SWE 681 / ISA 681
Secure Software Design & Programming:
Lecture 3: Low-level attacks
(and Defenses)

Dr. David A. Wheeler
2017-10-11
Abstract view of a program

Program

Process Data
(Structured Program Internals)

Input

Output

You are here

Call-out to other programs
(also consider input & output issues)
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 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 strings to prevent memory errors?
  – Bounds checks – must use right bounds
  – What to do on failure? Exit or?
  – What to do when string size == buffer size?
  – What should return value tell you?
  – What should be done if destination is larger?
Obvious solution in C

• “Obvious” solution when using C is to always check bounds
• However...
Many C functions don’t check bounds (examples)

- `gets(3)` – reads input without checking. Don’t use it!
- `strcpy(3)` – `strcpy(dest, src)` copies from `src` to `dest`
  - If `src` longer than `dest` buffer, keeps writing!
- `strcat(3)` – `strcat(dest, src)` appends `src` to `dest`
  - If `src + data in dest` longer than `dest` buffer, keeps writing!
- `scanf()` family of input functions – many dangerous options
  - `scanf(3)`, `fscanf(3)`, `sscanf(3)`, `vscanf(3)`, `vsscanf(3)`, `vfscanf(3)`
  - Many options don’t control max length (e.g., bare “%s”)
- Many other dangerous functions, e.g.:
  - `realpath(3)`, `getopt(3)`, `getpass(3)`
  - `streadd(3)`, `strecpy(3)`, and `strtrns(3)`
- It’s not just functions; ordinary loops can overflow
And C’s integer overflow semantics make overflow more likely

• Integers in C (and many other languages) use a fixed maximum number of bits
  – If exceed “maximum positive integer”, wraps to negative numbers & eventually back to 0
  – C/C++ give no warning/exception

• Buffer size calculations’ integers can wrap!
  – This can make buffer overflow attacks even more likely... and more dangerous
  – Calculate, then check resulting value before use
Two code solution alternatives: Bounds-checking & auto-resize

• Bounds-checking to stop overwrite; then if oversized:
  – Stop processing input
    • Reject and try again, or even halt program (turns into DoS)
  – Truncate data. Common approach, but has issues:
    • Terminates text “in the middle” at place of attacker’s choosing
    • Can strip off critical data, escapes, etc. at the end
    • Can break in the middle of multi-byte character
      – UTF-8 character can take many bytes
      – UTF-16 usually 2 bytes/character, but not if it’s outside BMP
    • Some routines truncate & return indicator so you can stop processing input
    • Way better to truncate than to allow easy buffer overflow attack

• Auto-resize – move string if necessary
  – This is what most languages do automatically (other than C)
  – Must deal with “too large” data
  – C: Requires more code changes/complexity in existing code
  – C/C++: Dynamic allocation manual, so new risks (double-free)
Solution 1: Traditional C solution (bounds-checking routines)

- Depend mostly on `strncpy(3)`, `strncat(3)`, `sprintf(3)`, `snprintf(3)`
  - First three are especially hard to use correctly
- `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
- `char *strncat(char *DST, const char *SRC, size_t LENGTH)`
  - Find end of string in DST (\0)
  - Append up to LENGTH characters in SRC there
- `int sprintf(char *STR, const char *FORMAT, ...);`
  - FORMAT is a mini-language that defines what to write
  - Results put into sprintf
  - FORMAT can include length control information
Solution 1: Traditional C solution - Strncpy/strncat problems

- Hard to use correctly
  - Do not NIL-terminate the destination string if the source string length is at least equal to the destination’s
    - So often need to write a NIL afterwards to make sure it’s there
  - strncat must be passed the amount of space left available, a computation easy to get wrong
  - Neither have simple signal of an overflow

- Strncpy(3) has big performance penalty vs. strcpy(3)
  - Strncpy(3) NIL-fills remainder of the destination
  - Big performance penalty, typically for no good reason

