

# Secure Coding in C and C++

## Exercise #3: Code Repair

Software Engineering Institute  
Carnegie Mellon University  
Pittsburgh, PA 15213



Software Engineering Institute

Carnegie Mellon University

© 2015 Carnegie Mellon University

# Notices

---

Copyright 2022 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material was prepared for the exclusive use of Trainees of online & offline course and may not be used for any other purpose without the written consent of [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

This material is distributed by the Software Engineering Institute (SEI) only to course attendees for their own individual study.

Except for any U.S. government purposes described herein, this material SHALL NOT be reproduced or used in any other manner without requesting formal permission from the Software Engineering Institute at [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

Although the rights granted by contract do not require course attendance to use this material for U.S. Government purposes, the SEI recommends attendance to ensure proper understanding.

This material and exercises include and/or can make use of certain third party software ("Third Party Software"), which is subject to its own license. The Third Party Software that is used by this material and exercises are dependent upon your system configuration, but typically includes the software identified in the documentation and/or ReadMe files. By using this material and exercises, You agree to comply with any and all relevant Third Party Software terms and conditions contained in any such Third Party Software or separate license file distributed with such Third Party Software. The parties who own the Third Party Software ("Third Party Licensors") are intended third party beneficiaries to this License with respect to the terms applicable to their Third Party Software. Third Party Software licenses only apply to the Third Party Software and not any other portion of the material as a whole.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

CERT® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM22-0263

# Planning

---

What qualities are we trying to achieve?

- security
- time performance
- memory performance
- robustness
- usability

The answers to these questions determine how we proceed.

# Tradeoffs

---

Should we use static or dynamic allocated memory?

## Statically allocated memory

- improves time performance
- might waste memory
- might limit the size of unexpectedly large inputs

## Dynamically allocated memory

- uses only as much memory as you need
- accommodates unexpectedly large inputs
- risks exhausting memory

# Exercise

---

Fix your code  
(45 minutes)



# Getting the File Name

---

Re-declare `size` as `size_t` and initialize

```
size_t size = 0;
```

Re-implement using dynamically allocated memory:

```
dplen = strlen(file);  
size = dplen + strlen(argv[1]) + 2;  
full_path = malloc(size);  
strcpy(full_path, file);  
if (full_path[dplen-1] != '/') {  
    full_path = strcat(full_path, "/");  
}  
full_path = strcat(full_path, argv[1]);  
printf("path:%s", full_path);
```

# Still Problematic

---

Possible integer overflow

```
size = dplen + strlen(argv[1]) + 2;
```

Test for possibility of overflow:

```
dplen = strlen(file);  
fnlen = strlen(argv[1]);  
if (dplen > SIZE_MAX - fnlen) {  
    /* handle error */  
}  
else size = dplen + fnlen;  
if (size > SIZE_MAX - 2) {  
    /* handle error */  
}  
else size += 2;
```

# More Integer Problems

---

Multiplication of an untrusted value (size), which is then used in a memory allocation

Replace

```
sigdb = malloc(  
    size * sizeof(struct sigrecord)  
);
```

with

```
if ( size > SIZE_MAX/sizeof(struct sigrecord) ) {  
    fprintf(stderr, "integer overflow.\n");  
    goto free_path;  
}  
sigdb = malloc(size * sizeof(struct sigrecord));
```



# The `fgetc()` Function

---

Do not convert the value returned by a character input/output function to `char` if that value is going to be compared to the EOF character.

Change

```
char c;  
while ((c = (char)fgetc(in)) != EOF) {
```

to

```
int c;  
while ((c = fgetc(in)) != EOF) {
```

# Negative Indices

---

This can be simply fixed by declaring `idx` as unsigned:

```
size_t idx = atoi(input);  
if (idx < size) {  
    printf(  
        "%d %s %s\n",  
        sigdb[idx].signum,  
        sigdb[idx].signame,  
        sigdb[idx].sigdesc  
    );  
}
```

# The `fscanf()` Function

---

The `fscanf()` function returns the value of the macro `EOF` if an input failure occurs before any conversion.

Otherwise, the function returns the number of input items assigned, which can be fewer than provided for or zero in the event of an early matching failure.

