Lec 03: Secure Coding

CSED415: Computer Security

Spring 2025

Seulbae Kim

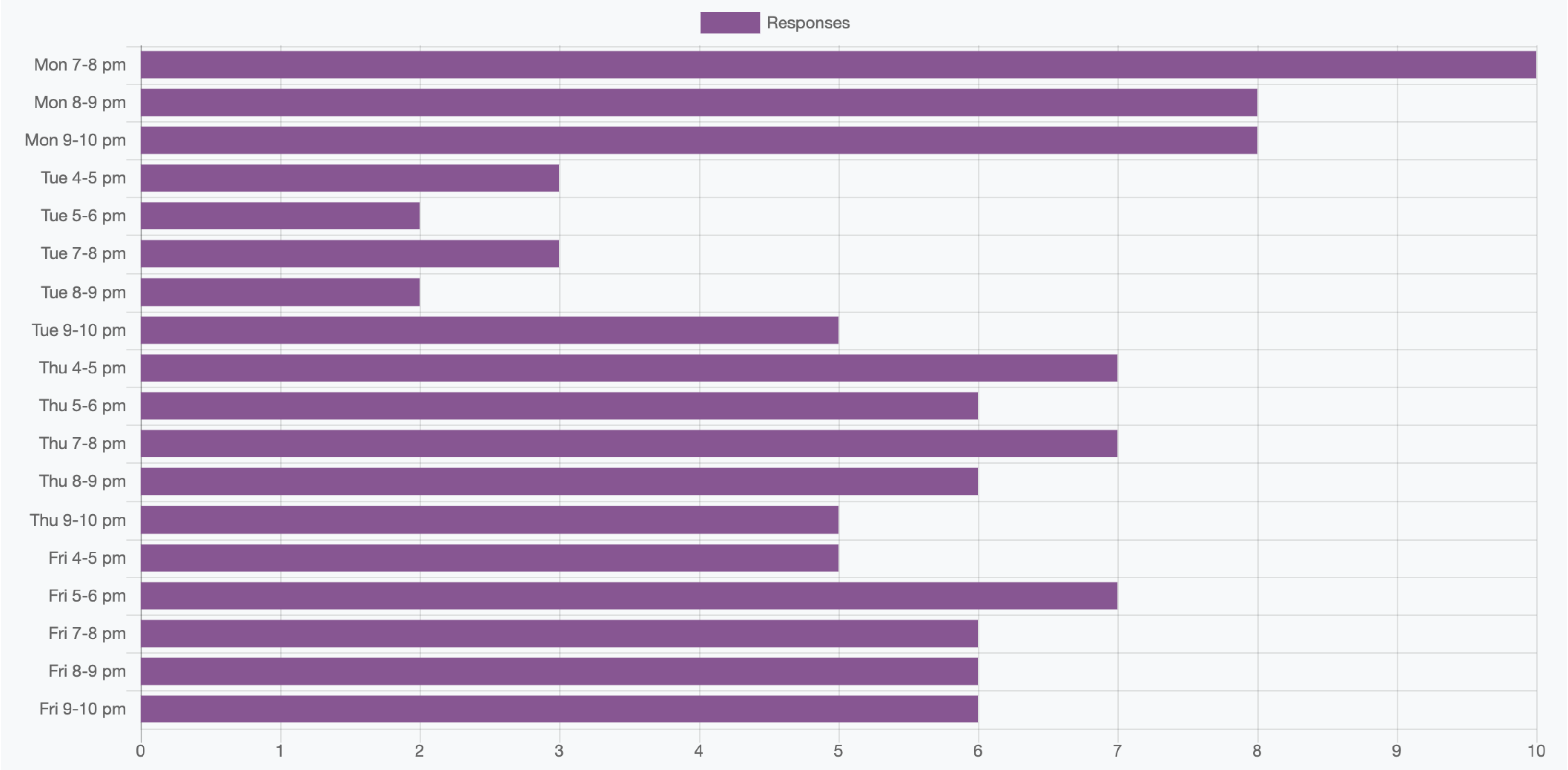
POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Reminder of important events

- Week 1-2: Lab 1 // now
- Week 3: Team forming
- Week 3-4: Lab 2
- Week 5-6: Lab 3
- Week 7: Project proposal
- Week 8: Midterm exam
- Week 9-10: Lab 4
- Week 13-14: Lab 5
- Week 15: Project presentation
- Week 16: Final exam

Office hours

- Poll result (as of Feb 25)
  - Poll closes at the end of today's course. Please cast your vote!



