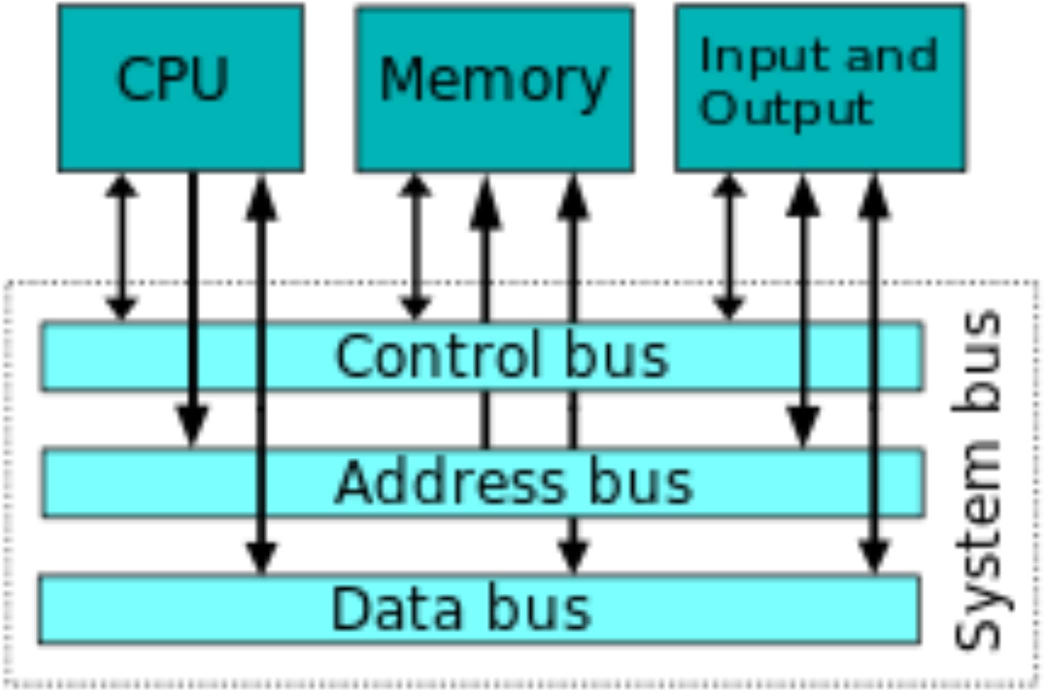
Essential C Security 101

Offensive Computer Security
Florida State University
Spring 2014

Outline of talk

- Intro to CPU & Registers
- Motivation
- Strings
- Pointers
- Dynamic Memory Management

Von Neumann Architecture



Registers (General Purpose)

EAX - Accumulator

- holds return value usually

EBX - Accumulator

- base

ECX - Count & Accumulator

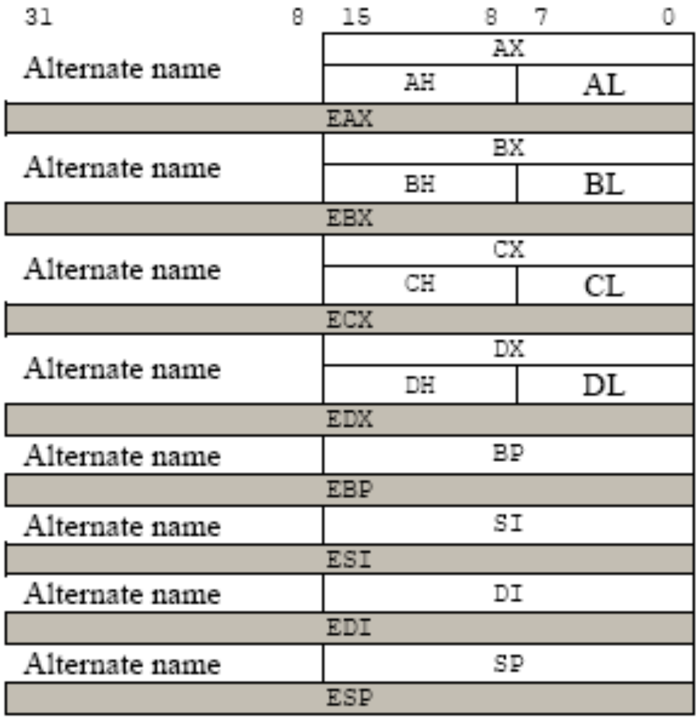
EDX - Data or I/O Address pointer

ESI - Source index

- for source of string / array operands

EDI - Destination index

- for dest of string / array operands



Registers (Important Ones)

