Lec 03: Secure Coding

CSED415: Computer Security

Spring 2024

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Grading (for newcomers)

- Midterm exam 25%
- Final exam 30%
- Lab assignments 25% (five labs, 5% each)
- Research project 10%
  - + additional 5% as bonus for extraordinary teams
    - Extraordinary: work that's potentially publishable at great conferences
- Participation 10%

→ Up to 105% including the bonus

Research project (for newcomers)

- We will have 4~5 teams
  - Subject to change considering the class size
- Select and work on ANY topic on computer security
  - e.g., VR/AR privacy (vision pro?) / breaking ChatGPT / crypto / …

Summary of schedule (for newcomers)

- Week 2-3: Lab 1 // now
- Week 3: Team forming
- Week 4-5: Lab 2
- Week 6-7: Lab 3
- Week 8: Midterm exam
- Week 9: Project proposal
- Week 10-11: Lab 4
- Week 12-13: Lab 5
- Week 15: Project presentation
- Week 16: Final exam

Administrivia – Lab01 is out (due March 10th)

POSTECH

- Warm-up lab with guided self-learning (check PLMS for details)
- Familiarize yourselves with the setting and tools
- Dust off your x86 assembly knowledge from CSED211
  - If you never learned machine language, this is a great opportunity to learn
- Learn basic static/dynamic analysis techniques
- Use the Q&A board on PLMS for questions
- Start early!
- Start early!
- Start early!

Recap

- Week 1 – basic principles and concepts
  - Understanding computer security and challenges
  - CIA+AA: pillars of a secure system
  - Analyzing security through threat modeling
  - 13 fundamental principles for secure design
    - You should be able to articulate each principle and give examples
    - e.g., “Tell me about the ‘open design’ principle”

This week: practical aspects

- Secure coding
  - How to implement (in)secure systems?
  - or, how to write (in)secure code?

- Trusting trust
  - Is secure coding enough for computer security?

Defensive Programming

Bug

- Some bugs can be exploited by an attacker to compromise the entire computer system
  - This class of bugs is called “vulnerability” (ref: Lec 02)

If we write a bug-free code, then the program has no vulnerability!

Defensive programming

- Process of designing and implementing a bug-free software so it continues to function even when under attack
- Such software is able to
  - Detect erroneous conditions resulting from attacks
  - Continue executing safely or fail gracefully
- Secure coding is a type of defensive programming, primarily focused on enhancing computer security

Insecure vs. secure

POSTECH

```
1 int func(char* input) {
2   char buf[8];
3   strcpy(buf, input);
4   /* ... */
5 }
```

Overflow! Insecure :(

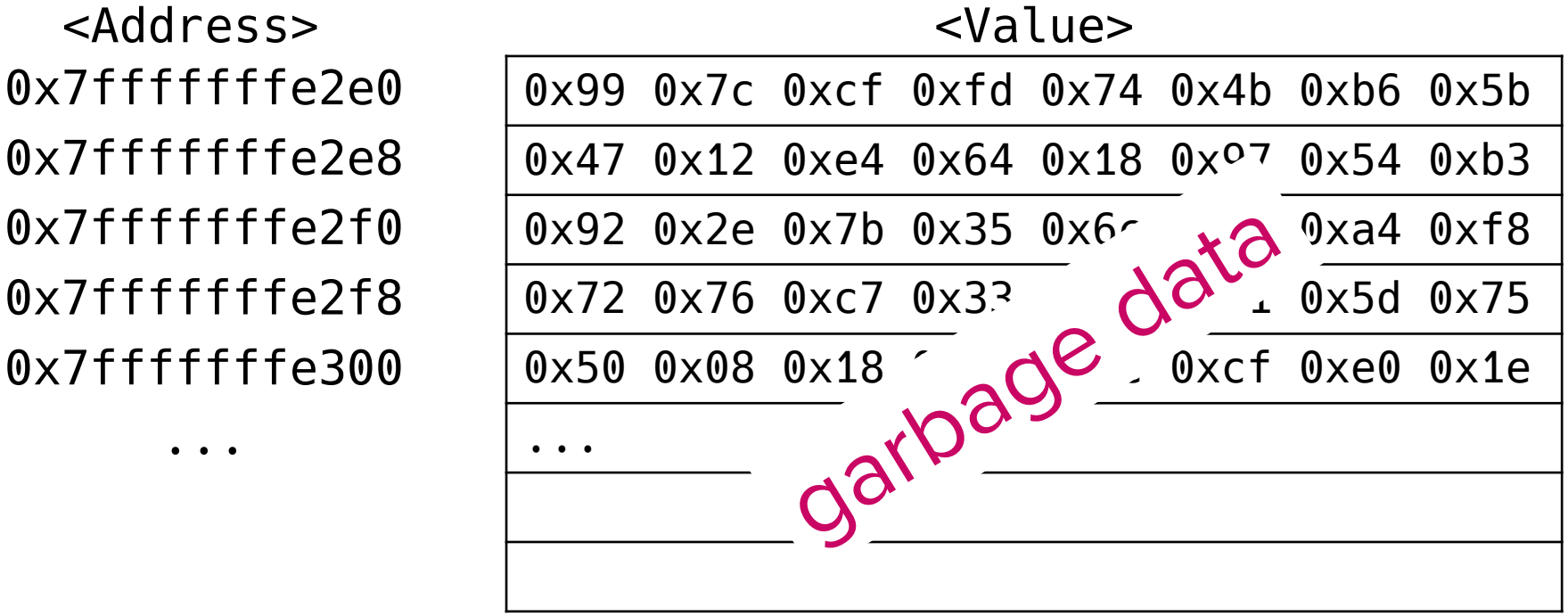
```
1 int func(char* input) {
2   char buf[8];
3   strncpy(buf, input, 8);
4   /* ... */
5 }
```