Recap

POSTECH

- Week 1: Basic principles and concepts
  - Understanding computer security and its challenges
  - CIA+AA: The pillars of a secure system
    - Confidentiality, Integrity, Availability + Authenticity and Accountability
  - Analyzing security through threat modeling
  - 13 fundamental principles for secure design
    - You should be able to articulate each principle and provide examples

This week: Practical cybersecurity

- Secure coding
  - How do we implement (in)secure systems?
  - How to write (in)secure code?
- Trusting trust
  - Is secure coding alone enough for computer security?

Lab 01: Warm-up

Lab 01: Warm-up

- Topic: Secure coding and virtual memory
- Due date: 11:59 PM, Friday, Feb 28, 2025
- Instructions (must read carefully):
  - https://postech-compsec.notion.site/Lab01-readme-sp25-19e18339579780b5b0a2fdfcd2152e55

Lab 01: Warm-up

- Connecting to the lab server

```
$ ssh csed415-lab01@141.223.181.16 -p 7022
Password: <check assignment page on PLMS>
```

- Running the target binary

```
$ ./target
Let's get warmed up! Invoke print_flag() to capture your flag.

------- Current table entries -------
Addr: 0x564f9ed27040 -> table[0]: 29
...
[+] Enter the index to modify:
```

Lab 01: Warm-up

- Examining the source code

```
$ cat target.c
$ vim target.c
$ nano target.c
```

- Goal: Obtain a flag from the binary

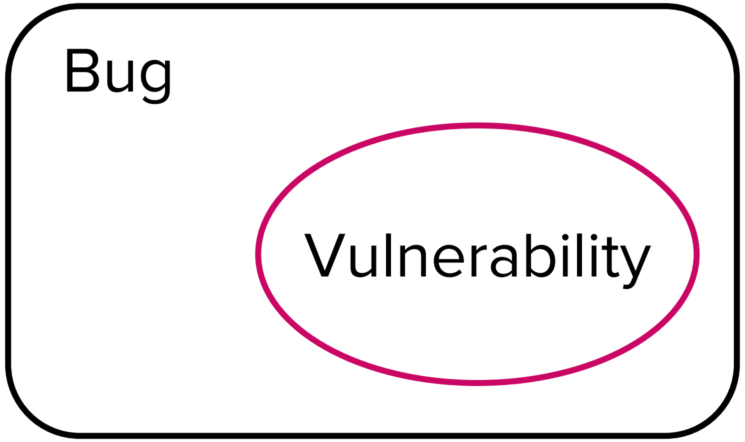
```
$ ./target
Let's get warmed up! Invoke print_flag() to capture your flag.
...
Great job! :)
This is your flag:

F38FE749E36CD0437AD0062D6836C393618FCC4A412FDE666C89EB41AC0814D5
B734EBE71127B13BFE569599D6521FF424CEBDF34F07A91CF70B8CB12E324283
788E593AF6DED9DC41A86AB64EA9A7D9A2EF718EB8B6ACFEF7A195EF241649E1
```

Defensive Programming

Recall: Bug vs Vulnerability

- Some bugs can be exploited by an attacker to compromise the entire system
  - Such exploitable of bugs are called "vulnerability" (ref: *Lecture 02*)



Key idea: If code is bug-free, then the program is vulnerability-free

Defensive programming

- Definition:
  - Process of designing and implementing a bug-free software so it remains functional even under attack
- Characteristics of such software:
  - Able to detect erroneous conditions resulting from attacks
  - Able to continue executing safely or terminate gracefully
- Secure coding is a type of defensive programming, primarily focused on enhancing computer security

Example of secure coding: Using secure APIs

- Recall C programming 101: `strcpy( )` is insecure

```c
void func(char* input) {
  char buf[8];
  strcpy(buf, input);
  /* ... */
}
```

Potential buffer overflow! Insecure :(

vs

```c
void func(char* input) {
  char buf[8];
  strncpy(buf, input, 8);
  /* ... */
}
```

Copies up to 8 bytes (size of `buf`). Secure! :)

Really??

