

CSE 443: Introduction to Computer Security

Module: Software Security

Safe Programming

Prof. Syed Rafiul Hussain
Department of Computer Science and Engineering
The Pennsylvania State University

Acknowledgements: Some of the slides have been adopted from Trent Jaeger (Penn State)

Avoiding Vulnerabilities

- How do we write programs to avoid mistakes that lead to vulnerabilities?
 - ▶ Prevent memory errors
 - ▶ Detect data handling errors (e.g., truncation)



Processing String Input

- Major cause of buffer overflows and other memory errors is the processing of **string input**
 - ▶ Read input into your program
 - read/fread, gets, scanf, and variants
 - ▶ Manipulate string data
 - strcpy, strcat, and variants
 - ▶ Comparing and converting strings
 - strtok, strcmp, strtol, and variants
- What **properties would you like to ensure** when you read and manipulate strings to prevent memory errors?

Processing String Input

- Major cause of buffer overflows and other memory errors is the processing of **string input**
- What **properties would you like to ensure** when you read and manipulate strings to prevent memory errors?
 - ▶ Should create a buffer containing a string that is within buffer bounds and is null terminated
 - That is, should be a **semantically correct C string**
 - ▶ But, how to check for these properties, how to detect failures, and what to do on failure?
- Many C functions for string processing work slightly differently

- See David Wheeler's "Secure Programming HOWTO" documentation and slides
 - ▶ Detailed guidance on which C library functions to use and which to avoid
 - ▶ And the future of such C library functions
 - Particularly for string processing
- Following slides are derived from his documentation and slides

No Bounds Checking

- Many C library functions do not check bounds
 - ▶ Don't use these functions
- Functions
 - ▶ **gets** – reads input without checking.
 - ▶ **strcpy** – `strcpy(dest, src)` copies from src to dest
 - If src longer than dest buffer, keeps writing!
 - ▶ **strcat** – `strcat(dest, src)` appends src to dest
 - If `strlen(src)+strlen(dest)` longer than buffer, keeps writing!
 - ▶ **scanf** family of input functions
 - Many options don't control max length (e.g., bare “%s”)

No Bounds Checking

- Many C library functions do not check bounds
 - ▶ Don't use these functions
- Example: `scanf`
 - ▶ `scanf(input, "%s", target);`
 - ▶ Moves input to target until null termination of "input"
 - ▶ Regardless of length of buffer allocated for "target"
- Such functions (used this way) are **inherently unsafe** if they receive adversary-controlled input

No Guarantee of Null Term.

- Even functions that provide some degree of bounds checking may **fail to guarantee null termination** of input
- Consider `strncpy`
- `char *strncpy(char *DST, const char *SRC, size_t LENGTH)`
 - ▶ Copy string of bytes from SRC to DST
 - ▶ Up to LENGTH bytes; if less, NIL-fills
 - ▶ **Scenario**: Suppose size of buffer DST is LENGTH and size of SRC is also LENGTH
 - ▶ then fills buffer DST **without null terminator**
- In that case, what happens for `strlen(DST)`?

Two Main Defense Options

- (1) Bounds check or (2) auto-resize buffer
 - ▶ Include null-termination
- **Bounds checking**
 - ▶ If reach bound
 - (a) Stop processing
 - (b) Truncate data
 - ▶ Stop processing can be **used for DoS attacks**
 - ▶ Truncation can lose valuable data or allow adversary to remove data chosen by adversary
 - E.g., in middle of multibyte (unicode) character
 - **Ideally, we want notification if inputs is truncated**

Two Main Defense Options

- (1) Bounds check or (2) auto-resize buffer
 - ▶ Include null-termination
- Auto-resize
 - ▶ If reach bound
 - (a) Create new buffer of desired size
 - ▶ This is what most other programming languages do
 - ▶ Auto-resize can present some challenges in C/C++ due to manual memory management
 - E.g., When to free a buffer that is no longer large enough to use?
 - Code gets a bit more complex

Bounds Checking Solutions

- Traditional: `strncat`, `strncpy`, `sprintf`, `snprintf`
 - ▶ First three are hard to use correctly
- `strncat/strncpy`
 - ▶ Lack of guarantee of null termination
 - ▶ No report of truncation, should it occur
 - ▶ Also, `strncpy` does dumb things like NULL-fills rest of buffer, incurring often unnecessary overhead