Copies up to 8 bytes. Secure! :)

Really??

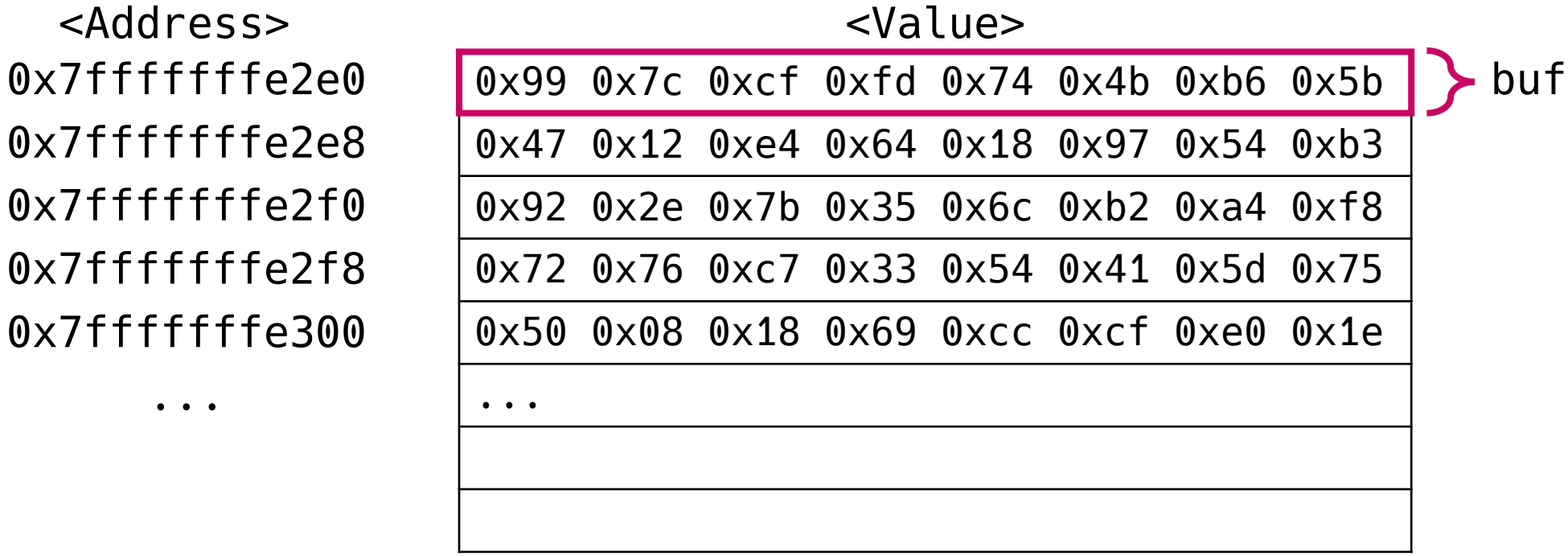
Memory view (64-bit architecture)