Memory view (Intel 64-bit architecture)

- Memory is a key-value storage, initialized with garbage data
  - Key: Address
  - Value: Memory contents

<Address>	<Value>							
0x7fffffffe2e0	0x99	0x7c	0xcf	0xfd	[illegible]	[illegible]	0xb6	0x5b
0x7fffffffe2e8	0x47	0x12	0xe4	[illegible]	[illegible]	[illegible]	0x54	0xb3
0x7fffffffe2f0	0x92	0x2e	[illegible]	[illegible]	[illegible]	0xb2	0xa4	0xf8
0x7fffffffe2f8	0x72	[illegible]	[illegible]	[illegible]	0x54	0x41	0x5d	0x75
0x7fffffffe300	0x50	[illegible]	[illegible]	0x69	0xcc	0xcf	0xe0	0x1e
...	...							

garbage data

Memory view (Intel 64-bit architecture)

- `char  buf[8]` occupies 8 contiguous bytes (one 64-bit cell)
  - Because a char is one byte
  - Q) What is the address of `buf[2]`?

<Address>	<Value>	
0x7fffffffe2e0	0x99 0x7c 0xcf 0xfd 0x74 0x4b 0xb6 0x5b	} buf
0x7fffffffe2e8	0x47 0x12 0xe4 0x64 0x18 0x97 0x54 0xb3	
0x7fffffffe2f0	0x92 0x2e 0x7b 0x35 0x6c 0xb2 0xa4 0xf8	
0x7fffffffe2f8	0x72 0x76 0xc7 0x33 0x54 0x41 0x5d 0x75	
0x7fffffffe300	0x50 0x08 0x18 0x69 0xcc 0xcf 0xe0 0x1e	
...	...	

Memory view (Intel 64-bit architecture)

- Other data (e.g., variables) may co-exist on the memory
  - e.g., `char buf2[8] = "AAAAAAAA";`
    `long password = -1;`

<Address>	<Value>								
0x7fffffffe2e0	0x99	0x7c	0xcf	0xfd	0x74	0x4b	0xb6	0x5b	buf
0x7fffffffe2e8	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	buf2
0x7fffffffe2f0	0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff	password
0x7fffffffe2f8	0x72	0x76	0xc7	0x33	0x54	0x41	0x5d	0x75	
0x7fffffffe300	0x50	0x08	0x18	0x69	0xcc	0xcf	0xe0	0x1e	
...	...								

Buffer overflows

- `strcpy(buf, input);`
  - If input is longer than buf, data overwrites adjacent memory regions
    - e.g., `input = "aaaabbbbccccddddeeeeffffgggghhhh\0"`

`(0x61 == 'a')` `(0x62 == 'b')`

Overflow!

0x61	0x61	0x61	0x61	0x62	0x62	0x62	0x62	buf
0x63	0x63	0x63	0x63	0x64	0x64	0x64	0x64	buf2 (Overwritten)
0x65	0x65	0x65	0x65	0x66	0x66	0x66	0x66	password (Overwritten)
0x67	0x67	0x67	0x67	0x68	0x68	0x68	0x68	
0x00	0x08	0x18	0x69	0xcc	0xcf	0xe0	0x1e	
...								

Buffer overflows

- `strncpy(buf, input, 8);`
  - Copies at most 8 bytes, preventing overflow
    - e.g., `input = "aaaabbbbccccddddeeeeffffgggghhhh\0"`
  - May still be unsafe. Why?

No longer overflown

<Value>

0x61	0x61	0x61	0x61	0x62	0x62	0x62	0x62	buf
0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	buf2
0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff	password
0x72	0x76	0xc7	0x33	0x54	0x41	0x5d	0x75	
0x50	0x08	0x18	0x69	0xcc	0xcf	0xe0	0x1e	
...								