EIP - Instruction Pointer

- (Points to Next instruction to be executed)
- Want to hijack during exploitation

ESP

- Stack pointer

EBP

- Stack base pointer

Tool we will be using

http://gcc.godbolt.org/

A project that visualizes C/C++ to Assembly for you. *(use g++ compiler, intel syntax, and no options)*

Quite useful for learning this stuff

(also interesting: https://github.com/ynh/cpp-to-assembly)

Lecture Source Material

[1] Seacord, Robert C. “Secure Coding in C and C++”. Second Edition. Addison Wesley. April 12, 2013

(not required, but highly recommended)

Motivation

- One of the most widely used programming languages of all time
  - Want to use a different language?
    - It’s backend is likely written in C!
      - Python
      - Ruby
      - Java!
- Vast majority of popular languages borrow from it

About C

Dennis Ritchie at ATT Labs

<u>Standards:</u>

- ANSI C89 (American National Standards Institute -- no longer around)
- ISO C90 (Int'l Org for Standarization)
- ISO C99
- ISO C11 (December 2011)
  - Dennis Ritchie died October 2011
    Way cooler than Steve Jobs...

TURING AWARD == BOSS

CCCCCCCCCCCCCCCCCCCCCC\xCC

USED IN EVERYTHING!

45 years and going strong!

- Operating Systems
- Embedded Systems
  - *Planes, Trains, Satellites, Missiles, Boats, etc..*
- Drivers, Libraries, Other languages...

You just cannot get away from it.

Strings

- String Types
- String functions
- Common Errors / Vulnerabilities
- Mitigations

Some C Terms for strings

- String - sequence (array) of characters up to and including the null character terminating it
- Length - the length of the sequence up till (not counting) the null character
- Size - number of bytes allocated to the array
- Count - number of elements in the array

size != length (depends on character size)

Length of Character / String

Atomic size (# bytes) of string depends on length of character!

- A single UTF-8 char = 1-4 bytes
- wide char = 2-4 bytes

Strings can be:

1. normal / “narrow”
2. wide character
3. multi-byte (heterogenous char types!)

Characters

char types:

1. char
2. unsigned char
3. signed char

wchar_t types:

- wchar_t
- unsigned wchar_t
- signed wchar_t

In general wchar_t is not meant to be signed / unsigned.

whcar_t is a integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales [1]

Examining characters

```
#include <string.h>
void foo()
{
  size_t len;
  char x;
  signed wchar_t y;
  unsigned char z;
  signed char zz;

  len = sizeof(x);
  len = sizeof(y);
  len = sizeof(z);
  len = sizeof(zz);
}
```

wchar_t

Windows typically uses UTF-16

- wchar_t is thus 2 bytes (16 bits…)

Linux / OSX typically uses UTF-32

- wchar_t is thus 4 bytes (32 bits…)

sizeof(wchar_t) is usually 2 or more bytes

- <u>size of a wchar_t array != count of the array</u>

length functions

- strlen (run time)
- sizeof (compile time)
- wcslen (for wide characters)
- …

Characters (from [1] page 38)

```
#include <string.h> // use compiler opt -fpermissive
void foo()
{
  size_t len;
  char cstr[] = "char string";
  signed char scstr[] = "char string";
  unsigned char uscstr[] = "char string";

  len = strlen(cstr);
  len = strlen(scstr);   // will trigger warnings
  len = strlen(uscstr); // will trigger warnings
}
```

strlen vs sizeof (derived from [1])

```
#include <string.h>
void foo()
{
  size_t len;
  char cstr[] = "char string";
  signed char scstr[] = "char string";
  unsigned char uscstr[] = "char string";

  len = strlen(cstr);
  len = sizeof(scstr);  // no warnings!  returns hardcoded value!
  len = sizeof(uscstr); // no warnings!  returns hardcoded value!
}
```

string functions

Copying:

• memcpy	Copy block of memory
• memmove	Move block of memory
• strcpy	Copy string (unbounded)
• strncpy	Copy characters from string
• strcpy_s	(A windows function, not C99/C11)
• strdup	(a POSIX function, not C99/C11)
• wcscpy	(A windows function, not C99/C11)
• wcscpy_s	(A windows function, not C99/C11)
• _mbscpy	(A windows function, not C99/C11)
• _mbscpy_s	(A windows function, not C99/C11)

string functions

Concatenation:

- strcat Concatenate strings
- strncat Append characters from string
- sprintf Format strings (also copying)
- snprintf Format strings (also copying)

Common Errors

We’ll cover some common errors:

- improperly bounded string copies
- off-by-one errors
- string truncation
- null termination errors

Things that cause “UNDEFINED BEHAVIOR” :)

- <u>potentially memory corruption</u>

improperly bounded string copies

Common culprits of old (now depreciated)

- gets (cannot be used safely)
- strcpy (unsafe, but can be used safely)

Newer common culprits… misuse of:

- strncpy
- memmove
- memcpy
- sprintf / snprintf

improperly bounded string copies

example from [1] p42. Short link to this in gcc.godbolt: [here]()

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

void foo() {
  char response[8];
  puts("Continue? [y] n: ");
  gets(response);
  if (response[0] =='n')
        exit(0);
  return;
}
```

```
char *gets(char *dest)
{
        int c = getchar();
        char *p = dest;
        while (c!= EOF && c != '\n')
        {
                *p++ = c;
                c = getchar();
        }
        *p = '\0';
        return dest;
}
```

improperly bounded string copies

```
#include <string.h>
int some_function(char *inputstring)
{
	char buf[256];
	/* make a temp copy of data to work on */
	strcpy(buf, inputstring);
	/* … */
	return 0;
}
```

improperly bounded string copies

```
#include <string.h>
int maybe_safer_function(char *inputstring)
{
	char buf[256];
	/* make a temp copy of data to work on */
	strncpy(buf, inputstring, strlen(inputstring));
	/* … */
	return 0;
}
```

The lesson:

- make sure “safe” functions are used correctly
  - otherwise no guarantee of safety / defined behavior

improperly bounded string copies

```
#include <string.h>
int some_other_function(char *inputstring)
{
        char buf[256];
        /* make a temp copy of data to work on */
        sprintf(buf, "%s",  inputstring);
        /* … */
        return 0;
}
```

off-by-one errors

Similar to unbounded copies

- involves reading/writing outside the bounds of the array

off-by-one errors (from [1] page 47)

```
void foo(){
  char s1[] = "012345678";  // len 9
  char s2[] = "0123456789";  // len 10
  char *dest; int i;

  strncpy(s1, s2, sizeof(s2));
  dest = (char * ) malloc(strlen(s1));

  for (i =1; i <=11; i++)
      dest[i] = s1[i];

