The Young Man And The C Reloaded

Dustin Laurence

- Optional: clone the repo: git@github.com:dllaurence/securec.git (ignore the parts I don't reference in the talk)
- If you don't already have them, install git, gcc and the toolchain, GNU make, clang, valgrind, and type 'make' at the top level.

Example: Signed Overflow

Consider the code in **`src/signed-overflow.c`** in the repo

- `will_overflow()` is the code of interest.
- The rest is driver code.

Two questions:

- What is the **intended** behavior of `will_overflow()`?
- What will the **actual** behavior be?

src/signed-overflow.c

```
int will_overflow(int n) { return (n + 1) < n; }
int plus_one(int n) { return n + 1; }
int main(void) {
    int prediction = will_overflow(INT_MAX);
    int actual = plus_one(INT_MAX) == INT_MIN;
    if (prediction == actual) { printf(“SUCCESS\n”); }
    else                      { printf(“FAILURE\n”); }
    return 0;
}
```

Results depend on the compiler and flags

Run `./test-signed-overflow.sh`:

- In all cases, `INT_MAX+1` actually wrapped to `INT_MIN`
- With -O0, `will_overflow()` correctly predicted the overflow.
- With -O1, it succeeded with GCC and failed with Clang.
- With -O2, it failed with both compilers.
- The behavior depended on compiler and optimization level!

WHY?!?

src/unsigned-overflow.c

```c
int will_overflow(unsigned n) { return (n + 1) < n; }
int plus_one(unsigned n) { return n + 1; }
int main(void) {
    int prediction = will_overflow(UINT_MAX);
    int actual = plus_one(UINT_MAX) == 0;
    if (prediction == actual) { printf(“SUCCESS\n”); }
    else                      { printf(“FAILURE\n”); }
    return 0;
}
```

But not for unsigned!

Run **`./test-unsigned-overflow.sh`**:

- In all cases, `UINT_MAX+1` wrapped to `0`
- In all cases, `will_overflow()` correctly predicted the overflow.
- The behavior was identical with both compilers and all optimization levels.

WHY did it work this time?

What If I Told You That Wasn't C?

What if I told you that the first program behaved unexpectedly because it was not actually written in C at all?

Red Pill, Blue Pill

"You take the blue pill—the talk ends, you wake up in your nice, comfortable text editor and believe whatever you want to believe. You take the red pill—you stay in this talk and I show you how deep the rabbit hole of undefined behavior goes."

Welcome To Reality

If you're still here, you have chosen to swallow the Red Pill.

- You might think the first program was written in C because the compiler accepted it. Remember:

The Compiler is a Machine. The Machines lie.

C and C++ Are Different

We usually think of a standard as precisely and uniquely defining the behavior of programming language constructs.

- True for some languages
- True with a few exceptional edge cases for others.

C and C++ are Terrifyingly Different

The Roll-Call Of Terror

In the C and C++ standards, there are four other possibilities (in order of increasing chaos and mayhem):

1. Locale-specific: e.g. `islower()` can return true for characters other than `'a'-'z'`.
2. Implementation-defined: e.g. sign bits may or may not be propagated when a signed integer is right-shifted.
3. Unspecified: e.g. the order of evaluation of function arguments.
4. And worst of all….

Undefined Behavior

“Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes **NO REQUIREMENTS**.”

If that isn't terrifying, you must have misunderstood.

“No Means No”

- “'When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose' – (famous post on comp.std.c)
- “Any undefined behavior in C gives license to the implementation to produce code that formats your hard drive.” – Chris Lattner, principal author of LLVM and Clang

What Actually Happens

Maybe compiler writers don’t actually do that (but c.f. Ken Thompson’s “Trusting Trust” paper!), but:

- The compiler **will** do whatever is fastest,
- **that** will create a vulnerability in your code,
- **that** will allow someone to run arbitrary code on your machine,
- and **that is the code that will format your hard drive.**

“Most of the security vulnerabilities...are the result of exploiting undefined behaviors in code.” (Seacord)

Undefined Behavior Lurks Everywhere

All of the following are undefined:

- Accessing beyond the ends of an array or memory block
- Just creating a pointer out of {bounds + one past end}
- Bit shifts the width of a type or greater
- Many uses of ++/-- twice in the same expression
- Modifying a string literal
- Most type-puns, depending on the exact standard
- Comparing pointers that do not point to the same block
- An unmatched ' or " (!!!)
- ***Some*** files ending w/o a final newline (!!!)
- ...and nearly 200 more cases

How did C and C++ End Up Like This?