- Memory is a key-value storage
  - Key: address, Value: memory contents
  - One cell takes 64 bits (== 8 bytes)



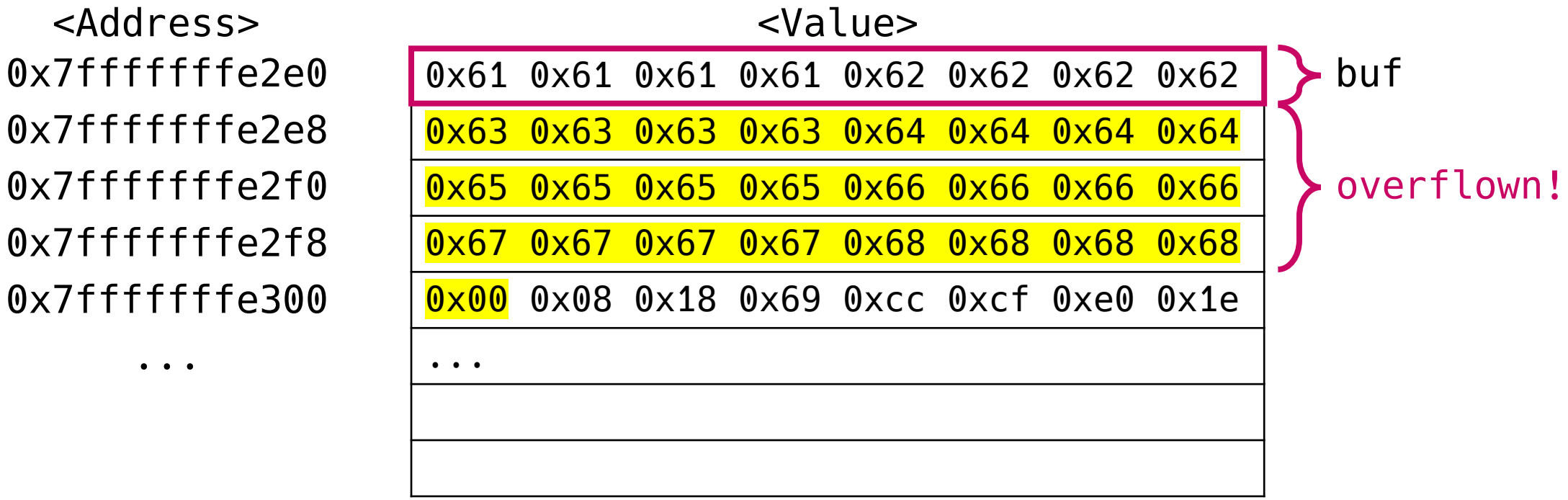
Memory view (64-bit architecture)

- `char buf[8]` takes one cell
  - Q) address of buf[2]?