- Like all bounds-checking, can terminate “in the middle”
  - Leading to potentially malformed data
  - Yet difficult to detect when it happens
Solution 1: Traditional C solution – sprintf problems

- Use sprintf’s format string to set maximum
  - Can set string “precision” field to set maximum length
  - E.G. "%.10s" means “<= 10 bytes” (notice “.”)
    - NIL written... unless it’s maximum size 😞
    - So you need to write the NIL afterwards, & everyone forgets it
  - Beware: "%10s" (without “.”) sets min field width
    - Useless for preventing buffer overflow
  - If the size is given as a precision of "*", then you can pass the maximum size as a parameter
    sprintf(dest, "%.s", maxlen, src);
  - Controls sizes of individual parameters
- Easy to get wrong, hard to get right
Solution 1: Traditional C solution – snprintf (now we’re talking!)

- int snprintf(char * s, size_t n, const char * format, ...);
  - Writes output to buffer “s” up to n chars (no easy buffer overflow)
  - Always writes \0 at end if n>=1 (hooray!)
  - Must provide format string for even trivial cases, don’t let attacker control format string
  - Returns “length that would have been written” or negative if error, so result-checking can be slightly annoying
  - Even if “n” is short & data source long, it will keep reading input to its end (to determine the return value).
  - This can be inefficient or a security problem if an input string is long or not necessarily \0-terminated (since it always reads to end)

- One of the best solutions for fixed-buffer, traditional C
- Sample:
  len = snprintf(buf, buflen, "%s", original_value);
  if (len < 0 || len >= buflen) ... // handle error/truncation
Solution 1: Traditional C solution (continued): snprintf + precision

• What if you want to limit the output, detect truncation, and limit the number of bytes read?
  – snprintf usually keeps reading (to report its return value)
• Good traditional option is snprintf and precision spec
• Sample:
  
```c
  len = snprintf(dest, destsize, "%.*s", (int) srcsize, src)
  if (len < 0 || len >= buflen) ... // handle error/truncation
```
• Notes:
  – You need the “(int)” – easily forgotten
  – “src” need not be \0-terminated, it’ll stop reading after “srcsize” bytes (and \0-terminate the destination)
  – In some circumstances can use destsize as srcsize
  – If need to determine if src lacks \0, may need to check specially
Solution 2: strlcpy/strlcat (bounds-checking)

- Simple routines for writing “no more than X bytes”
  - Easier to use correctly than strncpy/strncat
  - E.G., Always \0-terminates if dest has any space
  - strlcpy doesn’t \0-fill, unlike strncpy (good!)
  - Easy to detect if terminates “in the middle”
    - Returns “bytes would have written” like snprintf
  - Usage: if (strlcpy(dest, src, destsize) >= destsize) ... // truncation!
  - From OpenBSD developers

- However
  - Truncates “in the middle” like traditional functions – doesn’t resize
    - Check if truncation matters to you (at least it’s easy to check)
  - Keeps reading from input even after dest size filled, like snprintf
    - That’s a problem if src not \0-terminated!
  - Strlcat has to find end-of-string (“Schlemeil the painter”) – not normally issue
  - Only two routines; many others are troublesome
  - Not universally available
Solution 3: C++ std::string class (resize)

• If using C++, avoid using char* strings
• Instead, use std::string class
  – Automatically resizes
  – Avoids buffer overflow
• However, beware of conversion
  – Often need to convert to char* strings
    • E.g., when interacting with other systems
  – Once converted, problems return
  – Conversion is automatic
• Doesn’t help C (C++ only)
Solution 4: asprintf / vasprintf

