CSCE 465 Computer & Network Security

Instructor: Abner Mendoza

Week 1/2 Recap

- Security definition, components/objectives
- Security threats and attacks
- Achieving security: Security Policy, Mechanism, Assurance
- Basic Network Security

Program Security: Buffer Overflow

BUFFER OVERFLOW BASICS

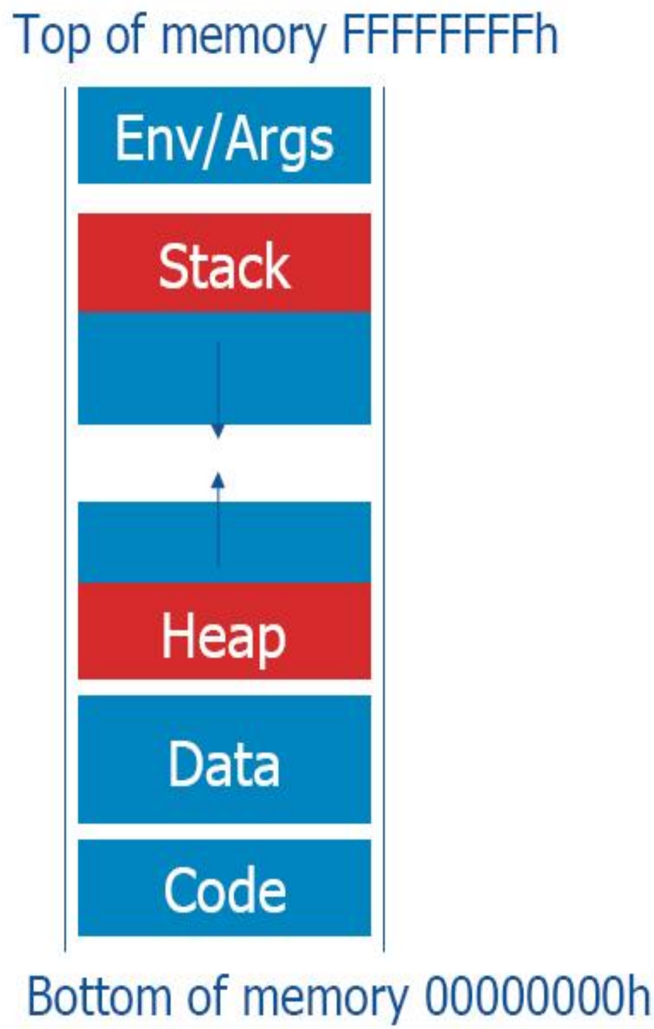
Introduction

- What is a buffer overflow?
  - A *buffer overflow* occurs when a program writes data outside the bounds of allocated memory.

- Buffer overflow vulnerabilities are exploited to overwrite values in memory to the advantage of the attacker

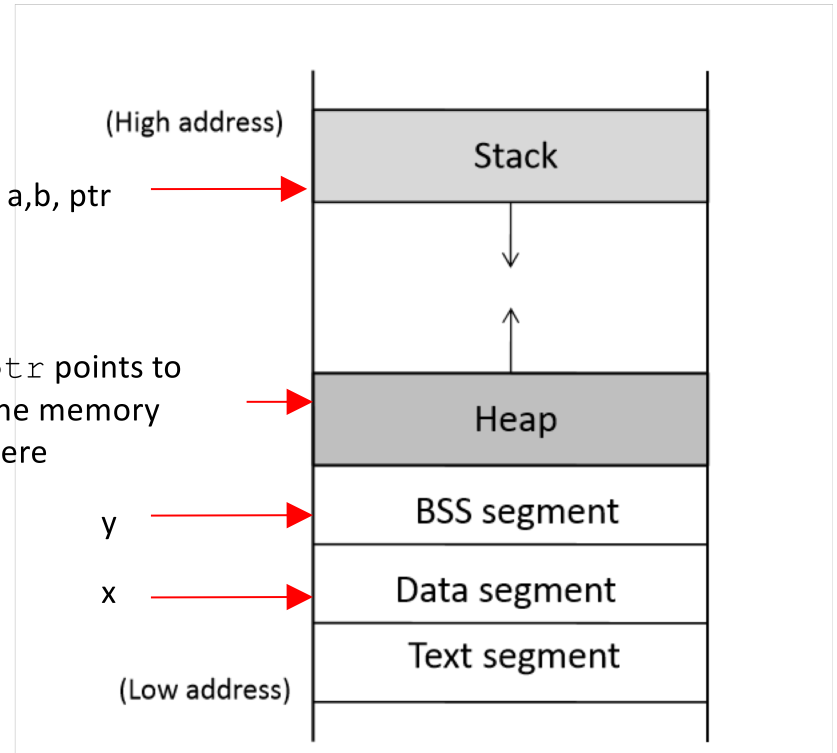
Process Memory Structure

- Code/Text section (.text)
- Data section (.data, .bss)
- Heap section
  - Used for dynamically allocated data
- Stack section
- Environment/Argument section
  - Used for environment data
  - Used for the command line data