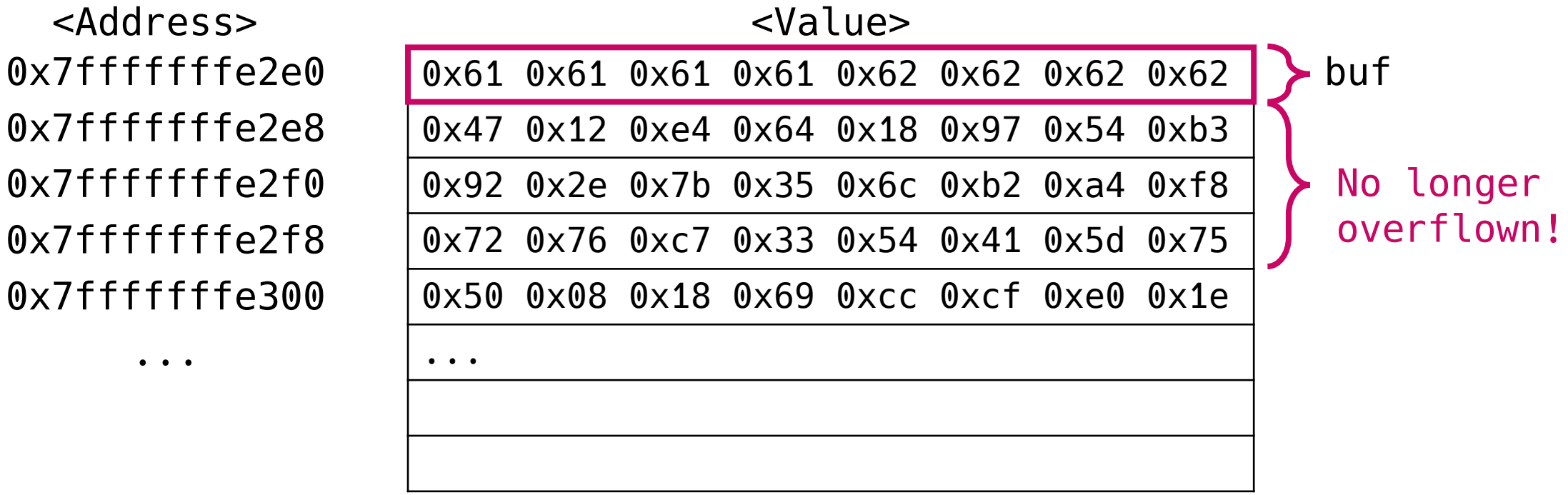
Memory view (64-bit architecture)

- `strcpy(buf, input);`
  - input = “aaaabbbbccccddddeeeeffffgggghhhh\0”
  - Q) Why is this potentially dangerous?



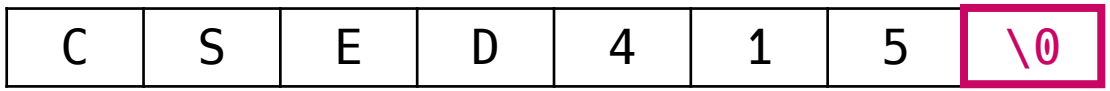
Memory view (64-bit architecture)

- `strncpy(buf, input, 8);`
  - input = “aaaabbbbccccddddeeeeffffgggghhhh\0”
  - Q) Why is this still considered unsafe?



C programming 101

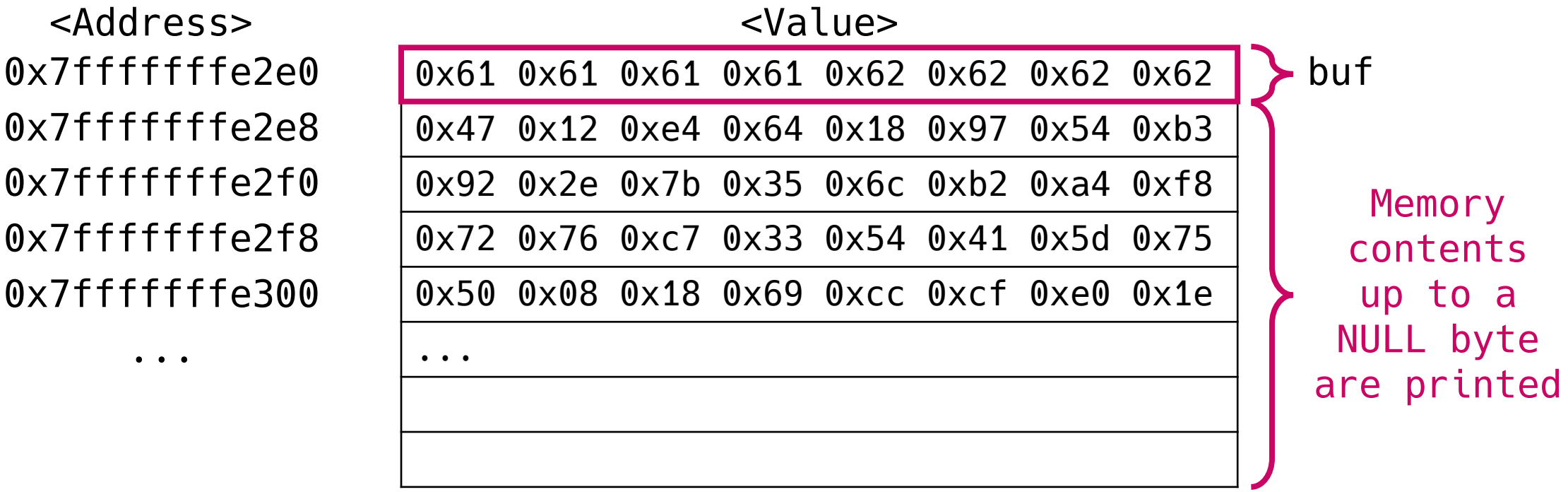
- Strings in C are char arrays
  - char cnum[8] = “CSED415”; // need a space for a **NULL** terminator



