = src[i];
    i = i + 1;
  }
}
```

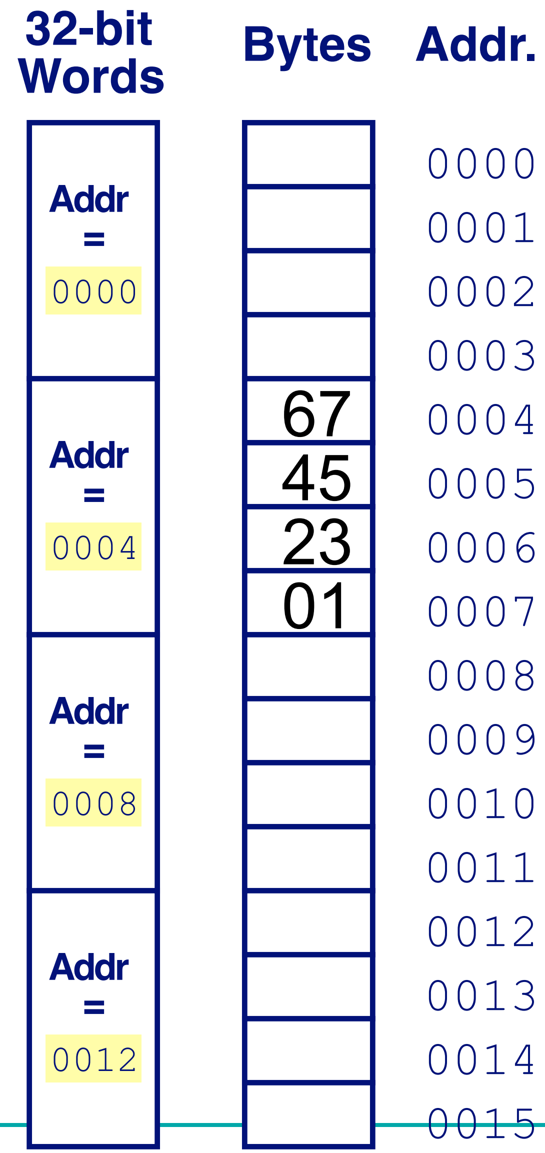
- If the memory allocated to `dst` is smaller than the memory needed to store the contents of `src`, a buffer overflow occurs.

Attack Targets and Locations

- Targets
  - Return address
  - Function pointer
  - Longjmp buffer
  - A flag relative to control flow …
- Locations
  - Stack
  - Heap or Data segment

Memory Organization (mod-l.seas.upenn.edu)

- Linux Kernel 2.6, GCC 4.1.2
- Without stack protection
- Word size = 32 bits (4 bytes)
- Little Endian
  - *(0x0004) = 0x01234567
- Stack grows to smaller addresses



Data Representations

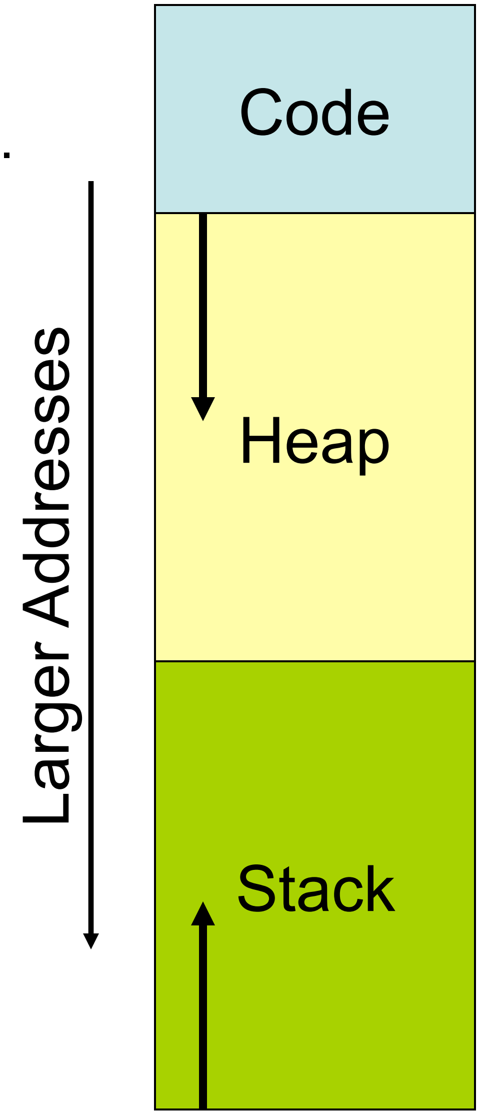
- Sizes of C Objects (in Bytes)
  - C Data Type
    - int 4
    - char 1
    - pointer 4
- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Character “0” has code `0x30`
      - Digit *i* has code `0x30`+*i*
  - String should be null-terminated
    - Final character = 0

