Finding Your C/C++ Pointer and Array Bugs

(a step-by-step tour to some useful tools beyond the debugger)

Klaus Kusche, May 2012

Contents

- Knowing your enemies
- First aid:
  Program checking, debugging, tracing
- Compiling your code with seatbelts:
  Address sanitizer & Co
- Dealing with plain off-the-shelf code:
  Valgrind and friends
- Similar tools for different purposes

Enemy #1: Bad pointers

- **NULL** pointer
- *Uninitialized* pointer:

  - Single pointer variable
    (simple - usually caught by the compiler)

  - Element of a struct or an array of pointers
    (much harder to find - compilers will *not* detect that!)

- Pointer to a local array or struct after the function has returned:

  „*use-after-return*“

Enemy #2: Arrays & pointer arithmetic

- Array *bounds violations*:
  - „*Off by one*“ errors in loops and size checks
  - *Unchecked* input values or strings exceeding the target array’s size
  - Missing **`'\0'`** *string termination*
- *Integer overflow* or negative values in index arithmetic or size calculations
- *Uninitialized integer values* used in pointer or index arithmetic

Enemy #3: Dynamic memory handling

- `malloc` object *bounds violations*
- „*use-after-free*": Accessing `free`'d heap objects
- *Double* `free` (of the same object)
- *Invalid* `free` (of a pointer not pointing to a `malloc` objects's *beginning*)
- Allocation/deallocation function *mismatch* (`new[]` + `delete`, `new` + `free`, `malloc` + `delete`, ...)
- (Memory *leaks*)
- (Memory *fragmentation*)

Enemy #4:
The dark corners of C / C++

- `printf` *format / argument mismatch*
  (fatal for *non-string argument* to `%s` !)
- *Variadic functions* in general (no typechecking!)
- Pointers ruined by
  32 bit / 64 bit *casts between pointer and* `int`
  (very common in 32 bit code ported to 64 bit!)
- *Non-pointer data* interpreted as a pointer:

  - wrong case in a *union*

  - *forced casts* (e.g. *base class ptr ==> derived class ptr*)

What's so nasty about these bugs?

- Immediate & debuggable *crash*:
  Be happy, you had very good luck! :-)
- Crash with *massively corrupted memory*:
  Debugger is unable to extract any info...
- *Delayed crash*:
  - Hours later
  - In completely unrelated parts of the program
- No crash at all:
  Program just silently gives *wrong results*...
- Random, *unreproducible behaviour*.

What makes them even more evil?

Array and pointer bugs
are by far the most frequent reason
for **security vulnerabilities**!

Exploit technique #1:

- Place your exploit code into some array.
- Overwrite the *return address* on the stack
  (or e.g. *method pointers* in objects)
  to jump to your exploit code…

Step #0:
The compiler is your friend - use it!

Most important & always forgotten:

Compile with

maximum warning level / options

and

maximum optimization level

(needed for dataflow analysis!).

Warnings are given for a reason,
read them carefully!

Step #1: Apply static program checkers

= Tools that try to find bugs
just by looking at the source.

Many marketing catchwords for the same basic principle:
Dataflow analysis, value or range propagation, symbolic execution, abstract interpretation, ...

==> *What range of values can a variable or pointer contain at a certain point of code?*

(`NULL` ? Undefined ? <0 ? Just between x and y ?)

`splint`, `uno`, ... (Open source), `pclint`,... (€)

Expectations and reality…

Many big companies swear on it and *require* static program analysis for all code written.

My personal experience:

Static analysis used as a quick check *usually provides only limited help:*

- Either detects *less* than a good compiler
- Or produces *tons of output* (>= 80 % *false positives*)
- Works well only with code *annotations* and carefully selected flags

Step #2: „My name is 'Dump', 'Core Dump' "

- Compile your code with *debugging info*: `gcc -g`
- Enable *dumps*: `ulimit -c` … (some large value)
- Let your program *crash* ==› core dump written
- Analyze the dump with the *debugger*:

  `gdb` *binary core*

  Display the *crash location*: „`where`"
  Display the *value of variables*: „`print` …"
- **Or**: Run your program within the debugger, set *watchpoints* on suspected variables

Step #3: Try `ltrace` and `strace` !