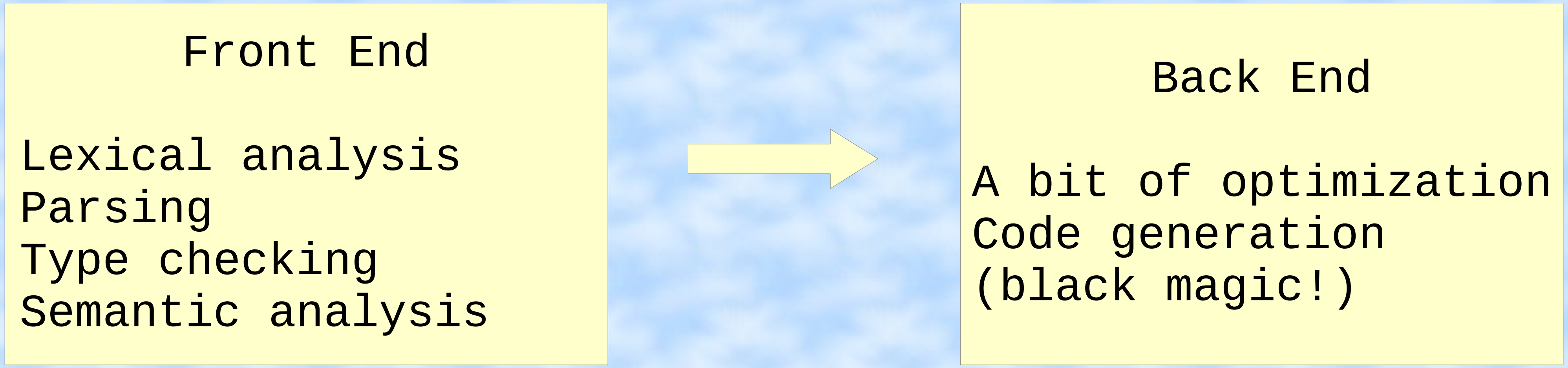
C and C++ design principles:

- "Make it fast, even if it is not guaranteed to be portable….Trust the programmer." – Original C standard committee charter
- "Leave no room for a lower-level language below C++ (except assembler)." – C++ "Low Level Programming Support Rules"

Performance at all costs turns out to be a monster with extremely inobvious consequences.

The Compiler We Think We Have

A lot of us have an old-fashioned mental picture of the compiler:



What We Think The Compiler Does

The major tasks of the compiler are:

- The front end discovers the meaning of the program, line by line.
- The back end generates code with the same meaning, line by line.

So naturally we program as though the source is executed line by line.