- C has no semantic notion of strings
  - A string at address **p**
    == a sequence of characters from **p** to the first **NULL** terminator
  - Without **‘\0’**, C does not know where the string ends

Memory view (64-bit architecture)

- `strncpy(buf, input, 8);`
  - e.g., `puts(buf);` // leads to an undefined behavior (UB)



Test it yourself

POSTECH

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

void func(char* input) {
  char buf[8];
  strncpy(buf, input, 8);
  puts(buf); // what does this print?
}

int main(void) {
  func("aaaabbbbccccddddeeeeffffgggghhhh");
  return 0;
}
```

One more step after strncpy

- Null-terminate the string

```
1 int func(char* input) {
2   char buf[8];
3   strcpy(buf, input);
4   /* ... */
5 }
```

Bad

```
1 int func(char* input) {
2   char buf[8];
3   strncpy(buf, input, 8);
4   /* ... */
5 }
```

Still bad

```
1 int func(char* input) {
2   char buf[8];
3   strncpy(buf, input, 8);
4   buf[7] = 0; // Beware: not buf[8]
5   /* ... */
6 }
```

Safe now

Secure Coding Guidelines

SEI CERT C coding standard

- https://wiki.sei.cmu.edu/confluence/display/c
  - Documented by CMU
  - Conformance to this standard are necessary (but not sufficient) to ensure the safety, reliability, and security of software systems developed in C
  - A lot of rules – do not need to memorize all, but recommended to read through them at least once when you have time

#1: Declare objects with appropriate storage duration

- The lifetime 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. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime

Example #1-1-noncompliant

POSTECH

```
1 const char *p; // Static variable
2 void dont_do_this(void) {
3   const char c_str[] = "This will change"; // Automatic variable
4   p = c_str; /* Dangerous */  // lifetime mismatch
5 }
6
7 void innocuous(void) {
8   printf("%s\n", p);  // out of scope → takes on an indeterminate value
9 }
```

Example #1-1-compliant

POSTECH

```c
void this_is_OK(void) {
  const char c_str[] = "Everything OK";
  const char *p = c_str;  // same storage duration
  /* ... */
}
/* p is inaccessible outside the scope of string c_str */
```

```c
const char *p;
void is_this_OK(void) {
  const char c_str[] = "Everything OK?";
  p = c_str;
  /* ... */
  p = NULL;     // p is cleared before c_str is destroyed
}
```

Example #1-2-noncompliant

POSTECH

```
1 char *init_array(void) {
2   char array[10]; // Automatic storage duration
3   /* Initialize array */
4   return array;    // The caller can access the destroyed array
5 }
```

Example #1-2-compliant

POSTECH

```c
#include <stddef.h>
void init_array(char *array, size_t len) {
  /* Initialize array */
  return;
}

int main(void) {
  char array[10]; // Keep object under same scope
  init_array(array, sizeof(array) / sizeof(array[0]));
  /* ... */
  return 0;
}
```

#2: Do not call system()

- system("cmd") executes a command by invoking 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 can be modified by an attacker
  - executable program specified by cmd can be spoofed by an attacker

Example #2-1-noncompliant


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

If input is “/home/lab01” then system(“ls /home/lab01”); is executed

If input is “/home/lab01; useradd attacker”
then two commands are invoked in sequence:
(1) ls /home/lab01 and
(2) useradd attacker

#3: Do not depend on the order of evaluation for side effects

- Evaluation of an expression may produce side effects
- Sequence points are points that distinguish evaluations during execution
  - e.g., between the evaluations of the operands of &&
- Do not depend on the order of evaluation for side effects unless there is an intervening sequence point

Example #3-noncompliant and compliant

POSTECH