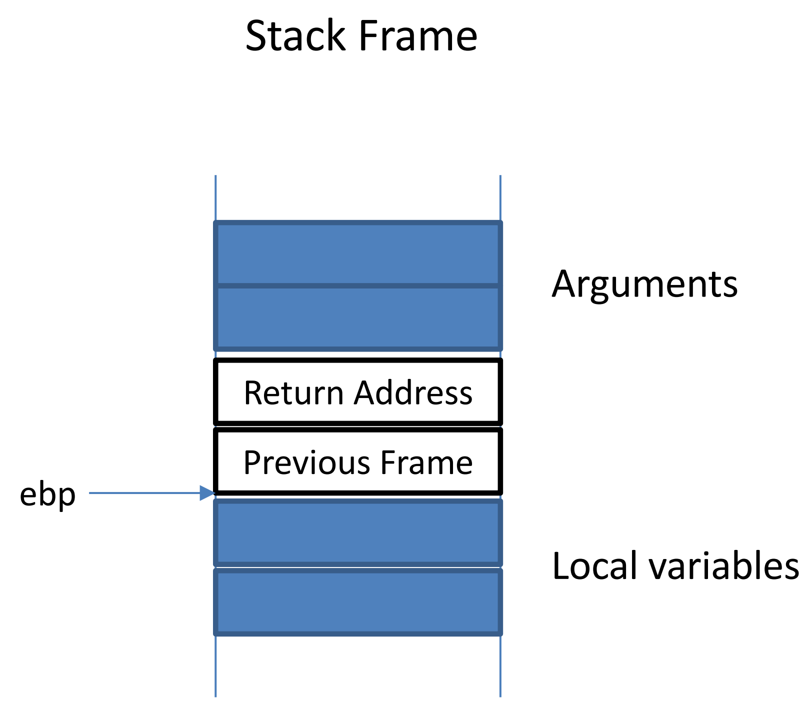
Process Memory Structure

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



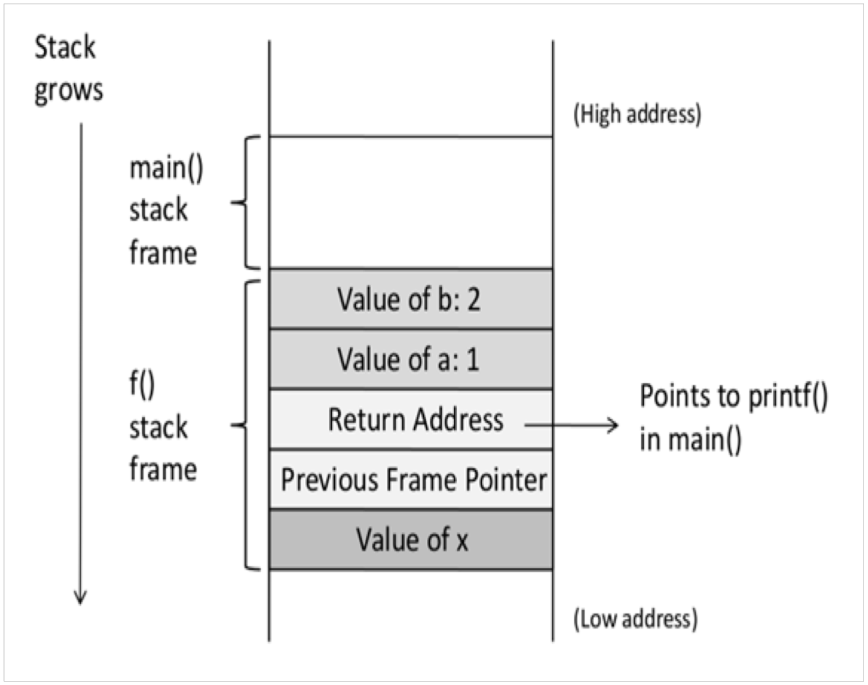
Example

```
void foo(int a, int b){
   int x,y;
   x = a+b;
   y = a-b;
}
```



Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



Virtual vs Physical Memory

- Most processes use virtual memory
- Mapping from Virtual to Physical handled by the OS

Impact

- Firstly widely seen in the first computer worm -- Morris Worm (1988, 6,000 machines infected)

- Buffer overflow is still the most common source of security vulnerability

- SANS (SysAdmin, Audit, Network, Security) Institute report that 14/20 top vulnerabilities in 2006 are buffer overflow-related

- Also behind some of the most devastation worms and viruses in recent history e.g. Zotob, Sasser, CodeRed, Blaster, SQL Slammer, Conficker, Stuxnet …

BO Attacks

- Goal: subvert the function of a privileged program so that the attacker can take control of that program, and if the program is sufficiently privileged, thence control the host.
- Involves:
  - Code present in program address space
  - Transfer execution to that code

Placing code in address space

- 2 ways to achieve subgoal:
  - Inject user code
  - Use what’s already there

Code Injection

- Code Injection: provide a string as input to the program, which the program stores in a buffer. The string contains native CPU instructions for the platform being attacked
- Works with buffers stored anywhere

Code already there

- Code of interest already in part of program
- Attacker only needs to call it with desired arguments before jumping to it
- E.g. Attacker seeks to acquire a shell, but code already in some library contains a call to exec(arg). Attacker must only pass a pointer to the string “/bin/sh” and jump to ‘exec’ call

How to jump to Attacker Code

- Activation Records: stores return address of function. Attacker modifies pointer to point to his code. This technique is known as “stack smashing”
- Function Pointers: similar idea, but seeks to modify an arbitrary function pointer.
- Longjmp buffers: again, the attacker modifies the buffer with his malicious code