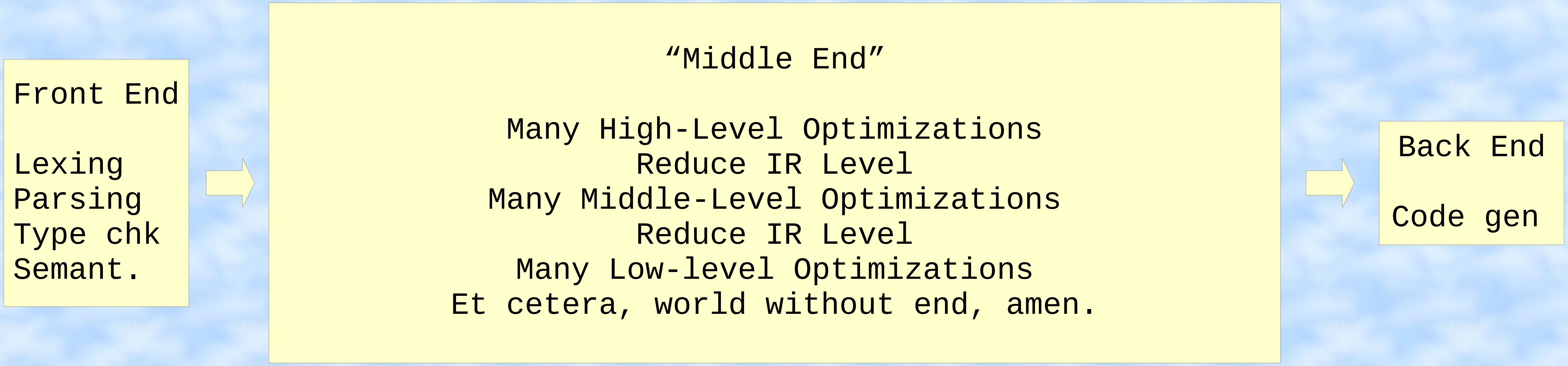
We think of undefined behavior as simply allowing the compiler to use single-machine instructions, “do what the hardware does,” and avoid run-time checks.

The Simple Compiler Picture Is Wrong

- This mental model worked OK back when some of us learned C (and went to school uphill both ways, etc.).
- It worked because compilers were stupid, not because it fit the C standard.
- They’re not stupid enough for that picture to work anymore.

The Compiler We Actually Have

A modern compiler looks more like like this:


Front End
Lexing
Parsing
Type chk
Semant.
“Middle End”
Many High-Level Optimizations
Reduce IR Level
Many Middle-Level Optimizations
Reduce IR Level
Many Low-level Optimizations
Et cetera, world without end, amen.
Back End
Code gen

What The Compiler Actually Does

The major tasks of the compiler are:

- The front end discovers the meaning of the program, line by line.
- Most of the code is in the "middle end," which transforms the line-by-line program in amazing and non-local ways.
- The back end generates code with the same meaning as the transformed program.
- But the transformed program itself need not have the same meaning as the original whenever undefined behavior occurs.

No Means No

- The only necessary relationships between the source and the object code are those imposed by the standard.
- The standard imposes no requirements on programs that invoke undefined behavior.
- ...really.

How Would A Compiler Exploit This License?

We can categorize functions into three types:

1. Functions which do not depend on any UB. The optimizer has to behave and therefore can't do anything “interesting”.
2. Functions which may or may not invoke UB depending on inputs (or other context). The optimizer has some but not complete license—this is the “interesting” case.
3. Functions which always depend on UB. Also uninteresting, the optimizer should just remove them entirely.

Optimization Requirements

- Must behave correctly if no UB occurs.
- Should be as fast (or small) as possible for this case.
- All behaviors are standard-conforming if UB occurs.
- Optimization in the face of UB is irrelevant because we “trust the programmer” not to write meaningless code.

Optimizing A Type 2 Function

Conclusion: for maximal performance **the optimizer should assume that a Type 2 function will never be passed arguments which would trigger UB!**

- Imposes the fewest constraints
- Allows maximal behavior in the no-UB case!

Example Type 2 Function

```c
// Behavior is Undefined if n == INT_MAX
int will_overflow(int n)
{
    return (n+1) < n;
}
```

UB-Enabled Optimization

What should the optimizer do with `will_overflow()`?

- `n+1` is undefined iff `n == INT_MAX`.
- Therefore, the optimizer should assume that `n` is never `INT_MAX`.
- Therefore `n+1 < n` can be simplified to zero!

C analog of optimized version

```
// Optimizer assumes that n will
// never be INT_MAX
int will_overflow(int n)
{
    return 0;
}
```

Actual generated assembly

```
; int-overflow-gcc-O2.s

xorl %eax, %eax ;;; %eax = 0
ret             ;;; return %eax
```

Now We Know What Happened

- will_overflow() is a type two function, and I passed it an argument that invoked undefined behavior.
- That means its behavior cannot be predicted from the source.
- The unsigned analog is a Type 1 function and the optimizer had to behave.