- asprintf() and vasprintf() are analogs of sprintf(3) and vsprintf(3), except auto-allocate a new string
  - int asprintf(char **strp, const char *fmt, ...);
  - int vasprintf(char **strp, const char *fmt, va_list ap);
  - Pass pointer to free(3) to deallocate
  - Returns # of bytes “printed”; -1 if error
- Simple to use, doesn’t terminate results in middle (“resize”)
  - char *result;
  - asprintf(&result, “x=%s and y=%s\n”, x, y);
- Widely used to get things done without buffer overflows
- Not standard (not in C11); are in GNU and *BSD (inc. Apple)
  - Trivial to recreate on others, e.g., Windows (< 20 LOC)
- Wide use can easily lead to memory leaks
- FreeBSD sets strp to NULL on error, others don’t
Solution 5: Various other C libraries

- Many C libraries have been devised to provide new functions that handle strings gracefully:
  - Glib (not glibc): Basis of GTK+, resizable & bounded
  - Apache portable runtime (APR): resizable & bounded
  - SafeStr

- Problem: Not standard, everyone does it differently
  - Making it harder to combine code, work with others
Solution 6: C11 Annex K bounds-checking

- C11 standard adds bounds-checking interfaces
  - Creates “safer” versions of C functions
  - Limits lengths of results
- E.G., strcpy_s(s1, s1max, s2);
  - Copies s2 to s1.
  - Doesn’t do “useless NIL” fill
  - On error, calls runtime-constraint handler function, controlled by set_constraint_handler_s(). This handler can permit returns
    - If it returns, returns 0 if ok, nonzero if a constraint failed
    - A key constraint: s1max > strnlen_s(s2, s1max)
- Does not automatically resize
- Not universally available.. I hope it will be
Solution 7: ISO TR 24731-2 (Dynamic)

- ISO TR 24731-2 defines some dynamic functions
- Most not widely implemented at this time
- “getline” automatically resizes to read a line
- Can create a “string stream” - a memory buffer instead of an open file
  - Can create using fmemopen(), open_memstream(), or open_wmemstream()
  - Then can use standard functions such as sprintf(), sscanf(), etc. with them
  - Dynamically allocates and resizes as necessary
- Again, not widely available
Compilation solutions

• Don’t need to modify source code
  – But do need source code (recompile it)

• Some approaches
  – Canary-based
  – Libsafe
  – Compiler-inserted alternatives (FORTIFY_SOURCE)
  – “Address sanitizer” (ASan) to be discussed later
    • ASan has a higher performance/memory cost
Canary-based approach (from Stackguard)

• “Stackguard” (Cowan) implemented “canary-based approach”
  – Insert “canary” value on stack, located before return value
  – Before returning, check that canary untouched
  – Make canary hard to forge (random / tricky value)
• Adds some overhead on procedure call/return
  – Often varying heuristics to determine when to apply
  – Overhead relatively low
• ProPolice
  – Like Stackguard, but also reorders values
• GCC -fstack-protector*
  – -fstack-protector adds canary if local char array >= N long (N defaults to 8)
  – -fstack-protector-strong adds canary in additional cases, e.g., -if local variable address is taken or passed, or if there’s an any array (ChromeOS uses this)
  – -fstack-protector-all adds canary to all functions (performance hit!)
• Microsoft /GS flag based on stackguard
Libsafe (library-level)

- Partial defense
- Wraps checks around some common traditional C functions. Wrapper:
  - Examines current stack & frame pointers
  - Denies attempts to write data to stack that overwrite the return address or any of the parameters
- Limitations:
  - Only protects certain library calls
  - Only protects the return address & parameters on stack, e.g., heap overflows are still possible
  - Cannot rely on it being there
  - Thwarted by some compiler optimizations
-D_FORTIFY_SOURCE=2 (gcc)

• GCC’s -D_FORTIFY_SOURCE=2 built into compiler
  – Replaces some string/memory manipulation function calls with bounds-checking version & inserts bound
  • Documentation lists: memcpy(3), mempcpy(3), memmove(3), memset(3), stpcpy(3), strcpy(3), strncpy(3), strncpy(3), strcat(3), strncat(3), sprintf(3), snprintf(3), vsprintf(3), vsnprintf(3), and gets(3)
  – Sometimes compile-time check, rest run-time
  – Unlike libsafe, has more info on expected bound