Replace

```
fscanf(in, "%i", &size);
```

with

```
if (1 != fscanf(in, "%i", &size)) {  
    /* handle error */  
}
```

to make sure we read in a size.

# The `atoi()` Function

---

`fscanf(in, "%i", &size)` indicates an error by returning a negative value.

But `atoi()` and related functions lack a mechanism for reporting errors for invalid values.

Both functions will silently truncate input if the input cannot be represented as a **signed int**.

Consequently, we need to replace the following function call

```
idx = atoi(input);
```

with a call to the `strtoul()` function

# Use the `strtoul()` Function

---

```
unsigned long ul;
char *end_ptr;

errno = 0;
ul = strtoul(input, &end_ptr, 0);

if (ERANGE == errno) {
    fputs("number out of range\n", stderr);
}
else if (ul > size) {
    fprintf(stderr, "value out of range\n");
}
else if (end_ptr == input) {
    fputs("invalid numeric input\n", stderr);
}
else {
    idx = (unsigned int)ul;
    printf("%lu %s %s\n", sigdb[idx].signum, sigdb[idx].signame,
sigdb[idx].sigdesc);
}
```

# Use the `strtoumax()` Function

---

The `strtoimax()` and `strtoumax()` functions are equivalent to the `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions, except that the initial portion of the string is converted to `intmax_t` and `uintmax_t` representation, respectively.

For more information, see

[ERR34-C. Detect errors when converting a string to a number](#)

# Use the `strtoumax()` Function

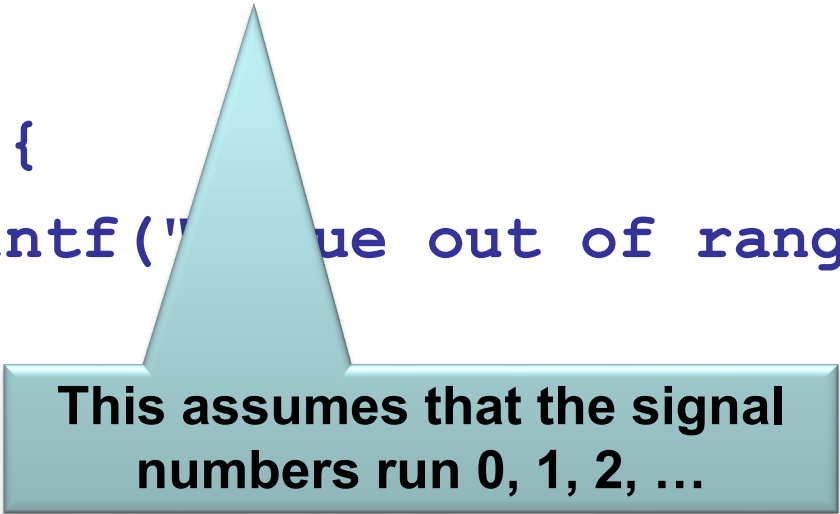
---

```
uintmax_t idx;  
char *end_ptr;  
  
errno = 0;  
idx = strtoumax(input, &end_ptr, 0);  
  
if (ERANGE == errno) {  
    fputs("number out of range\n", stderr);  
}  
else if (idx > SIZE_MAX) {  
    fprintf(stderr, "value out of range\n");  
}  
/* ... */
```

# The Wrong Signal

---

```
idx = atoi(input);  
if (idx < size) {  
    printf("%d %s %s\n", sigdb[idx].signum,  
           sigdb[idx].signame, sigdb[idx].sigdesc);  
}  
else {  
    printf("Value out of range.\n");  
}
```



**This assumes that the signal  
numbers run 0, 1, 2, ...**



# The Right Signal

---

```
idx = atoi(input);
bool found = false;
for (size_t j = 0; j < size; j++) {
    if (sigdb[j].signum == idx) {
        printf("%d %s %s\n", sigdb[idx].signum,
            sigdb[idx].signame, sigdb[idx].sigdesc);
        found = true;
        break;
    }
}
if (!found) {
    printf("Value out of range.\n");
}
```

# Bonus: The `getenv()` function

---

If `DATA_PATH` cannot be found, a null pointer is returned.

Change

```
file = getenv("DATA_PATH");  
if (file != '\0') {
```

to

```
file = getenv("DATA_PATH");  
if (file != NULL) {
```

The code works the same but is better style.

# Questions