  dest[i]='\0';
  printf("dest = %s", dest);
}
```

string truncation

When too large of a string is put *safely* into too small of a destination. Data is lost

- Sometimes this can lead to a vulnerability
  - Depends on application logic

null termination errors

- failure to properly null terminate strings
- strncpy/snprintf/strncat don’t null terminate

Mitigations

Follow best encoding practices:

- http://websec.github.io/unicode-security-guide/character-transformations/

Compiler flags:

- use safe functions safely
    - Adopt a single / unified model for handling strings (cover this at the end)
- _FORTIFY_SOURCE
    - stack cookies (we'll cover this in depth later)

Pointers

- How to
- Function Pointers
- Data Pointer Errors
- Global Offset Table (GOT)
- .dtors section

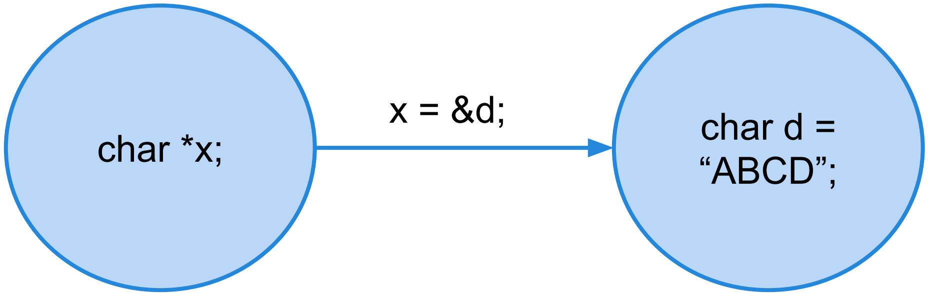
Pointer Operators

* and &

* (declaration operator)

* when used in declaring a variable instantiates (or type casts) a variable pointing to an object of a given type

- char *x; // x points to a char object / array
- wchar_t *y;
- int *z;



* (dereference operator)

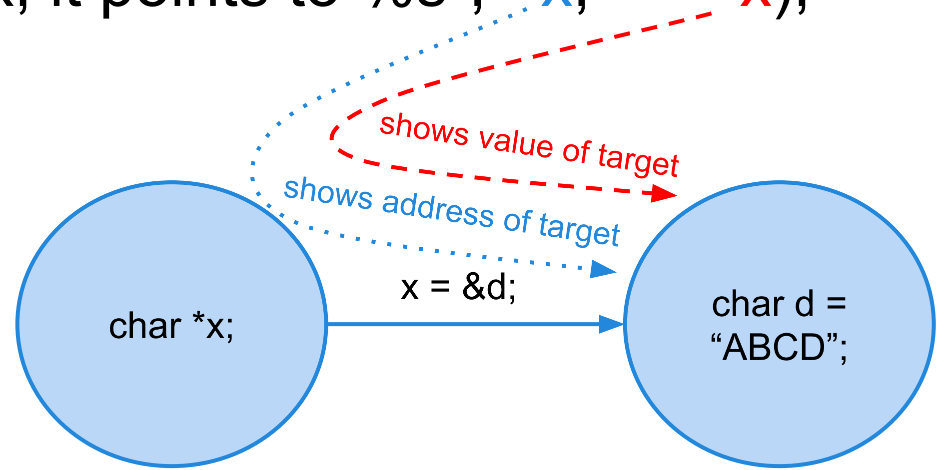
* is a unary operator which denotes indirection

- if the operand doesn’t point to an object or function, the behavior of * is undefined
    - *(NULL) will typically trigger a segfault
        - or vulnerability if 0x000000000000 is a valid memory-mappable address :)
            - OLD SCHOOL computers, but also many modern embedded systems

* (dereference operator)

Think of it as it moves forwards in this relationship.

printf("x contains at 0x%08x, it points to %s", x, *x);

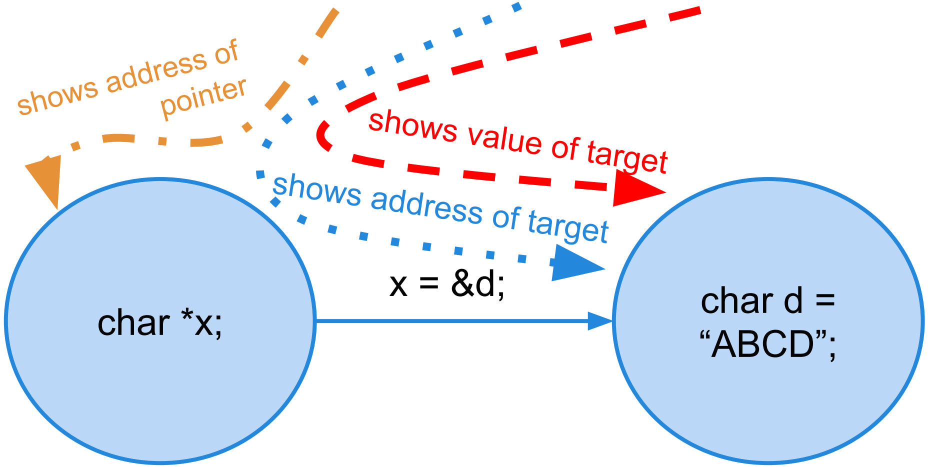


& (address-of operator)

& shows you the actual data stored in the pointer

printf("x is at 0x%08x, contains 0x%08x, it points to %s",

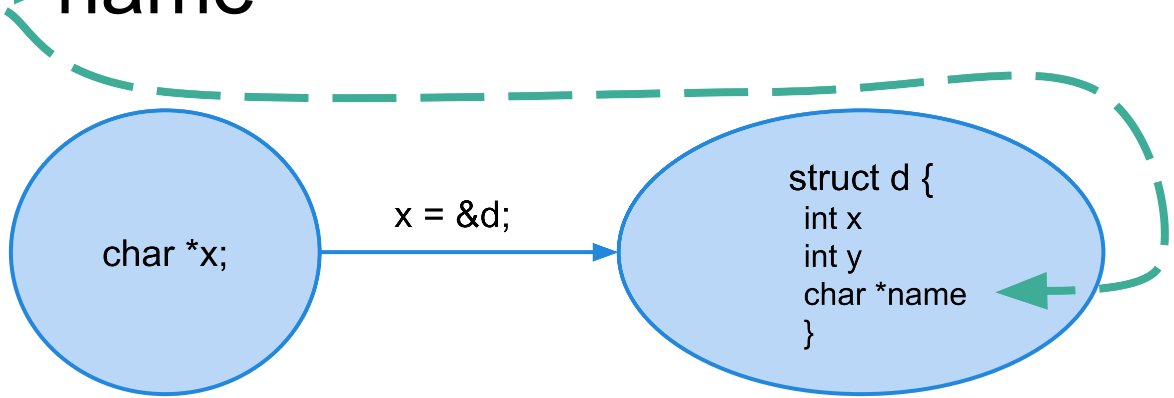
&x, x, *x);



-> (member-of operator)

-> dereferences a structure and accesses a member of that structure

- p->next (for linked lists)
- d->name



array indexing

expr1[expr2]

- returns the expr2th element of the `array' pointed to by expr1. Exactly equivalent to:
  - *(expr1 + (expr2))

d->name is equivalent to:

- (char *)*(d + sizeof(x) + sizeof(y))

Function Pointers

These get executed.

- via: call, jmp, jcc, ret...
- if they point to malicious instructions, will execute
- must be handled carefully

Function Pointers (from [1] p 126)