Undefined Behavior Is Inherently Unstable

Let's modify will_overflow slightly (unpredictable-ub.c):

```
int will_overflow(int n) {
    return (n + 1) == INT_MIN;
}
```

We'll do the same for the unsigned case in predictable-db.c

Should the results change? Will they? How?

Instability In Source Interpretation

Run **`./test-unpredictable.sh`**:

What happened this time?

- Again, `INT_MAX+1` always wraps to `INT_MIN`
- With gcc and -O2, `will_overflow()` failed.
- All other cases succeeded.

The results depended not only on the compiler and optimization level, but also on details of the source that appeared completely equivalent.

Instability In Time

Another example: GCC Bugzilla #71892 (Bacula developer)

- G++ 6.0 started deleting checks for `this` being `NULL` (which can't happen according to the C++ standard).
- And calls to memset that have no effect in C's model machine.

Some responses:

- You seem to be confusing "it worked OK until now" with "this code is valid according to the language standard."
- "...it got smarter."

Time Travel, or Why Dr. Who Uses C

- The standard even permits time travel as a consequence of UB, and this actually happens.
- Most of us unconsciously assume that a program is undefined starting at the point where undefined behavior occurs.
- In fact, the meaning of the **entire program** is **retroactively** undefined from it's initial invocation.
- Otherwise the optimizer would have to re-order instructions less aggressively. (Speed at any cost!)

What Is A “Safe Language”?

A computer language's abstraction is a virtual machine that is easier to work with than the assembly (or other language) it is built on.

When people say a language is “safe”, they mean:

- Its abstractions are air-tight
- Its virtual machine is a closed sandbox
- Its virtual machine cannot be broken from within
- You can’t tell from within that you’re in its Matrix

C and C++ Are Not Safe Languages

- By contrast, the C and C++ abstractions are leaky and their virtual machines are **NOT** sandboxes.

  Undefined Behavior indicates precisely where the abstraction can be broken without outside help.

  They are glitches in the C/C++ Matrix.

- Once broken, the source code is useless because its only meaning is within the Matrix (the virtual machine abstraction).

Welcome To The Terrifying Reality of C/C++

I told you the first program wasn't written in C at all, even though the second was.

- Now that you've swallowed the Red Pill, you know why that's true.
- You also know why the compiler accepted it anyway.

The Machines Lie

The Compiler is a Machine. The Machines lie.

They also may hurt you if you use glitches in the matrix rather than playing their blue-pill game.

Now What?

We have at least three options:

- The Blue Pill: write naive C and depend on what seems to work
- The Red Pill: figure out how to cope with Reality
- Walk out of the theater: abandon C and C++

Quitting C/C++ Isn’t Practical

- Critical legacy code (OS, virtual machines, ...)
- Infrastructure that needs extreme performance
- Tasks that require extreme control (crypto, embedded, realtime, low latency, constant time)
- Few suitable replacements for what C/C++ are good at
- Too much community and industry inertia to adopt replacements when available

The Blue Pill Doesn’t Work Well

I.e. do what seems to work even if technically undefined

- Really only works at moderate levels of security, reliability, and maintainability
- Most security holes in C/C++ involve undefined behavior at some level

The Red Pill Approach

- Learn to avoid the edge cases rather than knowing all 200+ of them (build our own abstraction layer).
- Make avoiding them a habit.
- Internalize the programming values that underly good habits.
- Use tools.
- Practice defense in depth.
- Don’t let the perfect be the enemy of the good.

Red Pill By Example

- A long list of sharp edges and dark corners does not make for a good talk.
- Generalize and apply the following examples to other cases.
- At the end I will give you some starting points to find those cases.

Prefer Unsigned Arithmetic

Begin with the will_overflow() example:

- How can we habitually avoid that problem?
- Many uses of signed types don't actually need to be signed (why are you looping on signed array indices?).
- Other uses can be eliminated with more work (this often improves the code in general).
- Habit: avoid signed arithmetic when it isn't necessary.

Trivial Example

```
for (size_t i=0; i<N; i++)
```

Instead of the common

```
for (int i=0; i<N; i++)
```

- size_t is unsigned (and as a bonus, more portable).
- Now we know at a glance that no computation with i is undefined because of signed overflow.
- Unfortunately, it often isn’t that simple.