```
1 extern void c(int i, int j);
2 int glob;
3
4 int a(void) {
5   return glob + 10;
6 }
7
8 int b(void) {
9   glob = 42;
10  return glob;
11 }
12
13 void func(void) {
14  c(a(), b());
15 }
```

// order of argument evaluation is unspecified

```
1 extern void c(int i, int j);
2 int glob;
3
4 int a(void) {
5   return glob + 10;
6 }
7 int b(void) {
8   glob = 42;
9   return glob;
10 }
11
12 void func(void) {
13  int a_val = a();
14  int b_val = b();
15  c(a_val, b_val);
16 }
```

#4: Do not read uninitialized memory

- If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate
  - 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 ptr; may not init. all memory

Example #4-1-noncompliant

POSTECH

```c
void set_flag(int number, int *sign_flag) {
  if (NULL == sign_flag) return;

  if (number > 0) {
    *sign_flag = 1;
  } else if (number < 0) {
    *sign_flag = -1;
  } // missing case: number == 0
}

int is_negative(int number) {
  int sign;
  set_flag(number, &sign); // sign is uninitialized
  return sign < 0; // comparison exhibits undefined behavior
}
```

Example #4-2-noncompliant

- CVE-2008-0166
  - Debian devs got creative and decided to use uninitialized memory for random number generation
  - Why is this a bad idea?

```
1 void func(void) {
2   struct timeval tv;
3   unsigned long junk;
4
5   gettimeofday(&tv, NULL);
6   srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
7 }
```

// utilize random stack contents as seed

Example #4-2-noncompliant

- CVE-2008-0166
  - Different compilers employ different optimization strategies
  - LLVM optimizes out uninitialized variables completely

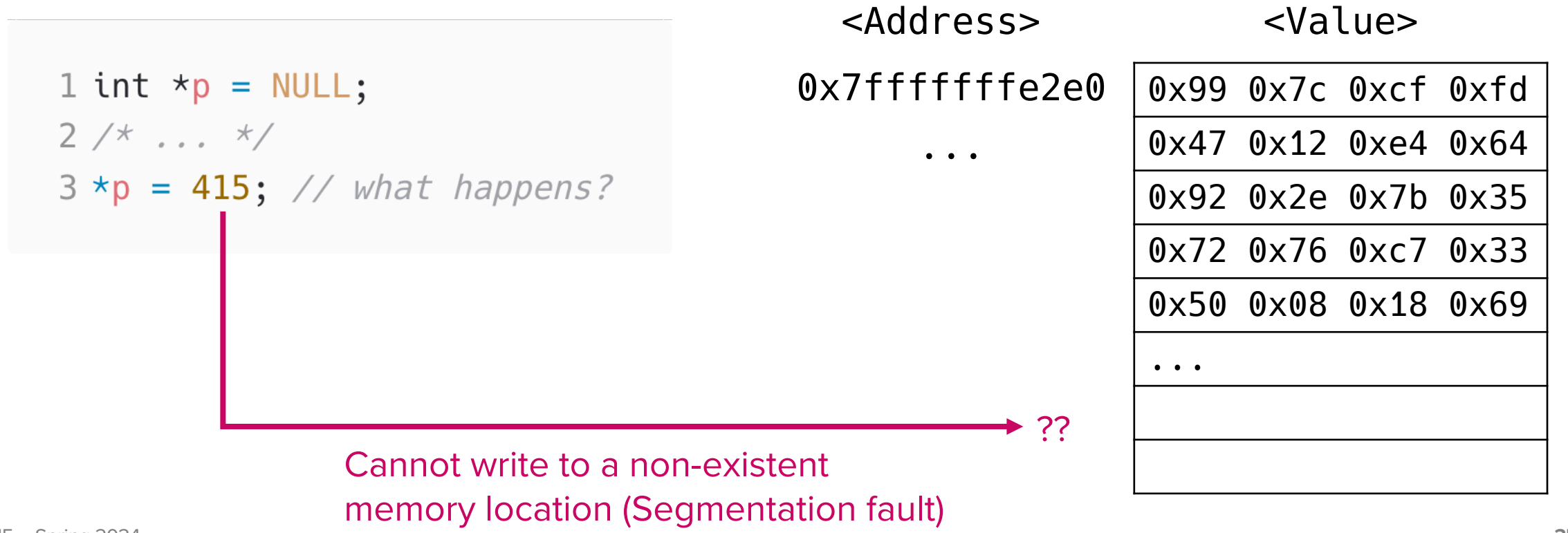
```
1 void func(void) {
2   struct timeval tv;
3   unsigned long junk;
4
5   gettimeofday(&tv, NULL);
6   srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
7 }
```

// removed by compiler

Rule #5: Do not dereference NULL pointers

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