• Ubuntu & Fedora by default use both -D_FORTIFY_SOURCE=2 and -fstack-protector
Some Runtime/OS-level defenses

• Make stack non-executable
  – Makes program somewhat harder to attack
  – Attacker can counter, e.g., set return value to existing code
  – Per-program: Some programs depend on executable stacks (e.g., nested procedure thunks)

• Randomize code/data memory locations
  – E.G., “Address Space Layout Randomization” (ASLR)
  – Makes program somewhat harder to attack
    • E.G., harder to find useful return value
  – Attacker can counter, e.g., with “NOP sled”
    • Long sequence of do-nothing, so jumping anywhere there works
  – Some areas hard to randomly move
  – Can impose overhead (esp. if every execution randomizes)
  – Can create hard-to-find bugs

• “Guard pages” after the end of memory allocations
  – Useful for heap-based buffers. OpenBSD malloc(), valgrind, electric fence, ...
  – Can detect accesses that go all the way to guard page, and can rig allocation to make this more likely
Address Space Layout Randomization

• Goal: Move the code and data so that you cannot predict where gadgets will be
  – What is the best way to make unpredictable?
  – What is the easiest way to make unpredictable?
ASLR

• Goal: Move the code and data so that you cannot predict where gadgets will be
  – What is the best way to make unpredictable?
    • Randomize code and data location for each instruction and variable
  – What is the easiest way to make unpredictable?
• Goal: Move the code and data so that you cannot predict where gadgets will be
  – What is the best way to make unpredictable?
    • Randomize code and data location for each instruction and variable
  – What is the easiest way to make unpredictable?
    • Just move the base address of the segment
    • Called Address Space Layout Randomization
ASLR Impact

• How does it prevent exploitation of attacks?

• Suppose you find a buffer overflow flaw
  – You insert shellcode onto the stack
  – And jump to the stack address

• With **ASLR on the stack segment**
  – Cannot predict the target stack address
  – Can you overflow return address?
ASLR Impact

• How does it prevent exploitation of attacks?
• Suppose you find a buffer overflow flaw
  – You launch an ROP attack
  – And jump to the code address of first gadget
• With ASLR on the code segment
  – Cannot predict the target code address
  – Why not?
ASLR Impact

• What kind of attack enables an adversary to disable ASLR?

Stack

Heap

Data

Text
ASLR Impact

- What kind of attack enables an adversary to disable ASLR?
  - Those that leak information about addresses
  - Disclosure attacks
ASLR Impact

- What kind of attack enables an adversary to disable ASLR?
  - Those that leak information about addresses
  - Disclosure attacks
  - Suppose you could read a function pointer from the stack
  - How would you use that?
Grow stack other way?

- Grow stack other direction
  - Some CPUs do this natively
  - Can implement in software if CPU doesn’t
- Does make some attacks harder, but:
  - Only affects some attacks on stack
  - Some buffers deeper in stack, *attack still works*
  - If not native to CPU, slower & doesn’t integrate with existing code
Countermeasure/ counter-countermeasure

• Most modern systems include partial countermeasures against buffer overflow attack
  – Randomize locations, etc.
  – But these countermeasures are, in general, circumventable by attacker
  – Countermeasure/CCM escalation

• Best approach, by far, is to ensure code isn’t vulnerable to buffer overflow in first place
  – Everything else is second best
Released under CC BY-SA 3.0

• This presentation is released under the Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) license
• You are free:
  – to Share — to copy, distribute and transmit the work
  – to Remix — to adapt the work
  – to make commercial use of the work
• Under the following conditions:
  – Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work)
  – Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one
• These conditions can be waived by permission from the copyright holder
  – dwheeler at dwheeler dot com
• Details at: http://creativecommons.org/licenses/by-sa/3.0/
• Attribute me as “David A. Wheeler”