```
char str[5] = “1234";
```

Linux	Addr.
31	bf00
32	bf01
33	bf02
34	bf03
00	bf04

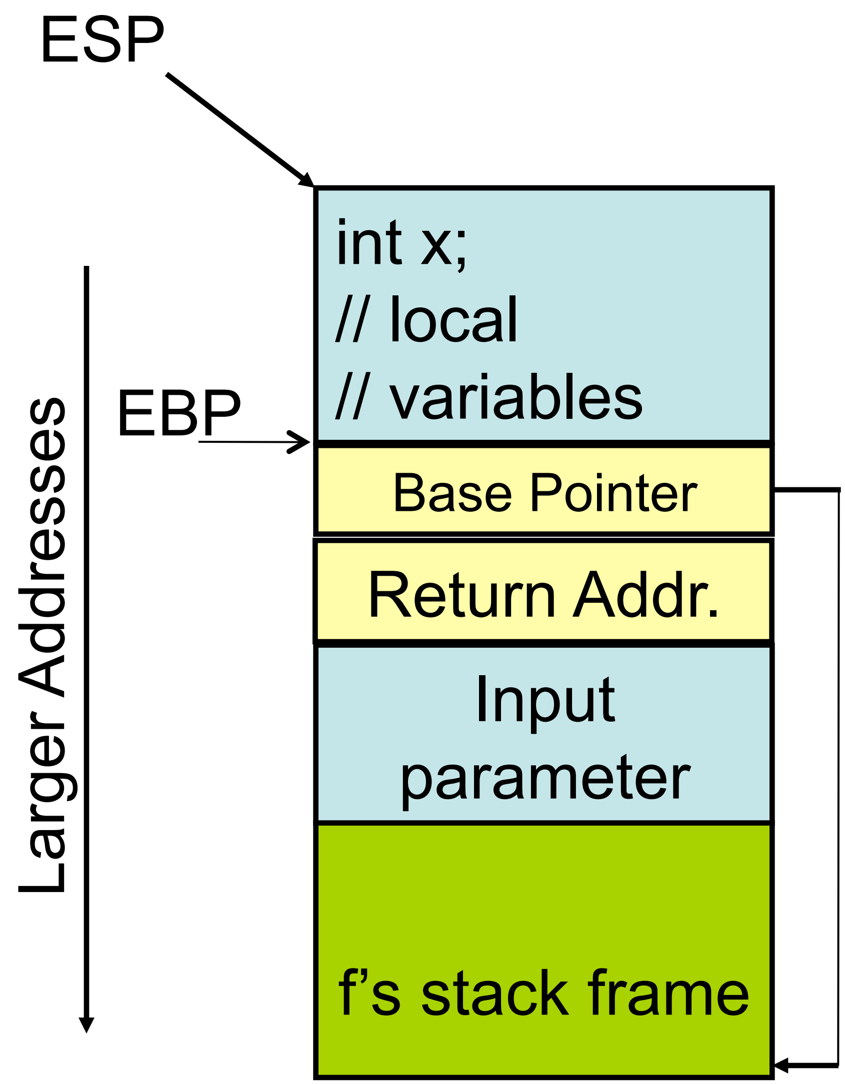
3 parts of C memory model

- The code & data (or "text") segment
  - contains compiled code, constant strings, etc.
- The Heap
  - Stores dynamically allocated objects
  - Allocated via "malloc"
  - Deallocated via "free"
  - C runtime system
- The Stack
  - Stores local variables
  - Stores the return address of a function



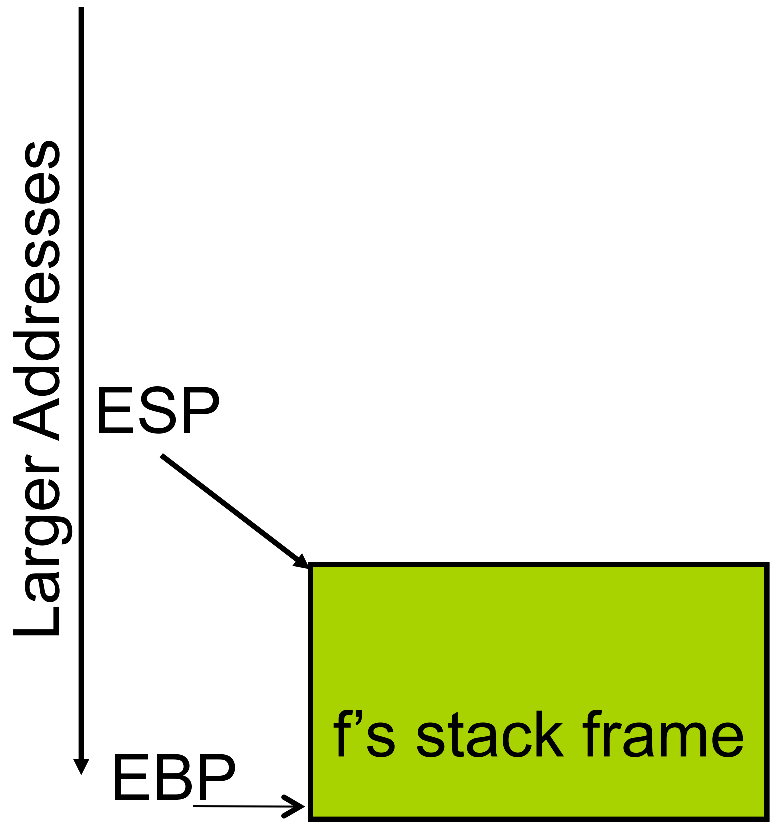
C’s Control Stack