```
1 int *p = NULL;
2 /* ... */
3 *p = 415; // what happens?
```



Example #5-noncompliant

- Q) Find the problem

```
1 #include <png.h> /* From libpng */
2 #include <string.h>
3
4 void func(png_structp png_ptr, int length, const void *user_data) {
5   png_charp chunkdata;
6   chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
7   /* ... */
8   memcpy(chunkdata, user_data, length);
9   /* ... */
10 }
```

// if length == -1

// malloc of size 0 returns NULL

// chunkdata is NULL, and gets dereferenced

Rule #6: Ensure (un)signed int operations do not wrap

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

```
1 int x = 2147483647;
2 int y = 1;
3 printf("%d\n", x + y); // result?
```

// prints -2147483648 (silent wraparound)

Example #6-noncompliant

POSTECH

```
1 void func(unsigned int a, unsigned int b) {
2   unsigned int sum;
3   sum = a + b;
4   char* buf = malloc(sum);
5   /* ... */
6 }
```

Example #6-compliant

POSTECH

```
1 void func(unsigned int a, unsigned int b) {
2   unsigned int sum;
3   if (UINT_MAX - a < b) {   // Beware: if (a + b < UINT_MAX) doesn't work
4     /* Handle error */
5   } else {
6     sum = a + b;
7   }
8   char* buf = malloc(sum);
9   /* ... */
10 }
```

Rule #7: Ensure that integer conversions do not result in lost or misinterpreted data

POSTECH

- Implicit and explicit int conversions must be guaranteed not to result in lost or misinterpreted data

```c
#include <stdio.h>

int main(void) {
  int i = 0x100;  // Uses two bytes
  printf("%d\n", i);
  char c = (char)i; /* explicit casting */
  printf("%d\n", c); // char type can hold one byte
  return 0;
}
```

Rule #8: Ensure that division and remainder operations do not result in divide-by-zero errors

POSTECH

- When the value of the second operand of the / or % operator is zero, i.e., divide-by-zero, the behavior is undefined (UB)

```
1 int main(void) {
2   int x = 1 / 0; // Floating point exception
3   return 0;
4 }
```

Complicated cases exist in practice

Example #8-noncompliant

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

```
1  // linux/fs/f2fs/f2fs.h
2  struct f2fs_sb_info {
3    struct super_block *sb;  /* pointer to VFS super block */
4    struct proc_dir_entry *s_proc;  /* proc entry */
5    /* ... */
6    block_t user_block_count;  /* # of user blocks */
7    block_t total_valid_block_count;  /* # of valid blocks */
8    /* ... */
9  }
10
11 // linux/fs/f2fs/segment.h
12 static inline int utilization(struct f2fs_sb_info *sbi)
13 {
14     return div_u64((u64)valid_user_blocks(sbi) * 100,
15                    sbi->user_block_count);
16 }
```

File system image metadata (superblock)

Corrupted due to crash (becomes zero)

Div-by-zero when mounting the corrupt image

Rule #9: Do not use floating-point variables as loop counters

POSTECH

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

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

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

Example #9-1-noncompliant

POSTECH

```
1 void func(void) {
2   for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
3     /* Loop may iterate 9 or 10 times */
4   }
5 }
```

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

The loop only iterates 9 times on an x86_64 processor
(our lab server)

Example #9-2-noncompliant


```
1 void func(void) {
2   for (float x = 100000001.0f; x <= 100000010.0f; x += 1.0f) {
3     /* Loop may not terminate */
4   }
5 }
```

`100000001.0f + 1.0f = 1.00000001.0f`

(incremented by an amount that is too small given the precision)

Rule #10: Do not form or use out-of-bounds pointers or array subscripts

POSTECH

```
1 enum { TABLESIZE = 100 };
2
3 static int table[TABLESIZE];
4
5 int *get_ptr(int index) {
6   if (index < TABLESIZE) {   // check if index if beyond array bounds
7     return table + index;    // is the check sufficient?
8   }
9   return NULL;
10 }
```

Example #10-compliant