Recall: C string basics

- Strings in C are just char arrays
  - `char cnum[8] = "CSED415\0";` // need a slot for a `NULL` terminator

C	S	E	D	4	1	5	\0

- C has no semantic notion of “string length”
  - A C string at address `p` is a sequence of characters from `p` to the first `NULL` terminator
  - Without a `'\0'`, C does not know where the string ends

Memory view (Intel 64-bit architecture)

- `strncpy(buf, input, 8);`
  - If input is longer than 8 chars, **`buf`** is not **`NULL`**-terminated. puts(buf); causes it to continue reading memory, potentially leaking data or causing undefined behavior

Memory contents up to a NULL byte are printed (leaked)

0x61	0x61	0x61	0x61	0x62	0x62	0x62	0x62	buf
0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	buf2
0xff	0xff	0xff	0xff	0xff	0xff	0xff	0xff	password
0x72	0x76	0xc7	0x33	0x54	0x41	0x5d	0x75	
0x50	0x08	0x18	0x69	0xcc	0xcf	0xe0	0x1e	
...								

Test it yourself

- Code: leak.c

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

void func(char* input) {
  long password = -1;
  char buf2[8] = "AAAAAAAA";
  char buf[8];
  strncpy(buf, input, 8);
  puts(buf); // what does this print?
}

int main(int argc, char* argv[]) {
  func("aaaabbbbccccddddeeeeffffgggghhhh");
  return 0;
}
```

- Compilation

```
$ gcc -fno-stack-protector test.c
```

- Execution

```
$ ./a.out
aaaabbbbAAAAAAAA□□□□□□□□ □□□
  input    buf2    password  and more

$ ./a.out | hd
(check hexdump of the output)
```

Fully securing the code

- NULL-terminate your C strings whenever you use `strncpy`!

```c
void func(char* input) {
  char buf[8];
  strcpy(buf, input);
  /* ... */
}
```

Unsafe

```c
void func(char* input) {
  char buf[8];
  strncpy(buf, input, 8);
  /* ... */
}
```

Still unsafe

```c
void func(char* input) {
  char buf[8];
  strncpy(buf, input, 8);
  buf[7] = 0; // Beware: not buf[8]
  /* ... */
}
```

Safe now with an explicit NULL-termination

Secure Coding Guidelines

SEI CERT C coding standard

- https://wiki.sei.cmu.edu/confluence/display/c
  - Documented by CMU
  - Conformance is necessary (but not sufficient) for reliable and secure software
  - There are many rules; no need to memorize all of them, but it is highly recommended to read through at least once

We will look at a few examples

Rule #1: Declare objects with appropriate storage duration

- Lifetime:
  - The <u>lifetime</u> of an object is the portion of program execution during which storage is guaranteed to be reserved for it
  - An object exists, has a constant address, and retains its last-stored value throughout its lifetime
  - If an object is referred to outside of its lifetime, the behavior is undefined
    - e.g., The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime

Example #1-1: Noncompliant

- Lifetime mismatch

```
const char *p; // Static (global) variable

void dont_do_this(void) {
  const char c_str[] = "This will change";    // Automatic (local) variable
  p = c_str; // Lifetime mismatch!
}

void innocuous(void) {
  printf("%s\n", p); // Takes on an indeterminate (i.e., random) value
}
```

Example #1-1: Compliant solutions

- Solution 1: Declaring variables with the same storage duration

```
void this_is_ok(void) {
  const char c_str[] = "Everything OK";
  const char *p = c_str; // same storage duration
}
/* p is inaccessible outside its scope */
```

- Solution 2: Explicitly clearing p before `c_str` is destroyed

```
const char *p;
void this_is_ok(void) {
  const char c_str[] = "Everything OK";
  p = c_str;
  /* ... */
  p = NULL;
}
```

Example #1-2: Noncompliant

- Returning an EOL object