Problems Preferring Unsigned

Unfortunately, preferring unsigned is harder than it sounds:

- Unsigned overflow/underflow can still be tricky.
- Signed/unsigned comparisons and conversions are still hard.
- Minimizing conversions makes signedness infectious—you end up needing to change other variables to match.
- “Hard” within the C abstraction is still better than the whim of the optimizer.

`stddef.h` and `stdint.h`

Habit: avoid “default” types like int.

- `size_t` is unsigned and large enough for the size of any object.
- `intmax_t` and `uintmax_t` are safe conversions for any width.
- Low-level usage of pointer values done through `intptr_t` and `uintptr_t` (if they exist!) is non-portable, but not undefined.
- Explicitly sized types (`int32_t`, `uint64_t`)

Habit: typedef types to document what they mean!

Yet Another Reason LibC Sucks

Lots of signed types in libc come from in-band error messages:

- E.g. read() returns a signed ssize_t solely to permit returning -1 in case of error.
- Comparing with our own unsigned variables is error prone.

Wrap it and get the conversion correct once and for all:

- `my_error my_read(size_t *bytes_read, …)`

Integer Habits

Habits:

- Habit: separate results and error codes (don’t fake algebraic data types without compiler support).
- Habit: use variables to mean only one thing.
- Related: function parameters should be in or out, not in-out.
- Habit: enforce better rules on external interfaces by wrapping aggressively.

Make The Compiler Your Batman

One of the other responses to GCC bug #71892:

"There is an option to disable both of these."

There are **LOTS** of useful flags that tell the compiler to

- ...warn at compile time about problematic constructs, or
- ...include run-time checks, or
- ...change the language to a dialect you find more congenial.
- They seem to be as rarely exploited as they are valuable.

Maybe We All Just Need A Hug

- Maybe The Machines aren’t entirely evil
- Maybe we’re just misusing the compiler
- Maybe it will be nicer to us if we ask politely?

Compile-Time Signed Arithmetic Checks

We can optionally choose a more friendly dialect of C by defining things the standard leaves undefined:

- -Wstrict-overflow: warn at compile-time when the optimizer exploits undefined overflow
- -fno-strict-overflow: don't fully exploit undefined overflow
- -fwrapv: fully define signed arithmetic (two's complement) (e.g. Postgres)

Run-Time Signed Arithmetic Checks

- -ftrapv: check at run-time and trap on signed overflow
- fsanitize=signed-integer-overflow: check at run-time and print a diagnostic on overflow (suboption of -fsanitize=undefined)
- -fsanitize-recover: attempt to continue after overflow to find more errors

Compile-Time Conversion Checks

Sign conversion problems are a complex topic you'll have to read up on, but the compiler can help here as well:

- -Wsign-compare: warn about comparisons between signed and unsigned that can produce incorrect results (part of -Wextra)
- -Wsign-conversion: warn about implicit conversions that could change the sign of a value

Dialects As Testing Tools

Even if you don't want to use a non-standard dialect, the dialect changing flags are still useful:

- If you're truly writing to the standard, your test suite should produce identical results with or without flags like -fwrapv.
- You can automate this by running your test suite under more than one set of flags…
- ...and more than one compiler.
- We all need to do this a lot more.

Lessons Learned

- Habit: know what flags are available.
- Habit: know what flags are used on a particular project.
- Habit: choose flags as carefully as you write code.
- Compiler-specific, even non-portable constructs are fine (unless they don't meet your projects design goals). You're still working within some well-defined dialect.
- Undefinedness is not fine. There is no upside to being at the mercy of the optimizer.

A (Simplified) Linux Kernel Bug

```
void agnx_pci_remove(struct pci *pdev)
{
  struct hw *dev = pci_get_drvdata(pdev);
  struct agnx *priv = dev->priv;
  if (!dev) return;
  // … use *dev
```

Programmer Illusion Vs Compiler Reality

- Presumably the programmer's reasoning was "if `dev` is `NULL`, I never actually use whatever `dev->priv` evaluates to, so it doesn't matter that I obtained a meaningless value."
- The same logic is used for the value of an uninitialized variable.
- The optimizer's reasoning was "this is a Type 2 function, so I can assume dev is never NULL and delete the NULL check."