- `ltrace` traces all *shared library calls* & results
- `strace` traces all *system calls* & results

Only of *limited use* for pointer problems:

==> What happened *just before the crash*?

==> Perhaps the program forgot to check
for *error return values*?
(e.g. NULL return value of `fopen` !)

Both tools don't require any preparation,
not even debug info in the code!

Step #4: Make your binaries foolproof…

Compiler-based solutions …

- … *add bookkeeping code*
  to each memory allocation & de-allocation
  (local var's on function entry and exit, …)
  to keep track of each valid memory block
- … *replace* the `malloc` / `free` library functions
- … perhaps change the *memory layout*
  (add guard words to separate valid blocks)
- … add ***checking code*** („points to valid data?“)
  to ***each pointer/array access***

Old bounds-checking gcc clones: **bgcc** and **MIRO** (1)

Still one of the *best (but slowest) checking logics*:

- Keeps track of all *local and global* ***variables*** and all *valid* ***heap objects***
- For each pointer, ***knows the object it points to*** (only tool which does this!!!)
- Checks not only accesses, but also all ***pointer arithmetic***

==> finds bad pointers *early* (when created, not when dereferenced)

Old bounds-checking gcc clones: **bgcc** and **MIRO** (2)

- Detects *all pointer & array bugs, including:*
  - Pointers *jumping to another valid object*
  - *Uninitialized pointers!*
  - Many cases of *use-after-return*
- Used to detect *all dynamic memory problems* (including *use-after-free*)
- Lists all memory *leaks* after program ended
- Doesn't catch *crashes in library code* not compiled with bgcc.
- Doesn't detect *uninitialized non-pointer values*.

Old bounds-checking gcc clones: **bgcc** and **MIRO** (3)

bgcc is *C only*,
with *leak finder* & *very good error messages*

MIRO checks *C and C++*, but without leak finder

- Huge CPU (· 10-30) and memory (· 3) overhead
- Have been „the king of the road“ for 1995 - 2008
- *Unmaintained* since 2005 (bgcc) / 2008 (MIRO)

  (slowly becoming incompatible with current software: For example, bgcc fails to catch all `malloc` / `free` calls with modern versions of `glibc`…)

Address Sanitizer („Asan“)

The new „*King of the road*“:

- Started by Google
- Included in standard `LLVM/clang` (for years)
  (`LLVM/clang` = Apple’s open source C/C++ compiler)

  and in standard `gcc` (since 4.8)

- Handles *C and C++*
- Much *faster* than anything else
  (slowdown <=2 !)

Address Sanitizer's brothers

- **Thread Sanitizer:**

  Detects *data races* in multithreaded code

- **Memory Sanitizer:**

  Detects *reads of uninitialized memory*

- **Leak Sanitizer:**

  Provides a *memory leak* listing

Address Sanitizer's principles

- ***Direct mapping*** of *each byte* in the address space
  to a *huge valid / invalid table*
  (byte based, not block/object based!)
  ==> ***Very fast*** (*only bit shift & add*, no searching)
  but allocates 16 TB of virtual memory
  (only mapped to real mem on access to corresponding bytes)
- ***Guard words*** are inserted around
  each local array and each heap block
  ==> „Off-bounds“ pointers are catched *before*
  they reach the next valid memory block

Address Sanitizer's features

- Bounds-checks *local, global and heap data*
  (needs additional compile/link options for global data)
- Detect *most* *use-after-free*
  and *some* *use-after-return* bugs
- Detects most *double* free etc.
- *Doesn't* detect crashes in system *libraries*
- *Doesn't* detect most *uninitialized values*
- *Doesn't* detect pointers randomly pointing or
  jumping to *another valid memory area*

Other bounds-checking compilers

- **FailSafe C** (open source):

  - C only

  - Not updated for > 5 years

  I never tried it …

- **Parasoft Insure++**:

  Most powerful & most expensive commercial product …

Step #5:
Valgrind runs *any* code checked!

Valgrind is an open source universal
x86 ***binary code interpreter*** framework* ...
* the truth is by far more complex!