```
char *init_array(void) {
  char array[10]; // Automatic storage duration
  /* initialize array */
  return array; // array is destroyed
}

int main(void) {
  char *array = init_array();
  /* Caller accesses a destroyed array */

  return 0;
}
```

Example #1-2: Compliant solution

- Solution: Keep objects under the same scope

```c
#include <stddef.h>

void init_array(char *array, size_t len) {
  /* initialize array */
  return;
}

int main(void) {
  char array[10];
  init_array(array, sizeof(array));
  /* safely access array here */

  return 0;
}
```

Rule #2: Do not call `system( )`

- `system("cmd");` executes a command through a shell
- `system("cmd");` can be exploited if
  - `cmd` is not sanitized or improperly sanitized
  - `cmd` is specified without a path name
  - Relative path used in `cmd` is modified by an attacker
  - Executable program specified by `cmd` is spoofed by an attacker

Example #2-1: Noncompliant

- system is invoked without sanitizing command parameter

```c
void run_ls(const char *usr_input) {
  char cmd[512];
  snprintf(cmd, 512, "ls %s", usr_input);
  system(cmd);
}
```

- If `usr_input="/tmp"`, then `system("ls /tmp");` is executed
  - Expected behavior
- If `usr_input="/tmp; useradd hacker"`, then two cmds are invoked
  - `ls /tmp`
  - `useradd hacker`

[Note] ; is a command separator

Rule #3: Do not read uninitialized memory

- If an object that has automatic storage duration is not initialized explicitly, its value is undefined (i.e., garbage)
  - Local, automatic variables are stored on the stack
  - Their initial values default to the current values of the stack

- Dynamic allocators have different behaviors
  - `calloc( )`: Zero-initializes allocated memory
  - `malloc( )`: Does not initialize allocated memory
  - `realloc( )`: Copies contents from original pointer.
    It may not initialize all associated memory

Example #3-1: Noncompliant

- A buggy program leading to undefined behavior

```c
void set_flag(int number, int *sign) {
  if (number > 0) {
    *sign = 1;
  } else if (number < 0) {
    *sign = -1;
  }
}

int is_negative(int number) {
  int sign;
  set_flag(number, &sign);
  return sign < 0;
}
```

// `sign` is never initialized if `number` is `0`
// Then, the result of `sign < 0` is undefined
(Depends on the garbage value existing on the stack)

Example #3-2: Noncompliant

- CVE-2008-0166
  - Debian Linux developers decided to use uninitialized memory for seeding a pseudo-random number generator
  - It may sound brilliant, but indeed was a terrible idea

```c
void gen_seed(void) {
  struct timeval tv;
  unsigned long junk; // uninitialized variable

  gettimeofday(&tv, NULL);
  srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
}
```

utilize random stack contents as seed

Example #3-2: Noncompliant

- CVE-2008-0166
  - Problem: Compilers may optimize your code
    - LLVM (clang) compiler optimizes out uninitialized variables from the binary

```
void gen_seed(void) {
  struct timeval tv;
  unsigned long junk; // uninitialized variable

  gettimeofday(&tv, NULL);
  srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
}
```

junk is removed by compiler, leading to reduced entropy (randomness)

Rule #4: Do not dereference NULL pointers

- A NULL pointer represents a pointer that does not refer to a valid object or memory location

```c
int *p = NULL;
/* ... */
*p = 415;
```

<Address>	<Value>							
0x7fffffffe2e0	0x99	0x7c	0xcf	0xfd	0x74	0x4b	0xb6	0x5b
0x7fffffffe2e8	0x47	0x12	0xe4	0x64	0x18	0x97	0x54	0xb3
...	0x92	0x2e	0x7b	0x35	0x6c	0xb2	0xa4	0xf8
	0x72	0x76	0xc7	0x33	0x54	0x41	0x5d	0x75
	0x50	0x08	0x18	0x69	0xcc	0xcf	0xe0	0x1e
	...							

??

Cannot write to a non-existent memory location and triggers a segmentation fault

Example #4: Noncompliant

- A vulnerability in libPNG