```
f() {
   g(parameter);
}

g(char *args) {
   int x;
   // more local
   // variables
   ...
}
```



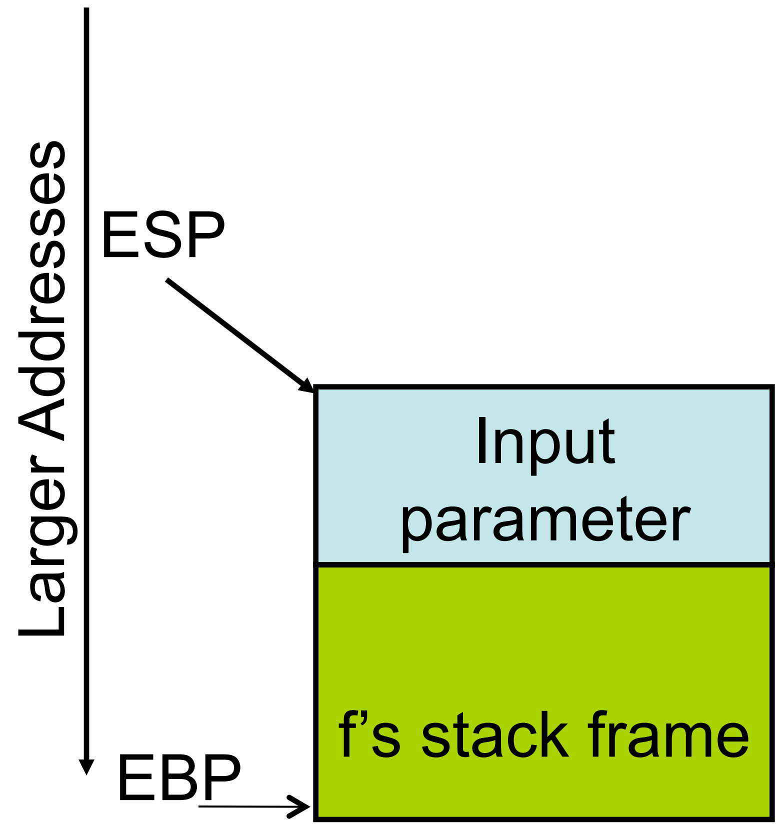
C's Control Stack