```c
enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(int index) {
  if (index >= 0 && index < TABLESIZE) {
    return table + index;
  }
  return NULL;
}
```

do explicit check

```c
#include <stddef.h>
enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(size_t index) {
  if (index < TABLESIZE) {
    return table + index;
  }
  return NULL;
}
```

use unsigned type

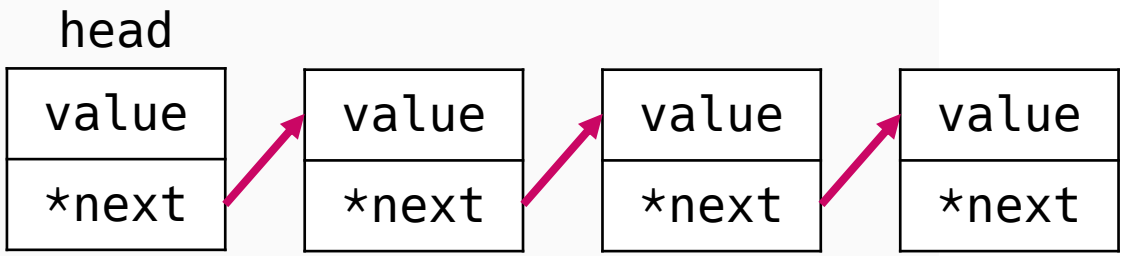
Rule #11: Do not access freed memory

- Freed memory == dangling pointer
  - free(ptr);
- Evaluating a dangling pointer is undefined behavior
  - Dereferencing it // *ptr
  - Using it as an operand // *ptr + 20
  - Type casting it // (char*)ptr
  - Using it as the right-hand side of an assignment // *p = *ptr

Example #11-noncompliant

- A famous example from Data Structure 101

```
1 #include <stdlib.h>
2
3 struct node {
4   int value;
5   struct node *next;
6 };
7
8 void free_list(struct node *head) {
9   for (struct node *p = head; p != NULL; p = p->next) {
10    free(p); // p is freed before p->next is executed.
11  }          // p->next reads freed memory
12 }
```



Example #11-compliant


```c
#include <stdlib.h>

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

void free_list(struct node *head) {
  struct node *q;
  for (struct node *p = head; p != NULL; p = q) {
    q = p->next; // Keep track of p->next before freeing p
    free(p);
  }
}
```

Rule #12: Detect and remove code that has no effect or is never executed

POSTECH

- Dead code and unreachable code can cause unexpected behavior
  - Usually optimized out during compilation by modern compilers, but not always!

- This rule is related to Lab01

Example #12-1-noncompliant

POSTECH

```
1 int func(int condition) {
2     char *s = NULL;
3     if (condition) {
4         s = (char *)malloc(10);
5         if (s == NULL) {
6             /* Handle Error */
7         }
8         /* Process s */
9         return 0; // function always returns here
10    }
11
12    if (s) {
13        /* This code is unreachable */
14    }
15    return 0;
16 }
```

Example #12-1-compliant

POSTECH

```
1 int func(int condition) {
2     char *s = NULL;
3     if (condition) {
4         s = (char *)malloc(10);
5         if (s == NULL) {
6             /* Handle Error */
7         }
8         /* Process s */
9         // return 0;   // simple fix
10    }
11
12    if (s) {
13        /* now reachable */ // do cleanup, e.g., free(s);
14    }
15    return 0;
16 }
```

Example #12-2-noncompliant

POSTECH

```
1 int s_loop(char *s) {
2     size_t i;
3     size_t len = strlen(s); // strlen returns the number of chars preceding '\0'
4     for (i = 0; i < len; ++i) {
5       /* Code that doesn't change s, i, or len */
6       if (s[i] == '\0') { // this condition can never be satisfied
7         /* This code is never reached */
8       }
9     }
10    return 0;
11 }
```

Leads to Denial of Service (DoS)

And many more…

- Do check out the guideline!
  - https://wiki.sei.cmu.edu/confluence/display/c

Open question

- Why don’t we write an analyzer that checks for all rules in the secure coding standard?

Coming up next: Trusting trust

- Is code-level analysis enough to build secure systems?

Questions?