Attacks on Memory Buffers

- **Buffer** is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
    - If more data is stuffed into it, it spills into adjacent memory
  - If executable code is supplied as "data", victim's machine may be fooled into executing it – we'll see how
    - Code will self-propagate or give attacker control over machine
- **First generation exploits:** stack smashing
- **Second gen:** heaps, function pointers, off-by-one
- **Third generation:** format strings and heap management structures

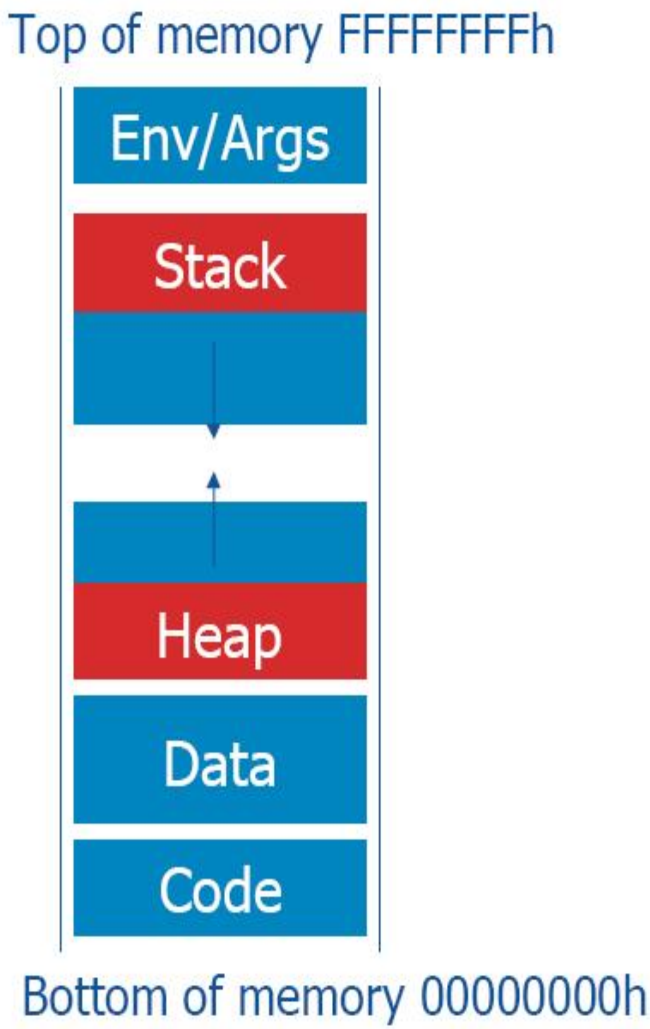
STACK SMASHING

Stack Smashing

- Process memory is organized into three regions : Text, Data and Stack
- Text/code section (.text)
  - Includes instructions and read-only data
  - Usually marked read-only
  - Modifications cause segment faults
- Data section (.data, .bss)
  - Initialized and uninitialized data
  - Static variables
  - Global variables
- Stack section
  - Used for implementing procedure abstraction

Process Memory Structure

- Code/Text section (.text)
- Data section (.data, .bss)
- Heap section
  - Used for dynamically allocated data
- Stack section
- Environment/Argument section
  - Used for environment data
  - Used for the command line data