```
f() {
  g(parameter);
}

g(char *args) {
  int x;
  // more local
  // variables
  ...
}
```



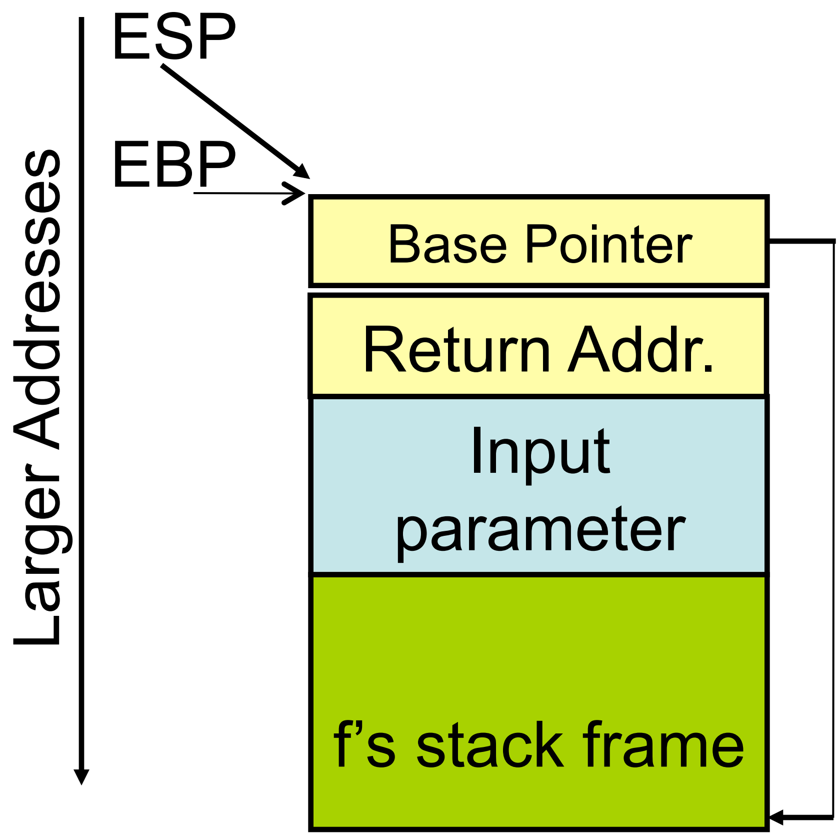
C's Control Stack

```
f() {
  g(parameter);
}

g(char *args) {
  int x;
  // more local
  // variables
  ...
}
```



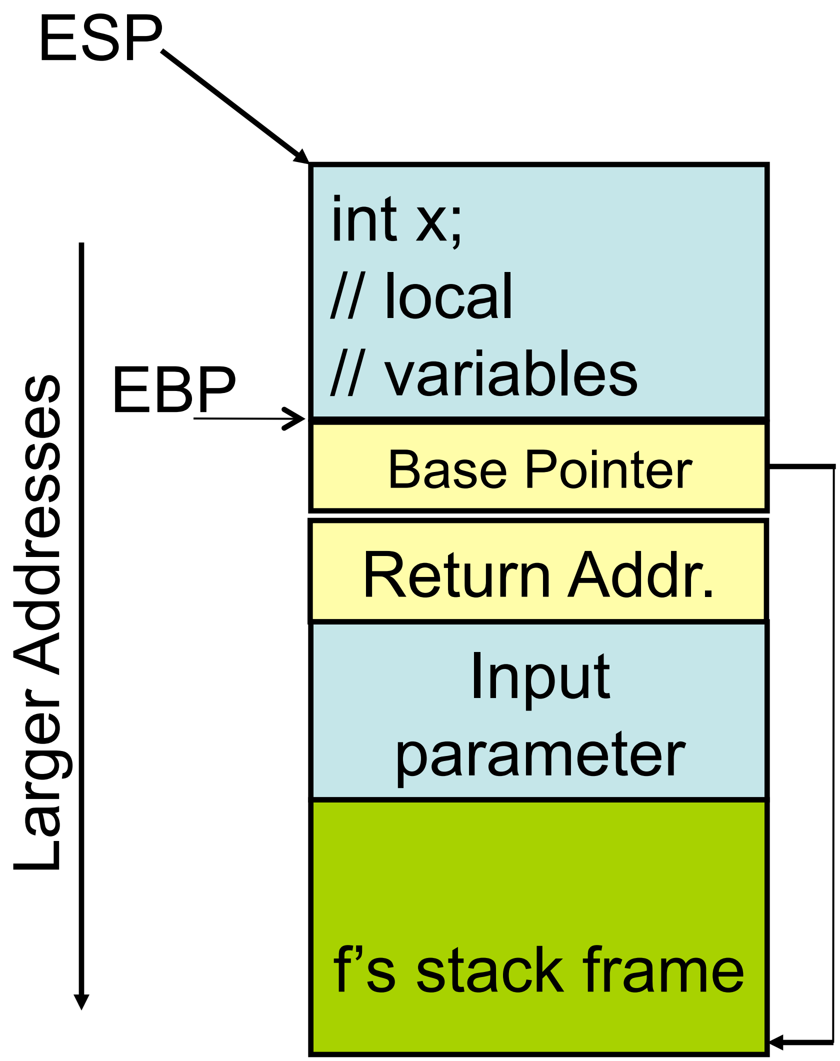
C's Control Stack

