CSE 451: Operating Systems
Spring 2003

Lecture 3
C and stack smashing

Steve Gribble
gribble@cs.washington.edu
323B Sieg Hall

Today's agenda

- Administrivia
  - office hours
    - Doug: Thursday at 11am in the TA offices (226a)
    - Valentin: Friday at 2pm, in 232
- A few last C brain teasers
- Exploiting your knowledge of C's weaknesses
  - broad strokes of buffer overrun vulnerabilities

Brain teasers…

#2: spot the bug

```
typedef struct ll_st {
  struct ll_st  *next;
  int           value;
} linked_list_element;

  ...

void free_linked_list(linked_list_element *head) {

  free(head);
  free_linked_list(head->next);

}

  ...
```


#3: spot the bug

```
typedef struct {
   char test_string[5];
} embedded_string;

char *extract_string(embedded_string extract_from_me) {
  return extract_from_me.test_string;
}

void main() {
   char *x;
   embedded_string y;

   …
   x = extract_string(y);
   strcpy(x, “hi!”);
   …
}
```

#4: predict the output

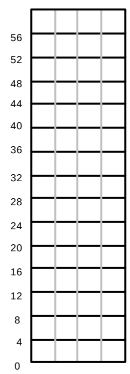
```
#include <stdio.h>

void main(void) {
  char input[256];

  gets(input);
  printf(“User inputted: ‘%s’\n”, input);

  return;
}
```

#5: spot the bugs

```
void foo(int print, int value) {
  char *string;

  string = (char *) malloc(10*sizeof(char));

  if (input > 1) {
    sprintf(string, “value: %d”, value);
    printf(string);
    free(string);
  }

  return;
}
```


#6: spot the bug (subtle)

```
unsigned short  x, *x_ptr;
unsigned int    y;
unsigned char  *c_ptr;

// assign some values
y = 0; x=0xFFFF;

// point x_ptr into the “middle” of y
c_ptr = (char *) (&y);
x_ptr = (unsigned short *) (c_ptr+1);
*y_ptr = x;
```

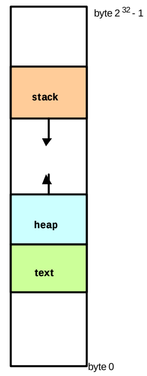


Buffer overrun vulnerabilities and C

Examples and structure taken from “Smashing the stack for fun and profit”, by Aleph One

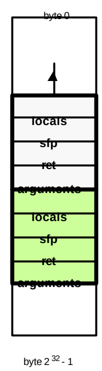
Memory Organization

- UNIX programs make use of three memory regions
  - heap
    - malloc'ed memory
  - text
    - code and static variables
  - stack
    - local variables



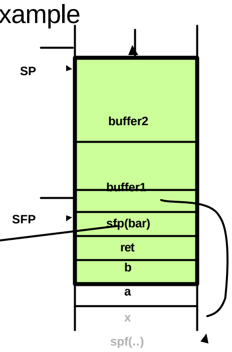
Stack frames

- By convention, the stack is organized into a set of *stack frames*
  - every time you call a procedure, you add a frame to the "top" of the stack
    - remember, stacks grow downwards on x86/Linux, so the "top" of the stack grows down towards the bottom of memory
  - a stack frame has to keep track of a number of things:
    - the arguments passed in to the procedure
    - the local variables used in the procedure
    - the return address to return to afterwards
    - the address of the previous stack frame to unwind the stack to



A specific example

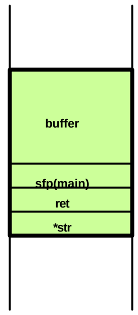
```
void func(int a, int b) {
  char buffer1[8];
  char buffer2[12];
}

void bar() {
  int x=1;

  func(1,2);
  x++;
}
```



4/6/2003 © 2003 Steve Gribble 12

Buffer overruns

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  large_string[255] = '\0';
  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```



Key insight

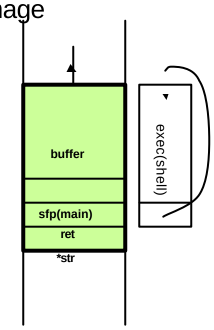
- We were able to change the return address by overflowing a buffer
  - many programs have buffer overrun vulnerabilities because of poor coding techniques:

```
char *x;
char  header[30];

x = read_www_request_from_network();
strcpy(header, x);
```

  - why not exploit this vulnerability by changing the return address to something, er, “creative”?

Controlled damage

- Overrun buffer to accomplish two things:
  - embedded carefully constructed code in the buffer (and therefore in the stack)
    - execute a shell
    - mail /etc/passwd somewhere
  - overwrite return address to divert the program to your code
    - difficult: figuring out what address to stuff in there
    - need to know where (precisely) the buffer is in memory



The remaining pieces

- constructing a buffer that is valid code
  - write a program to execve ("/bin/sh")
    - use gdb to disassemble
    - construct string using disassembly
- execve( ) has a pointer to a string as argument
  - you are writing code that you are "inserting" in another program
  - the code you are writing doesn't know where it will live in that other program
  - your code needs to figure out where itself lives once it gets there, so it can pass the right address of the string "/bin/sh" as an argument
- you need to figure out where the buffer you are overflowing lives
  - so that you can overwrite the return address to point to the buffer

4/6/2003 16

Buffer overruns: huge problem for Internet

- buffer overruns are exploited heavily
  - worms (SQL slammer, Morris worm, etc.)
  - distributed denial of service attack zombie recruit tools
  - hackers attempting to bust specific machines
- buffer overruns are 50% of reported vulnerabilities
- buffer overruns exist in many widely deployed software packages
  - including sendmail, which runs on most Unix systems

4/6/2003 17

Combating buffer overruns

- Better languages
  - eliminate buffer overruns altogether [Java]
  - doesn't deal with legacy code
- Tools to inspect legacy code
  - easy to find some bugs ("grep gets *.c")
  - hard to find all bugs this way, because C is so messy
- Operating system or architecture support
  - randomize stack placement in programs
  - make the stack non-executable

4/6/2003 18