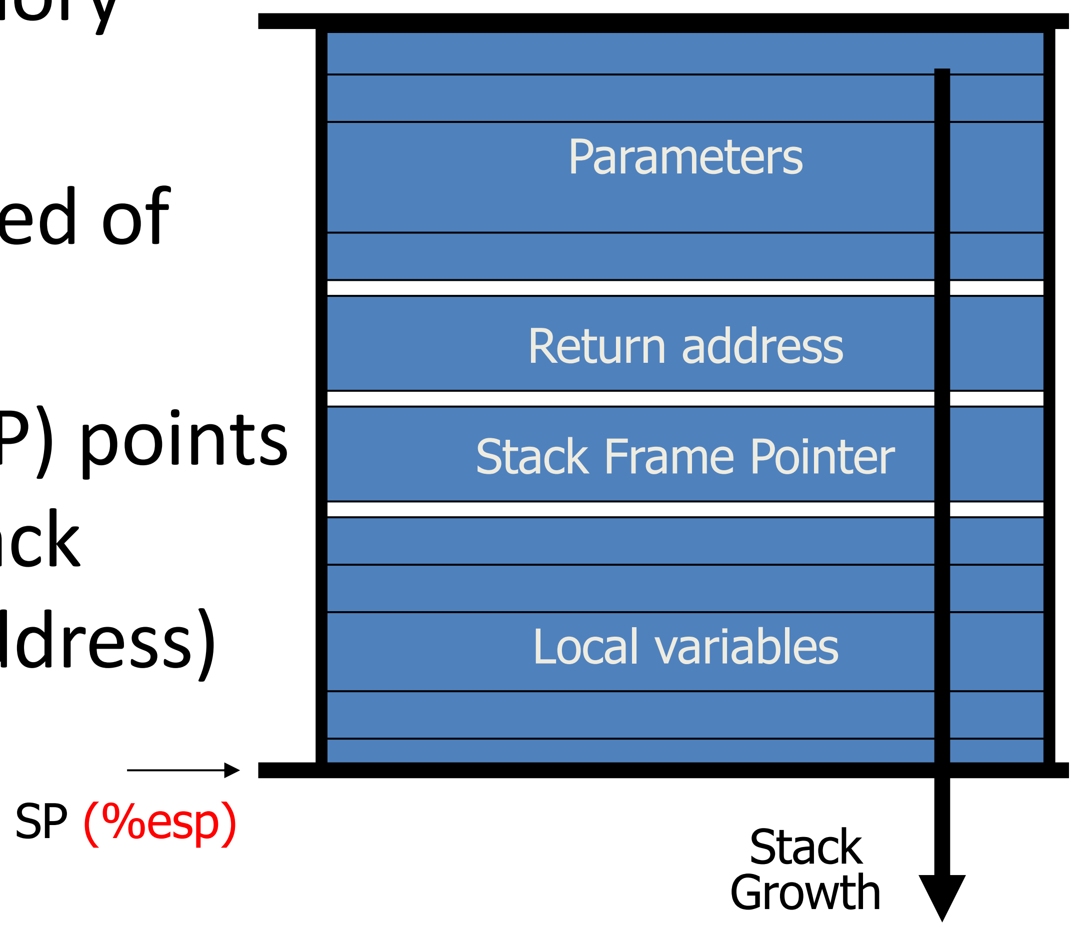


What Happens When Memory Outside a Buffer Is Accessed?

- If memory doesn't exist:
  - –Bus error
- If memory protection denies access:
  - –Page fault
  - –Segmentation fault
  - –General protection fault
- If access is allowed, memory next to the buffer can be accessed
  - –Heap
  - –Stack
  - –...

Stack Frame

- The stack usually grows towards lower memory addresses
- The stack is composed of frames
- The stack pointer (SP) points to the top of the stack (usually last valid address)



Stack Buffers

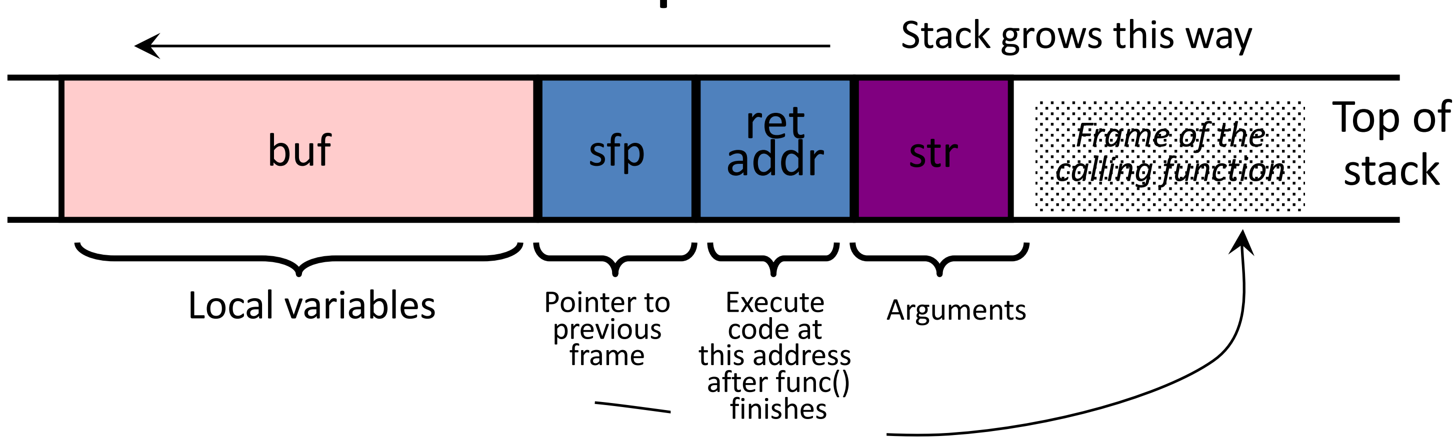
- Suppose Web server contains this function

```
void func(char *str) {
      char buf[126];
      strcpy(buf,str);
}
```

Allocate local buffer (126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame with local variables is pushed onto the stack



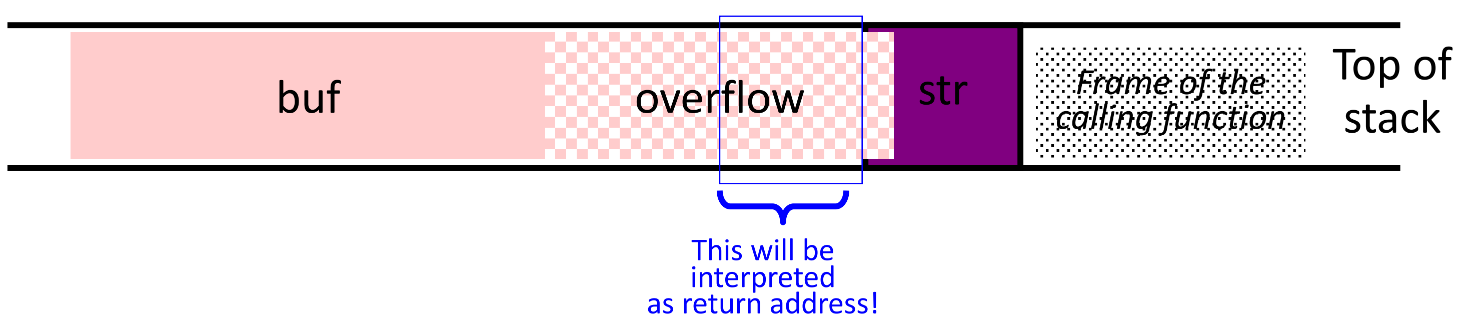
What If Buffer Is Overstuffed?

- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
      char buf[126];
      strcpy(buf,str);
}
```

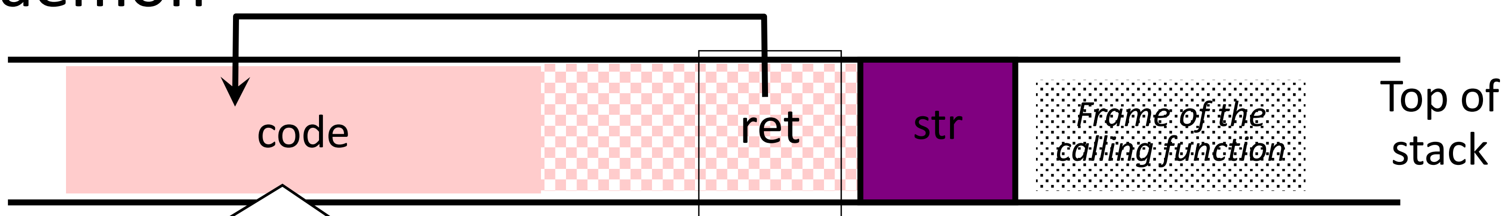
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon

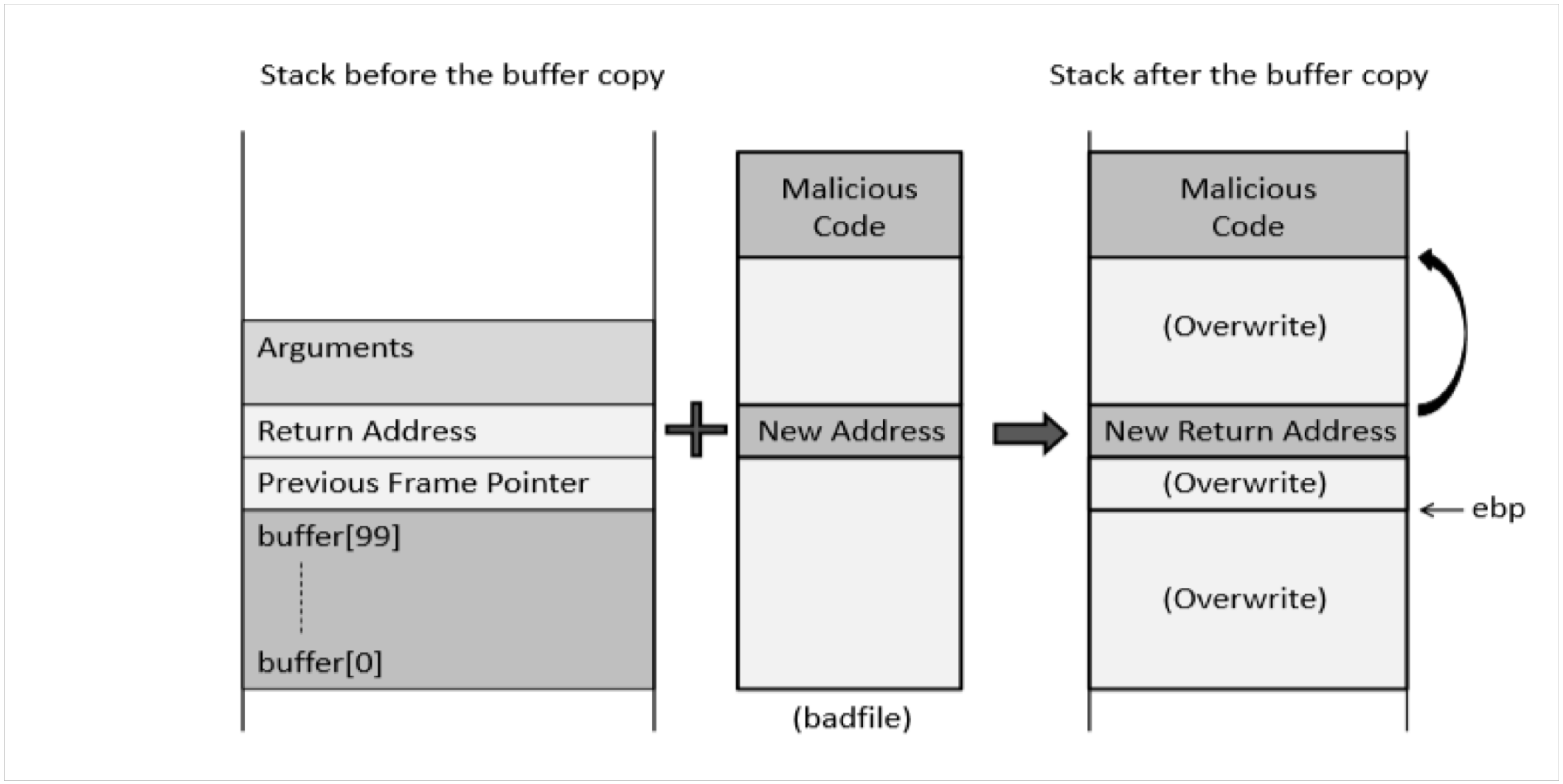


Attacker puts actual assembly instructions into his input string, e.g., binary code of execve("/bin/sh")

In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address

- When function exits, er will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

How to Run Malicious Code



The Shell Code