If the programmer and the compiler disagree, the compiler wins.

How To Fix?

Again, two possibilities:

1. Eliminate undefined behavior (write strictly to the C standard)
2. Define the behavior as the code stands (write to a dialect)

1A: Eliminate Undefined Behavior (C89)

```
void agnx_pci_remove(struct pci *pdev)
{
  struct hw *dev = pci_get_drvdata(pdev);
  struct agnx *priv = NULL;
  if (!dev) return;
  priv = dev->priv;
  // … use *dev
```

1B: Eliminate Undefined Behavior (C99+, C++)

```
void agnx_pci_remove(struct pci *pdev)
{
  struct hw *dev = pci_get_drvdata(pdev);
  if (!dev) return;
  struct agnx *priv = dev->priv;
  // … use *dev
```

Fix 1 Observations

- It wasn’t the value that was undefined, it was the code.
- You must always sanitize inputs before using them in any way at all. It’s safest to sanitize at the earliest possible moment.
- To my taste, Linus is wrong—C99-style declarations mixed with the code is cleaner.
- You may disagree, or use C89, but you must guarantee that your initializers don’t invoke undefined behavior.

Making Fix 1 A Habit

Habit: sanitize all values at the earliest possible moment in the code.

Lay out every function so that it is clear that this is being done:

- First Section: sanitize function parameters.
- Second Section: Acquire resources one by one, sanitizing each before acquiring the next.
- On non-recoverable errors, abort at the earliest possible moment.
- The rest of the function assumes valid inputs and resources.

A Standard Function Layout

```
// Sanitize inputs
if (!pdev) { /* return error */ }
// Acquire drvdata
struct hw *dev = pci_get_drvdata(pdev);
if (!dev) { /* return error */ }
// Acquire agnx
struct agnx *priv = dev->priv;
if (!priv) { /* return error */ }
```

More General Layout

1. Abort ASAP if the caller has not satisfied its contract.
2. Then do everything that needs undoing on later failure, undoing in reverse order and aborting if the callee cannot meet its contract.
3. Then do everything else that could fail, again aborting immediately if the callee cannot meet its contract.
4. Satisfy the callee’s contract and return normally.

Underlying Values

- Consistently do common tasks the same way every single time.
- Make sure that the code is not only correct, but looks correct.
- Keep logically coupled pieces of code (e.g. acquisition and sanity checking) as close and visually coupled as possible.

Fix 2: Define The Behavior

```
# Add to makefile
CFLAGS += -fno-delete-null-pointer-checks
```

Fix 2 Considered

- -fno-delete-null-pointer-checks tells GCC not to delete tests for NULL even when the optimizer thinks they're redundant.
- This is probably how you thought the compiler behaved already.
- This was the solution chosen for Linux (and Bind).
- The decisive issue was that it's also how a lot of kernel hackers thought the compiler behaved (i.e. existing code did this all over the place).

Lessons Learned

- Value: don’t “normalize deviance.” Either eliminate dependence on undefined behavior or get the compiler to define it unambiguously.
- As always, what matters most is not being at the mercy of the optimizer.

External Tools

We talked about the compiler as a debugging tool because its always available. There are many other tools:

- Many other useful GCC/Clang compiler flags
- Run your test suite on multiple compilers
- Clang Static Analyzer
- Valgrind
- Etc, etc.
- (whisper: D and Rust deserve more consideration than they get)

FINIS

“Nothing in life is so exhilarating as to be shot at without result.” – Winston Churchill

“That’s just why C is fun.” – Le Faux

Discussion?

Entry points to further reading:

- What Every C Programmer Should Know About Undefined Behavior, Lattner, Chris, LLVM Blog, May 13, 2011.
- A Guide To Undefined Behavior In C and C++, Regeher, John, Embedded In Academia blog, July 9, 2010
- Secure Coding in C and C++, 2nd Ed., Seacord, Robert C.