```
f() {
   g(parameter);
}

g(char *args) {
   int x;
   // more local
   // variables
   ...
}
```



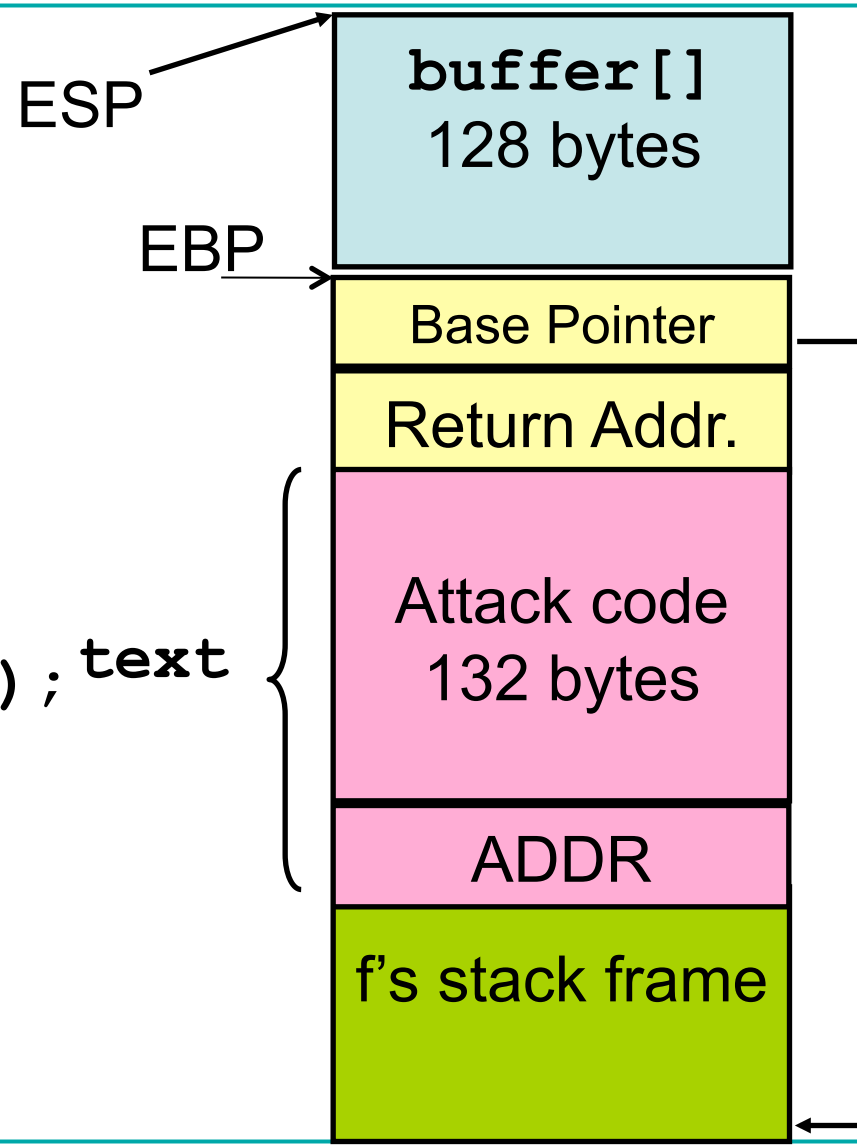
C's Control Stack