==> doesn't need the source, not even debug info!

==> works on plain, *unmodified exe's and lib's*!
(no need to recompile / relink!)

==> also *checks all library code*!

... where ***plugins*** may add code
before and after each instruction executed!

Valgrind's **memcheck** plugin

- … maintains a „*valid*“ bit and an „*initialized*“ bit (set at first write) *for each byte* in memory,
- … checks *each memory access*,
- … replaces the **`malloc`** / **`free`** (`new` / `delete`) library calls and *all system calls*.

The bad news:

- Code runs 10-30 times *slower*
- … and becomes about 15 times *larger*!
- 3 times as much *memory* is needed for data!

Memcheck's power …

Memcheck detects

- almost all *dynamic memory (heap) problems*
- all accesses to *uninitialized data*
- all accesses to *invalid memory areas*
- most *system calls with invalid pointers*

… in your code and in *any library*!

… and it gives a complete memory *leak* listing!

… and blind spots

Memcheck will **<u>not</u>** detect

- bounds violations for *<u>local and global data</u>*

  (it checks bounds *<u>only</u>* for `malloc`'ed blocks, it *<u>can't insert guards</u>* on stack or global data!)
- most local object pointers *<u>used after return</u>*
- pointers jumping to *<u>another valid memory area</u>*

Valgrind's **SGCheck** plugin

... detects *what memcheck misses* (but *nothing else*):

For *local and global data* only (but *not* the heap!):
- *Bounds violations*
- *Pointers jumping between objects*
- *Use-after-return*

How?

- It reads the *size and location* of each local / global array from the *debug info*.
- For *each pointer* to locals/globals, it remembers *to which array it is pointing* (like bgcc / MIRO).

Valgrind's other plugins…

- **Cachegrind:**
  Cache and branch prediction hit rate
- **Callgrind, BBV, Lackey:**
  Execution profiling and call graphs
- **Helgrind, DRD:**
  Multithreading lock & race condition check
- **Massif, DHAT:**
  Heap object access profiling

Projects similar to Valgrind

DrMemory (new, active Open Source project, developed at Google for Chrome):

- Also works on *unmodified* exe's and lib's by *runtime code modification*
- Also uses *runtime code instrumentation*
- Offers almost the same features as Valgrind's memcheck
- Said to be *faster*
- x86_32 only (no 64 bit version yet)

The commercial competition

Market leader: **IBM/Rational Purify / Quantify**

- About as powerful (and as slow) as Valgrind
- Works by analyzing and adding checking code to all exe's and lib's *before* execution

  ==› no source or special compiler needed

  ==› separate „code instrumentation" step
  for all exe's and lib's needed (slow!)

- Very expensive (›› 5000 € per seat and year!)

Others: **Micro Focus BoundsChecker**, …

Wrong tool #1: gcc's „Stack Smashing Protector"

Compile with **`-fstack-protector`**

Catches *only* (*without* showing the culprit!) ...

- ... writes behind the end of *local arrays* which *damage the return address*
- ... by inserting a *guard value* *below the return address* of each function call
- ... and checking it when the function *returns*

==> *Fast*, very little overhead! (< 5 %, often *on by default*)

==> **Security feature, but useless for debugging!**

Wrong tool #2: Simple `malloc` replacements

<u>Replace</u> the `malloc/free` (`new/delete`) library:

Google Perftools, Dmalloc, MemProf, Mpatrol, ...

<u>Main purpose:</u>

Find memory <u>leaks</u>.

Dmalloc & Mpatrol (and in many cases standard glibc itself !) also detect *<u>simple</u>* cases of

- double free, free of bad pointers
- malloc object bounds violations (at `malloc/free` time!) by inserting boundary guard words

==> Won't help against most of our enemies!

Wrong tool #3: VM-based `malloc` replacements

Electric Fence / DUMA (old, unmaintained!) use

Virtual Memory Management

for protection: They allocate

- one separate VM page per malloc object
- + one *invalid page between two allocated pages*.

==> They detect *some* gross bounds violations
and *some* use-after-free cases ...

==> ... but require *huge* amounts
of real & virtual memory!