```
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    exit(0); }
```

- System calls in assembly are invoked by saving parameters either on the stack or in registers and then calling the software interrupt (0x80 in Linux)

Attack Procedure High Level View

- Compile attack code
- Extract the binary for the piece that actually does the work (shell code)
- Insert the compiled code into the buffer
- Figure out where overflow code should jump
- Place that address in the buffer at the proper location so that the normal return address gets overwritten

Buffer Overflow Issues

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- Overflow portion of the buffer must contain **correct address of attack code** in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

Guessing the Buffer Address

- In most cases the address of the buffer is not known
- It has to be “guessed” (and the guess must be very precise)
- Given the same environment and knowing size of command-line arguments the address of the stack can be roughly guessed
- The stack address of a program can be obtained by using the function

```
unsigned long get_sp(void) {
__asm__("movl %esp,%eax");
}
```

- We also have to guess the offset of the buffer with respect to the stack pointer

NOP Sled

- Use a series of NOPs at the beginning of the overflowing buffer so that the jump does not need to be exactly precise

- This technique is called no-op sled

Set-UID Concept

- **Allow user to run a program with the program owner's privilege.**
- Allow users to run programs with temporary elevated privileges
- Example: the `passwd` program

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 41284 Sep 12  2012 /usr/bin/passwd
```

Set-UID Concept

- Every process has two User IDs.
- **Real UID (RUID)**: Identifies real owner of process
- **Effective UID (EUID)**: Identifies privilege of a process
  - Access control is based on EUID
- When a normal program is executed, RUID = EUID, they both equal to the ID of the user who runs the program
- When a Set-UID is executed, RUID ≠ EUID. RUID still equal to the user's ID, but EUID equals to the program **owner**'s ID.
  - If the program is owned by root, the program runs with the root privilege.

OTHER BO VULNERABILITIES

Off-By-One Overflow

- Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
      char buffer[512]; int i;
      for (i=0; i<=512; i++)
            buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
      if (argc==2)
            notSoSafeCopy(argv[1]);
}
```

This will copy **513** characters into buffer. Oops!

1-byte overflow: can't change RET, but can change pointer to <u>previous</u> stack frame

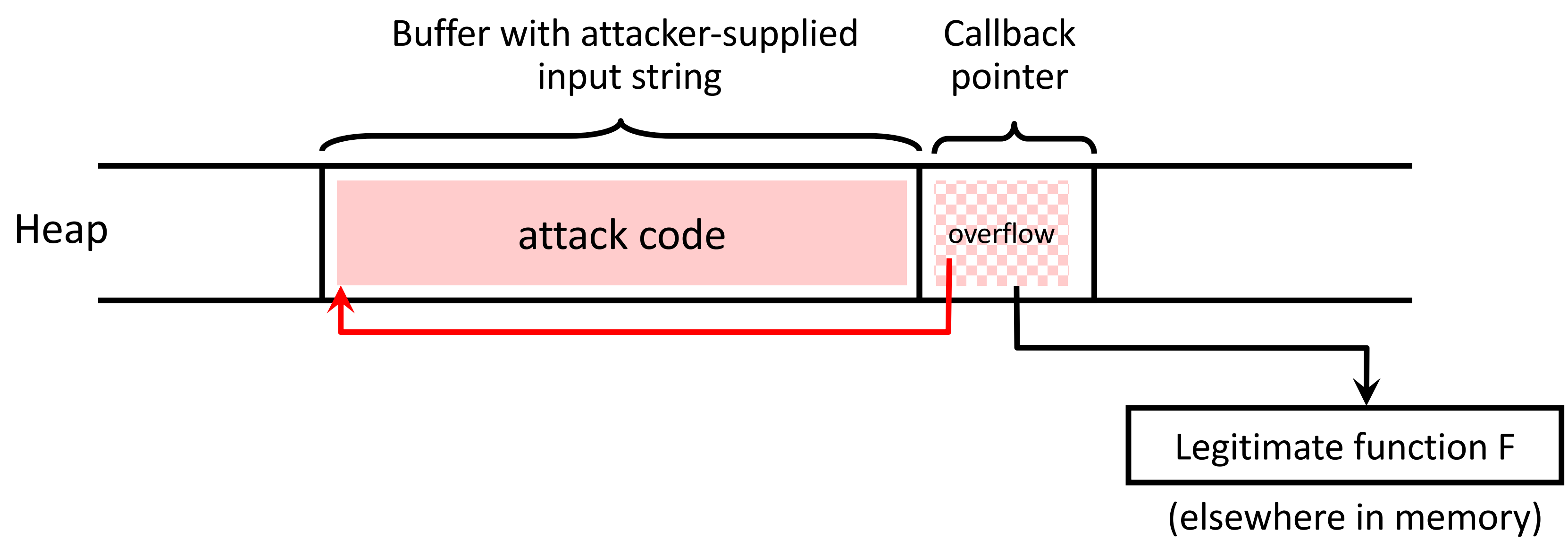
- On little-endian architecture, make it point into buffer
- RET for previous function will be read from buffer!

Heap Overflow

- Overflowing buffers on heap can change pointers that point to important data
  - Sometimes can also transfer execution to attack code
  - Can cause program to crash by forcing it to read from an invalid address (segmentation violation)
- Illegitimate privilege elevation: if program with overflow has sysadm/root rights, attacker can use it to write into a normally inaccessible file
  - For example, replace a filename pointer with a pointer into buffer location containing name of a system file
    - Instead of temporary file, write into AUTOEXEC.BAT

Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then another function G can call F as (*P)(…)



Format Strings in C

- Proper use of printf format string:

```
… int foo=1234;
  printf("foo = %d in decimal, %X in hex",foo,foo); …
```

  - This will print

```
foo = 1234 in decimal, 4D2 in hex
```

- Sloppy use of printf format string:

```
… char buf[13]="Hello, world!";
  printf(buf);
  // should've used printf("%s", buf); …
```

  - If buffer contains format symbols starting with %, location pointed to by printf's internal stack pointer will be interpreted as an argument of printf. This can be exploited to move printf's internal stack pointer.

Writing Stack with Format Strings

- %n format symbol tells printf to write the number of characters that have been printed

```
… printf("Overflow this!%n",&myVar); …
```

  - Argument of printf is interpeted as destination address
  - This writes 14 into myVar ("Overflow this!" has 14 characters)

- What if printf does not have an argument?

```
… char buf[16]="Overflow this!%n";
  printf(buf); …
```

  - Stack location pointed to by printf's internal stack pointer will be interpreted as address into which the number of characters will be written

More Buffer Overflow Targets

- Heap management structures used by malloc()
- URL validation and canonicalization
  - If Web server stores URL in a buffer with overflow, then attacker can gain control by supplying malformed URL
    - Nimda worm propagated itself by utilizing buffer overflow in Microsoft's Internet Information Server
- Some attacks don't even need overflow
  - Naïve security checks may miss URLs that give attacker access to forbidden files
    - For example, http://victim.com/user/../../autoexec.bat may pass naïve check, but give access to system file
    - Defeat checking for "/" in URL by using hex representation

BO DEFENSE

Buffer Overflow Defenses

- Writing correct code
- Non-executable buffers
- Randomize stack location or encrypt return address on stack by XORing with random string
  - Attacker won't know what address to use in his string
- Array bounds checking
- Code pointer integrity checking

Writing correct code

- Use safe programming languages, e.g., Java
  - What about legacy C code?
- Use compilers that warn about linking to unsafe functions e.g. gcc
- Static analysis of source code to find overflows
- Black-box testing with long strings
- Use safer versions of functions
  e.g., gets and strcpy should be replaced with getline and strlcpy

Problem: No Range Checking

- strcpy does not check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Does Range Checking Help?

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
    - Programmer has to supply the right value of n
- Potential overflow in htpasswd.c (Apache 1.3):

```
… strcpy(record,user);
  strcat(record,":");
  strcat(record,cpw); …
```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

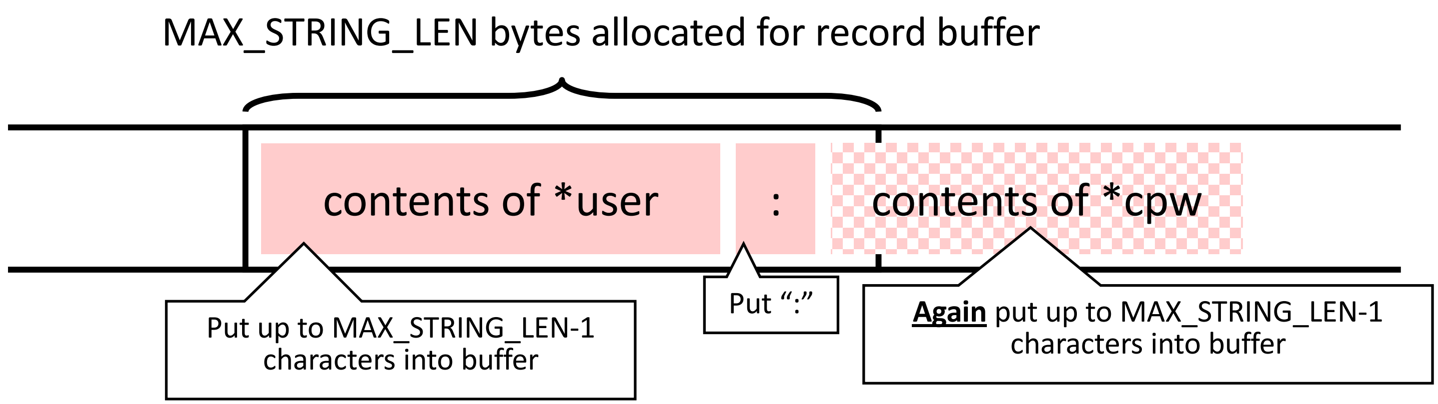
- Published "fix" (do you see the problem?):

```
… strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,":");
  strncat(record,cpw,MAX_STRING_LEN-1); …
```

Misuse of strncpy in htpasswd "Fix"

- Published "fix" for Apache htpasswd overflow:

```
… strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,":");
  strncat(record,cpw,MAX_STRING_LEN-1); …
```



- Note: Strlcpy can count and return the length of the entire source string while strncpy cannot

Bugs to Detect in Source Code Analysis

- Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code

- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Non-executable buffers

- Works by marking a region of memory as non-executable. To stop buffer overflow, exploits, the data section has to be marked non-executable.
- Problem with recent systems, since they emit executable code within the data section, but more applicable to stack segment since no legitimate program has code in stack.

Non-Executable Stack

- NX bit on every Page Table Entry
  - AMD Athlon 64, Intel P4 “Prescott”, but not 32-bit x86
  - Code patches marking stack segment as non-executable exist for Linux, Solaris, OpenBSD
- Some applications need executable stack
  - For example, LISP interpreters
- Does not defend against return-to-libc exploits
  - Overwrite return address with the address of an existing library function (can still be harmful)
- …nor against heap and function pointer overflows

Address Randomization: Motivations.

- Buffer overflow and return-to-libc exploits need to know the (virtual) address to which pass control
  - Address of attack code in the buffer
  - Address of a standard kernel library routine
- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine

- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Address Space Layout Randomization

- Arranging the positions of key data areas randomly in a process' address space.
  - e.g., the base of the executable and position of libraries (libc), heap, and stack,
  - Effects: for return to libc, needs to know address of the key functions.
  - Attacks:
    - Repetitively guess randomized address
    - Spraying injected attack code
- Vista/Windows 7 has this enabled, software packages available for Linux and other UNIX variants

Array bounds checking

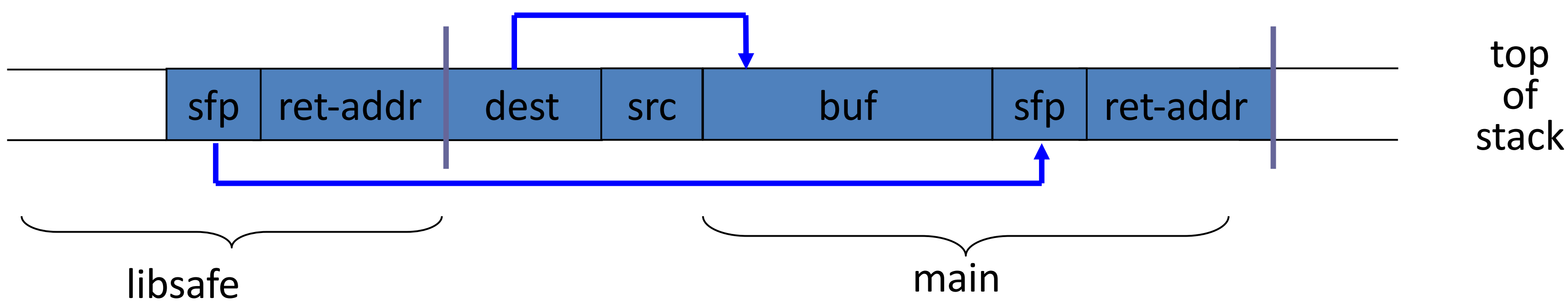
- Completely stops BO attacks
- All reads and writes to arrays will be bound checked. This is the case with memory-safe languages like Java and .net languages
- Solves the problem at the cost of performance

Run-Time Checking: Libsafe

- Dynamically loaded library
- Intercepts calls to strcpy(dest,src)
  - Checks if there is sufficient space in current stack frame

|frame-pointer – dest| > strlen(src)

  - If yes, does strcpy; else terminates application

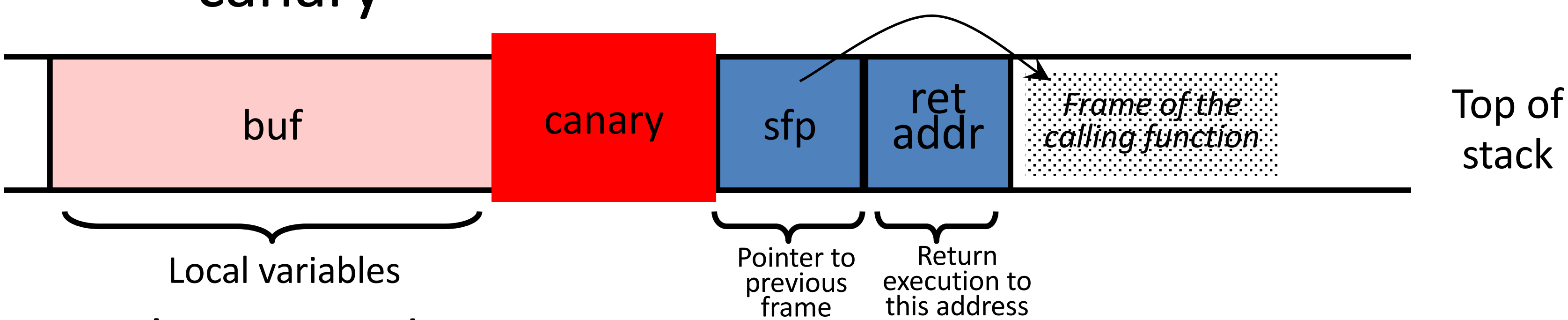


Code pointer integrity checking

- Works by detecting whether a code pointer e.g. return address, has been corrupted before dereferencing it.
- Prevents only BO attacks exploiting automatic buffers
- Much better performance than array bounds checking
- Eg. StackGuard

Run-Time Checking: StackGuard

- Embed "canaries" in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- PointGuard also places canaries next to function pointers and setjmp buffers
  - Worse performance penalty
- StackGuard can be defeated!
  - Phrack article by Bulba and Kil3r

Next Week

- Intro to Malware
- Malware and Reverse Engineering Basics
- Homework 2 Released