Bounds Checking Solutions

- Traditional: `strncat`, `strncpy`, `sprintf`, `snprintf`
 - ▶ First three are hard to use correctly
- `sprintf`
 - ▶ Use format string to express bounds checks
 - ▶ `%.10s` means “ ≤ 10 bytes” (notice “.”)
 - `%10s` sets minimum (!) length
 - ▶ Or can use `*` to pass bounds value as an argument
 - `sprintf(dest, "%. *s", maxlen, src);`
 - `maxlen` holds the maximum bytes to copy (still need “.”)
 - ▶ Does not appear to ensure null termination
 - ▶ Or inform on truncation
- Hard to use all these things correctly

Bounds Checking Solutions



- Traditional: `strncat`, `strncpy`, `sprintf`, `snprintf`
 - ▶ First three are hard to use correctly
- `snprintf`
 - ▶ **`int snprintf(char *s, size_t n, const char *format, ...);`**
 - ▶ Writes output to buffer “s” up to *n* chars (**bounds check**)
 - ▶ Always writes `\0` at end if $n \geq 1$ (**null termination**)
 - ▶ Returns “length that would have been written” or negative if error (**enable checking for truncation or errors**)
- Example
 - ▶ `len = snprintf(buf, buflen, "%s", original_value);`
 - ▶ `if (len < 0 || len >= buflen) ... // handle error/truncation`

Bounds Checking Solutions



- What if you want to bounds check, null-terminate string, detect truncation, and ...
 - ▶ **limit the number of bytes read?**
 - ▶ `snprintf` reads to end of input string normally
- **Can use `snprintf` with precision specifier**
 - ▶ `len = snprintf(dest, destsize, "%.s", (int) srcsize, src)`
 - ▶ `if (len < 0 || len >= buflen) ... // handle error/truncation`
 - ▶ Can be a bit quirky though
 - ▶ Need the “(int)”

Bounds Checking Solutions

- **Future:** more streamlined bounds checking solutions
- **strcpy and strcat**
 - ▶ Simpler, full-featured bounds checking
 - ▶ Always null-terminates, if dest has any space (have to leave room, but can specify)
 - ▶ strcpy doesn't null-fill, unlike strncpy (good!)
 - ▶ Easy to detect if terminates "in the middle"
 - ▶ Returns "bytes would have written" like sprintf

- Versions of printf that **support auto-resize**
- `asprintf` and `vasprintf`
 - ▶ analogous to `sprintf` and `vsprintf`, but auto-allocate a new string
- Simple to use and do not terminate results in middle because it resizes the string buffer
- Example
 - ▶ `char *result;`
 - ▶ `asprintf(&result, "x=%s and y=%s\n", x, y);`
 - ▶ Allocate memory for “result” based on size of resulting (no pun intended) string
 - ▶ You will have to free that yourself

- Resizing is also supported for other unsafe functions to avoid memory errors
- scanf family of functions
- Use the “%m” qualifier to allocate buffer dynamically to hold the input
- Example
 - ▶ `char *result;`
 - ▶ `sscanf(input, "%ms", &result);`
- Again, you must free the auto-allocated memory
 - ▶ Only if the `sscanf` was successful

- Resizing is also supported for other unsafe functions to avoid memory errors
- `getline` function
- Works in a manner analogous to `scanf` family
 - ▶ No qualifier necessary though
- Example
 - ▶ `FILE *stream;`
 - ▶ `char *line = NULL;`
 - ▶ `size_t len = 0;`
 - ▶ `while ((nread = getline(&line, &len, stream)) != -1) {`
 - `/* operate on "line" */`
- Will only auto-allocate when more space is needed

- Lots of memory errors occur due to sloppy string handling
- Even if you think you are doing the right thing (e.g., `strncpy` and `strncat`), you are prone to flaws
 - ▶ Due to truncation and/or lack of null-termination
- No reason to fall victim to simple errors
 - ▶ Although still have to **compute bounds correctly** for checking in some cases
- Should start using safe string handling functions **NOW**
- Also, use “**assert**” for error checking

- Programs have function
 - ▶ Adversaries can exploit unexpected functions
- Vulnerabilities due to malicious input
 - ▶ Subvert control-flow or critical data
 - Buffer, heap, integer overflows, format string vulnerabilities
 - ▶ Injection attacks
 - Application-dependent
- If applicable, write programs in languages that eliminate classes of vulnerabilities
 - ▶ E.g., Type-safe languages such as Java