```c
#include <stdio.h>

void good_function(const char *str){
        printf("%s", str);
}

int main(){
  static void (*funcPtr)(const char *str);
  funcPtr = &good_function;
  (void)(*funcPtr)("hi ");
  good_function("there!\n");
  return 0;
}
```

```
mov   rax, QWORD PTR main::funcPtr[rip]
mov   edi, OFFSET FLAT:.LC1
call  rax

mov   edi, OFFSET FLAT:.LC2
call  good_function(char const*)
```

Data Pointers errors (Lets see the difference)

```c
// Bad developer
#include <string.h>
#include <stdlib.h>

int main(void){
  char s1[] = "012345678";
  char dest;

  dest = *(char * ) malloc(strlen(s1));
}
```

```c
// Good developer
#include <string.h>
#include <stdlib.h>

int main(void){
  char s1[] = "012345678";
  char *dest;

  dest = (char * ) malloc(strlen(s1));
}
```

1 byte

8 bytes (64 bit machine)

Data Pointers errors (Lets see the difference) The good

mov QWORD PTR [rbp - 8], RAX

This moves 8 bytes (QWORD size) to dest.

dest is at [rbp - 8], and the -8 is simply where it is on the stack relative to the base pointer. (we'll cover this in detail later)

Code editor

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

int main(void){
   char s1[] = "012345678";
   char *dest;

   dest = (char * ) malloc(strlen(s1));

}
```

Assembly output

```
main:
        push    rbp
        mov rbp, rsp
        sub rsp, 32
        movabs  rax, 3978425819141910832
        mov QWORD PTR [rbp-32], rax
        mov WORD PTR [rbp-24], 56
        lea rax, [rbp-32]
        mov rdi, rax
        call    strlen
        mov rdi, rax
        call    malloc
        mov QWORD PTR [rbp-8], rax
        mov eax, 0
        leave
        ret
```