```
f() {
  g(parameter);
}

g(char *args) {
  int x;
  // more local
  // variables
  ...
}
```



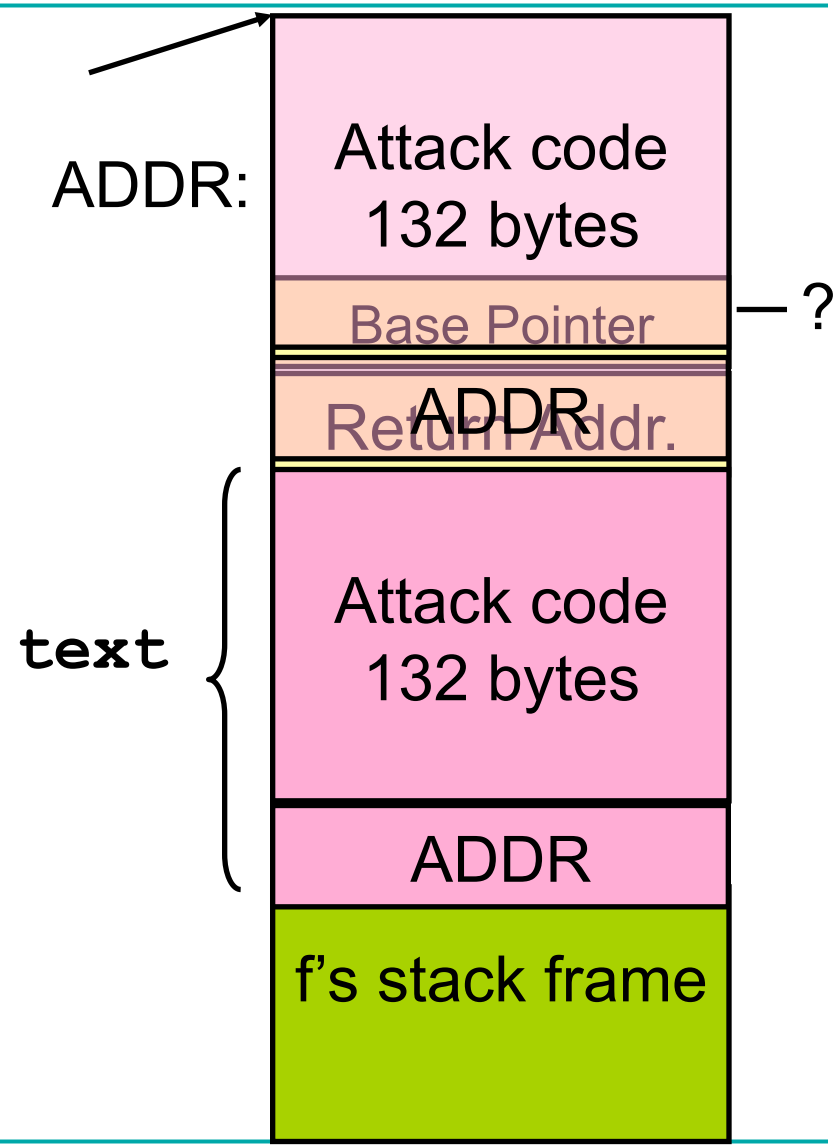
Buffer Overflow Example

```
f() {
   g(parameter);
}

g(char *text) {
   char buffer[128];
   strcpy(buffer, text);
}
```



Buffer Overflow Example