```
#include <png.h>
#include <string.h>
                         -1
void set_chunkdata(int length, const void *user_data) {
  png_charp chunkdata;
  chunkdata = (png_charp)malloc(length + 1);
    NULL                                     0
  /* ... */   malloc(0) returns NULL

  memcpy(chunkdata, user_data, length);
                 NULL pointer dereference!
  /* ... */
}
```

Rule #5: Ensure (un)signed int operations do not wrap

POSTECH

- Signed and unsigned integers have valid ranges
  - `int32_t` : [-2147483648 to 2147483647]
  - `uint32_t`: [0 to 4294967295]

```
int x = 2147483647; // INT_MAX
int y = 1;
printf("%d\n", x + y); // Q) What does this print?
```

// prints –2147483648 (integer wraparound)

Example #5: Noncompliant

- An uint wrap during the addition

```
void func(unsigned int a, unsigned int b) {
  unsigned int sum;
  sum = a + b;
  char *buf = malloc(sum);  // sum can become 0
  /* ... */
}
```

Example #5: Compliant solution

- Check if wrap happens before adding (un)signed integers

```
void func(unsigned int a, unsigned int b) {
  unsigned int sum;
  if (UINT_MAX - a < b) {
    /* handle error */
  } else {
    sum = a + b;
  }
  char *buf = malloc(sum);
  /* ... */
}
```

// Beware: `if (a + b > UINT_MAX)` does not work because `a + b` might already have wrapped

Rule #6: Ensure that division and remainder operations do not result in divide-by-zero errors

- Division (/) or remainder (%) by zero causes undefined behavior
  - Must always check for zero before dividing

```
int main(void) {
  int x = 1 / 0; // Floating point exception!
  return 0;
}
```

Looks like an easy rule, but complicated cases exist in practice

Example #6: Noncompliant

- CVE-2018-13097 in Linux kernel (F2FS file system)

```
// linux/fs/f2fs/f2fs.h
struct f2fs_sb_info {   // Linux file systems store FS image metadata in a superblock (sb)
  struct super_block *sb;
  struct proc_dir_entry *s_proc;
  block_t user_block_count; /* # of user blocks */
  block_t total_valid_block_count; /* # of valid blocks */
  /* ... */
};

// linux/fs/f2fs/segment.h
static inline int utilization(struct f2fs_sb_info *sbi) {
  return div_u64((u64)valid_user_blocks(sbi) * 100,
                 sbi->user_block_count);
}
```

Can be corrupted upon crash and become zero

Div-by-zero when mounting the corrupt image

Rule #7: Do not use floating-point variables as loop counters

- Computers cannot accurately represent all real numbers
  - The precision differs depending on the CPU

```python
#!/usr/bin/python3
print(0.1 + 0.2 == 0.3) # Q) result?
```

```
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
```

Example #7-1: Noncompliant

- Using a floating-point variable as a loop counter leads to undefined behavior

```c
void func(void) {
  for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
    printf("hi\n");
  } // Q) how many "hi"s?
}
```

Logically, the loop should repeat 10 times
(0.1, 0.2, 0.3, ..., 1.0)

The loop only iterates 9 times on our lab server (Intel x86_64 processor) and a M4 Macbook (Apple silicon)

Example #7-2: Noncompliant

- Another case of using a floating-point variable as a loop counter leading to undefined behavior

```c
void func(void) {
  for (float x = 100000001.0f; x <= 100000010.0f; x += 1.0f) {
    printf("hi\n");
  } // Q) how many "hi"s?
}
```

Logically, the loop should repeat 10 times
(100000001.0, 100000002.0, …, 100000010.0)

The loop never terminates because x is incremented by an amount that is too small given the precision

Rule #8: Do not access freed memory

- Pointers to deallocated memory are called "dangling pointers"
  - `free(ptr);` → `ptr` becomes a dangling pointer
- Evaluating a dangling pointer leads to undefined behavior
  - Evaluations include:
    - `*ptr` : Dereferencing the pointer
    - `ptr + 32` : Using it as an operand of arithmetic operations
    - `(char*)ptr` : Type casting it
    - `*p = *ptr` : Using it as the right-hand side of an assignment

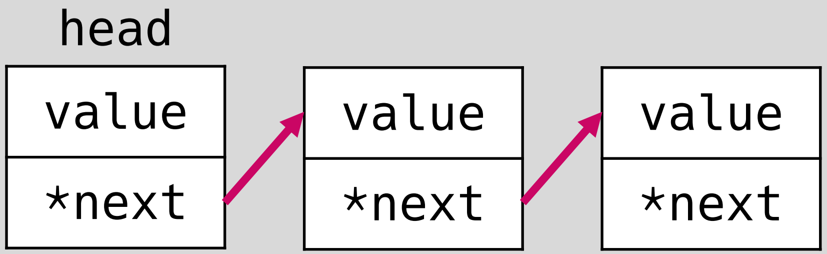
Example #8: Noncompliant

- Recall: Linked list from Data Structure