Data Pointers errors (Lets see the difference) The bad

movzx EAX, BYTE PTR [rax]

mov BYTE PTR [rbp - 1], al

This moves 8 bits to dest

Code editor

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

int main(void){
  char s1[] = "012345678";
  char dest;

  dest = *(char * ) malloc(strlen(s1));

}
```

Assembly output

```asm
main:
        push    rbp
        mov rbp, rsp
        sub rsp, 16
        movabs  rax, 3978425819141910832
        mov QWORD PTR [rbp-16], rax
        mov WORD PTR [rbp-8], 56
        lea rax, [rbp-16]
        mov rdi, rax
        call    strlen
        mov rdi, rax
        call    malloc
        movzx   eax, BYTE PTR [rax]
        mov BYTE PTR [rbp-1], al
        mov eax, 0
        leave
```

Global Offset Table (GOT)

Windows & Linux use a similar technique for linking and transferring control to a library function

- linux’s is exploitable
- windows’s is safe

Global Offset Table (GOT)

As part of the Executable and Linking Format (ELF), there is a section of the binary called the Global Offset Table

- The GOT holds absolute addresses
  - essential for dynamically linked binaries
  - every library function used by program has a GOT entry
    - contains address of the actual function

Global Offset Table (GOT)

Before the first use of a library function, the GOT entry points to the run time linker (RTL)

- RTL is called (passed control),
    - RTL finds function’s real address and inserted into the GOT

Subsequent calls don’t involve RTL

Global Offset Table (GOT)

GOT is located at a fixed address in every ELF

- Because RTL modifies it, it is not write-protected
    - Attackers can write to it
        - via arbitrary-memory-write vuln
        - redirect existing function to attacker's shellcode

- learn more with objdump

.dtors Section

only with the GCC compiler. Similar to GOT, contains the destructor function pointer(s).

- constructor = .ctors
  - called before main() is invoked
- destructors = .dtors
  - both segments are writeable by default.

Dynamic Memory Management

- C Memory Management
- Common C Memory Management Errors
  - initialization errors, use-after-free, null dereffs, memory leaks, double free, ...
- Doug Lea’s Memory Allocator (next time)
- Double Free Vulnerabilities (next time)

C Memory Management (HEAP)

C99 provides 4 memory allocation functions:

- **malloc(size_t size)**: allocates **size** bytes and returns a pointer to the memory address. Memory is not zeroed / initialized
- **aligned_alloc(size_t alignment, size_t size)**: allocates **size** bytes for an object to be aligned by a specific **alignment**.
- **realloc(void *p, size_t size)**: changes the size of the memory pointed to by pointer **p** to be of **size** bytes. The contents up to that point will be unchanged. The remainder is attempted to be freed, in which case if is reused without initialization / zeroing may have the old values in place.
- **calloc(size_t nmemb, size_t size)**: allocates memory for an array of **nmemb** elements of **size** bytes each and returnsa pointer to the allocated memory. **Note that memory is set to 0**

wat is alignment?

- originally a processor design requirement.
- Back in the 90’s, On most early unix systems, an attempt to use misaligned data resulted in a bus error, which terminated the program
- modern intel (and probably ARM and others) supports the use of misaligned data, it just impacts performance

wat is alignment?

Imagine memory organized (64 bit) like so:

- objects lie in neatly aligned byte slots
- (lie on a multiple of the object's size_t value)

byte 0	byte 1	byte 2	byte 3				byte 7
QWORD A							
DWORD B				DWORD C			
QWORD D							
BYTE E	?						

Another dynamic memory function

alloca() uses the stack for dynamic memory allocation

- not C99 or POSIX
- but still found in BSD, GCC, and many linux distros
- can stack overflow...

Common C Memory Management Errors

- Initialization Errors
- Failure to check return values
- Dereferencing a NULL pointer
- Using Freed memory
- Multiple frees on same memory
- Memory Leaks

Initialization Errors

- Failure to initialize
- Falsely assuming malloc zeros memory for you
- Don’t assume free() zero’s either

Failure to check return values

Memory is limited and can be exhausted

- Programmer failure to check return code of malloc, calloc, …
  - return NULL pointers upon failure
- Using null pointer without checking is bad...

Using Freed memory

It is possible to access free'd memory unless ALL pointers to that memory have been set to NULL or invalidated.

Example (from [1] on page 156):