```
f() {
   g(parameter);
}

g(char *text) {
   char buffer[128];
   strcpy(buffer, text);
}
```



Assembly Instructions

- Register
  - %esp : stack pointer
  - %ebp : frame base pointer
  - %eax : function return value
  - …
- Constant Numbers: $0x0, $-16
  - Moving Data: movl src dest
  - Arithmetic/Logical Operation:
  - addl $12, %esp
  - subl , andl
  - Stack Operation: pushl, popl
  - Control Flow: jmp, call, leave, ret…

Details: C calling conventions

```
int function(int a, int b, int c) {
  char buffer1[4];
  int ans = a + b + c;
  char buffer2[10];
  return ans;
}

int main() {
  return function(1,2,3);
}
```

Compile with: gcc -S -o example.s example.c

Resulting Assembly (1)

```
   .file  "example.c"
   .text
.globl function
   ...
.globl main
   .type  main, @function
main:
   pushl  %ebp                           // Set up stack frame
   movl     %esp, %ebp
   subl   $8, %esp                       // Align the stack on 16-byte boundary,
   andl     $-16, %esp                   // reserve some space on the stack
   subl   $16 %esp
   pushl  $3                             // Push arguments onto the stack
   pushl  $2
   pushl  $1
   call     function                     // Push return address, jump to function:
   addl     $12, %esp                    // Pop arguments off the stack
   leave                                 // Tear down stack frame, Undo stack alignment
   ret
```

Resulting Assembly (2)

```
.globl function
    .type       function, @function
function:
    pushl                   %ebp           // Set up stack frame
    movl        %esp,       %ebp
    subl   $32,       %esp                 // Allocate local storage
    movl        12(%ebp), %eax
    addl        8(%ebp),  %eax             // ans = a + b + c
    addl        16(%ebp), %eax
    movl        %eax,     -4(%ebp)
    movl        -4(%ebp),   %eax           // %eax holds the return value
    leave                                  // Tear down stack frame
    ret                                    // Pop return address & jump to it
    .size       function, .-function
```

Smaller Addresses ←

	buf2	buf1	ans	ebp	ret	a	b	c
esp	-18	-8	-4	ebp		8	12	16

Constructing a Payload

- Idea: Overwrite the return address on the stack
  - Value overwritten is an address of some code in the "payload"
  - The processor will jump to the instruction at that location
  - It may be hard to figure out precisely the location in memory

- You can increase the size of the "target" area by padding the code with no-op instructions
- You can increase the chance over overwriting the return address by putting many copies of the target address on the stack

[NOP]…[NOP]{attack code} {attack data}[ADDR]…[ADDR]

More About Payloads

- How do you construct the attack code to put in the payload?
  – You use a compiler!
  – Gcc + gdb + options to spit out assembly (hex encoded)
- What about the padding?
  – NOP on the x86 has the machine code 0x90
- How do you guess the ADDR to put in the payload?
  – Some guesswork here
  – Figure out where the first stack frame lives: OS & hardware platform dependent, but easy to figure out
  – Look at the program -- try to guess the stack depth at the point of the buffer overflow vulnerability.

Project hints

- Use mod-l.seas.upenn.edu
  - minus.seas.upenn.edu still has stack protection turned on
  - 'uname -a' will give you some useful information about which machine you're connected to
- GCC has changed significantly since the Aleph One tutorial was written:
  - 16 bit vs. 32 bit architecture
  - GCC now automatically reserves bytes of "free" space in main() frame.
  - Syntax of inline assembly is different
  - GCC (>=4.1) supports canaries in main() frame, extra credit

If you must use C/C++

- Avoid the (long list of) broken library routines:
  - strcpy, strcat, sprintf, scanf, sscanf, *gets*, read, …
- Use (but be careful with) the "safer" versions:
  - e.g. strncpy, snprintf, fgets, …
- *Always* do bounds checks
  - One thing to look for when reviewing/auditing code
- Be careful to manage memory properly
  - Dangling pointers often crash program
  - Deallocate storage (otherwise program will have a memory leak)

- Be aware that doing all of this is difficult.

Defeating Buffer Overflows

- Use a typesafe programming language
  - Java/C# are not vulnerable to these attacks

- Some operating systems patch, E.g. eniac-l
  - move the start of the stack on a per-process basis
  - turn on non-executable stack
  - Gcc (>=4.1) supports Stack-Smashing-Protection (SSP)
    - Padding canaries between local vars and return address