```
#include <stdlib.h>

struct node {
  int value;
  struct node* next;
};

void free_list(struct node *head) {
  for (struct node *p = head; p != NULL; p = p->next) {
    free(p);
  } // problem?
}
```



// p is freed before `p->next` is executed.
// `p->next` , i.e., `(*p).next`, dereferences a dangling pointer (p)

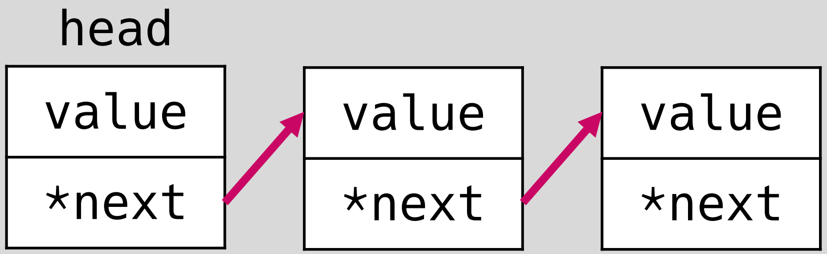
Example #8: Compliant solution

- Properly unlinking a linked list

```
#include <stdlib.h>

struct node {
  int value;
  struct node* next;
};

void free_list(struct node *head) {
  struct node *next_node;
  for (struct node *p = head; p != NULL; p = next_node) {
    next_node = p->next;   // Store p->next BEFORE p is freed
    free(p);
  }
}
```



Rule #9: Detect and remove code that has no effect or is never executed

- Dead code: Sections of code that will never be executed under any logical path during runtime
  - Problems
    - Maintenance overhead: Developers might be confused by the code that appears to serve a purpose but is never actually run
    - Unexpected compiler optimizations: Compilers may remove or reorder dead code in ways that produce surprising results or undefined behavior
    - Security risks: Dead code can be illegally executed by an attacker

Example #9-1: Noncompliant

- A function with unreachable region

```
int func(int condition) {
  char *s = NULL;
  if (condition) {
    s = (char *)malloc(10);
    if (s == NULL) {
      /* handle error */
    }
    /* process s*/
    return 0;
  }
  if (s) { /* unreachable! */
    free(s); // do clean-ups
  }
  return 0;
}
```

Example #9-1: Compliant solution

- Fix:

```
int func(int condition) {
  char *s = NULL;
  if (condition) {
    s = (char *)malloc(10);
    if (s == NULL) {
      /* handle error */
    }
    /* process s*/
    free(s);
  }
  // if (s) {
  //   free(s); // do clean-ups
  // }
  return 0;
}
```

Example #9-2: Noncompliant

- Infinite loop

```
int s_loop(char *s) {
  size_t i;
  size_t len = strlen(s); // strlen returns the number of chars preceding '\0'

  for (i = 0; i < len; ++i) {
    /* do something */

    if (s[i] == '\0') { // This condition can never be satisfied
      /* unreachable! */
      break;
    }
  } // Infinite loop causes a Denial of Service (DoS)

  return 0;
}
```

And many more...

- Refer to the guideline for more rules:
  - https://wiki.sei.cmu.edu/confluence/display/c

Coming up next: Trusting trust

- Is writing secure code sufficient to build secure systems?
  - Unfortunately, no. We will discuss why in the next lecture

Questions?