```
for(p = head; p != NULL; p = p->next)
    free(p);
```

Using Freed memory

Example (from [1] on page 156):

```
for(p = head; p != NULL; p = p->next)
   free(p);
```

This dereferences p after the first free(p)

```
   free(p);
   p = p->next (in the loop)
```

Using Freed memory

Safer way to do this example:

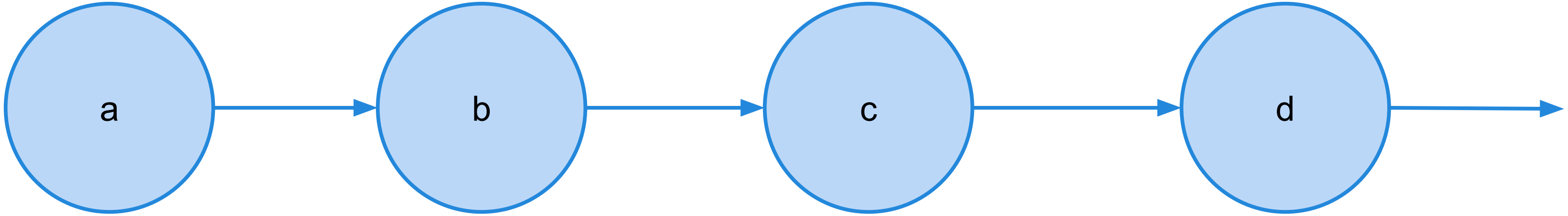
```
for (p = head; p != NULL; p = q) {
   q = p->next;
   free(p);
}
```

So after the first free(p), it no longer dereferences p:

```
free(p);
p = q;
q = p-> next;
...
```

Multiple frees on same memory

Last example tried to free up a linked list:



Not the same as this bug case

Multiple frees on same memory

```
x = malloc(n * sizeof(int));
    /* lots of code with accessing x */
    /* … */
free(x);
y = malloc(n * sizeof(int));
    /* lots of similar (pasted)code with accessing y */
    /* … */
free(x);
return;  // example from [1] p157
```

Multiple frees on same memory

Common causes:

- cut and paste errors
- sloppy error handling

Result:

- can corrupt heap memory manager
- crash / memory corruption (vulnerability)
- memory leakage

Memory Leaks

- Failure to free dynamically allocated memory after finished using it.
  - leads to memory exhaustion
    - Can be a DoS vulnerability

Memory Allocator

The memory manager on most systems runs as part of the process

- linker adds in code to do this
    - usually provided to linker via OS
        - OS's have default memory managers
            - compilers can override or provide alternatives
- Can be statically linked in or dynamically

Memory Allocator

In general requires:

- A maintained list of free, available memory
- algorithm to allocate a contiguous chunk of n bytes
  - Best fit method
    - chunk of size m >= n such that m is smallest available
  - First fit method
- algorithm to deallocate said chunks (free)
  - return chunk to list, consolidate adjacent used ones.

Memory Allocator

Common optimizations:

- Chunk boundary tags
    - [tag][------------chunk ---------][tag]
        - tag contains metadata:
            - size of chunk
            - next chunk
            - previous chunk (like a linked list sometimes)

Conclusion Mitigations

Pointers:

- _FORTIFY_SOURCE
  - stack canaries
- W^X / NX (More on this later on)
- Encoding / Decoding Function pointers

Conclusion Mitigations

String models (From CERT C Secure Coding Standard, by Robert C. Seacord 2008):

1. Caller Allocates; Caller Frees (C99/OpenBSD/C11)
2. Callee Allocates; Caller Frees (ISO/IEC TR 24731-2)
3. Callee Allocates, Callee Frees (C++ std)

Conclusion Mitigations

Dynamic Memory:

- NULL-ify pointers after free-ing them. free() does not set the pointer to NULL
- ASLR (more on this later)
- Testing testing testing
  - There are tools:
    - valgrind, insure++, Microsoft's Application Verifier, IBM's Purify

Questions?

Reading: 0x260 up to 0x280 (HAOE)