Beyond Traditional Compilation

Why the Linux community should stop the single compiler monopoly

Kostya Serebryany <kcc@google.com>
Linux Plumbers / LLVM, Aug 19 2015
“Dynamic Testing Tools” team at Google

● Goal: our users find their bugs w/o our help
  ○ 10000+ bugs fixed since 2008

● Chromium, Android, server-side devs; C++

● Since 2011: compiler instrumentation
Traditional C/C++ compilation

- foo.c
- foo.o
One compiler to compile them all
A **monopoly** (from **Greek** *monos* μόνος (alone or single) + *polein* πωλεῖν (to sell)) exists when a specific person or **enterprise** is the only supplier of a particular commodity  [...] 

Monopolies are [...] characterized by a lack of economic **competition** to produce the **good** or **service**, a lack of viable **substitute goods**
Monopoly is bad

- Yet “the one compiler” monopolized the Linux ecosystem
  - Kernel sources
  - GLIBC
  - Distribution builds
Why break the monopoly?
int main(int argc, char **argv) {
    int stack_array[100];
    stack_array[1] = 0;
    return stack_array[argc + 100];  } // BOOM

% ancc++ -O1 -fsanitize=address a.cc; ./a.out
==10589== ERROR: AddressSanitizer stack-buffer-overflow
READ of size 4 at 0x7f5620d981b4 thread T0
    #0 0x4024e8 in main a.cc:4
Address 0x7f5620d981b4 is located at offset 436 in frame
<main> of T0's stack:
This frame has 1 object(s):
    [32, 432) 'stack_array'
ASan report example: use-after-free

```c
int main(int argc, char **argv) {
    int *array = new int[100];
    delete [] array;
    return array[argc]; } // BOOM
```

```bash
% ancc++ -O1 -fsanitize=address a.cc && ./a.out
==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
    #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte region
freed by thread T0 here:
    #0 0x4058fd in operator delete[](void*) _asan_rtl_
    #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
    #0 0x405579 in operator new[](unsigned long) _asan_rtl_
    #1 0x4042f3 in main a.cc:2
```
ASan report example: stack-use-after-return

int *g;

void LeakLocal() {
    int local;
    g = &local;
    LeakLocal();
    return *g;
}

% ancc -g -fsanitize=address a.cc
% ASAN_OPTIONS=detect_stack_use_after_return=1 ./a.out
==19177==ERROR: AddressSanitizer: stack-use-after-return
READ of size 4 at 0x7f473d0000a0 thread T0
    #0 0x461ccf in main
    a.cc:8

Address is located in stack of thread T0 at offset 32 in frame
    #0 0x461a5f in LeakLocal() a.cc:2
This frame has 1 object(s):
    [32, 36) 'local' <= Memory access at offset 32

int main() {
    LeakLocal();
    return *g;
}
TSan report example: data race

int X;
std::thread t([&]{X = 42;});
X = 43;
t.join();

% ancc -fsanitize=thread -g race.cc && ./a.out
WARNING: ThreadSanitizer: data race (pid=25493)
  Write of size 4 at 0x7fff7f10e338 by thread T1:
    #0 main::$_0::operator()() const race.cc:4 ... 
  Previous write of size 4 at 0x7...8 by main thread:
    #0 main race.cc:5
int main(int argc, char **argv) {
    int x[10];
    x[0] = 1;
    return x[argc];
}

% ancc -fsanitize=memory a.c -g; ./a.out

WARNING: Use of uninitialized value
    #0 0x7f1c31f16d10 in main a.cc:4

Uninitialized value was created by an allocation of 'x' in the stack frame of function 'main'
UBSan report example: int overflow

int main(int argc, char **argv) {
    int t = argc << 16;
    return t * t;
}

% ancc -fsanitize=undefined a.cc -g; ./a.out
a.cc:3:12: runtime error:
signed integer overflow: 65536 * 65536
cannot be represented in type 'int'
int main(int argc, char **argv) {
    return (1 << (32 * argc)) == 0;
}

% ancc -fsanitize=undefined a.cc -g; ./a.out

a.cc:2:13: runtime error: shift exponent 32 is too large for 32-bit type 'int'
Kernel/GLIBC/Distros

- **Kernel**
  - KASAN: in trunk, 65+ bugs found
    - 35 use-after-free, 18 heap-out-of-bounds, 8 stack-out-of-bounds, 2 global-out-of-bounds, 2 user-memory-access
  - KTSAN: POC, 1 bug found & fixed
  - KMSAN: nope
  - KUBSAN: ???

- **GLIBC:**
  - Can build with ASan (tons of hacks)
  - 10+ bugs found

- **Ubuntu distro:**
  - Can build 60+ key libs with ASan/MSan/TSan using external scripts
  - Hard to use and maintain
• Cool, but “the one compiler” already has some of these too!

• Yes, but not all
• Yes, as the result of competition
• Wait, there is more
Sanitizers are not enough

- ASan, TSan, MSan, UBSan are “best-effort tools”:
  - They do not prove correctness
  - They are only as good as the tests are

- Beyond Sanitizers:
  - Improve test quality (aka test coverage) by fuzzing
  - Protect from security-sensitive bugs in production (hardening)
Control-flow-guided (coverage-guided) fuzzing

- Acquire a test corpus (e.g. crawl the web)
- Minimize the corpus according to some metric, e.g. \((\text{code coverage})/(\text{execution time})\)
- Mutate tests from the corpus and execute them
- Run the mutations with code coverage instrumentation
- Add the mutations to the corpus if new coverage is discovered
Sanitizer Coverage instrumentation

- `-fsanitize-coverage=`
  - `func/bb/edge`: records if a function, basic block or edge was executed
  - `indirect-calls`: records unique indirect caller-callee pairs
  - `8bit-counters`: similar to AFL, provides 8 state counter for edges
    - `(1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+)`

- Provides the status in-process and dumps data on disk at exit and
  - i.e. supports in-process and out-of-process clients
- Should be combined with ASan, MSan, LSan, or UBSan
- Typical slowdown within 10%
  - 8bit counters may be unfriendly to multi-threaded apps
libFuzzer

- Lightweight in-process control-flow guided fuzzer
  - Provide your own target function
    - void TestOneInput (const uint8_t *Data, size_t Size);
  - Build: -fsanitize-coverage=edge[,indirect-calls][,8bit-counters]
  - Build: -fsanitize={address,memory,undefined,leak}
  - Link with libFuzzer

- Targeted at libraries/APIs, not at large applications
Example: OpenSSL

SSL_CTX *sctx;
int Init() { ... }
extern "C" void TestOneInput(unsigned char *Data, size_t Size) {
    static int unused = Init();
    SSL *server = SSL_new(sctx);
    BIO *sinbio = BIO_new(BIO_s_mem());
    BIO *soutbio = BIO_new(BIO_s_mem());
    SSL_set_bio(server, sinbio, soutbio);
    SSL_set_accept_state(server);
    BIO_write(sinbio, Data, Size);
    SSL_do_handshake(server);
    SSL_free(server);
}
How quickly can you find Heartbleed with fuzzing?

I. 1 Second
II. 1 Minute
III. 1 Hour
IV. 1 Day
V. 1 Month
VI. 1 Year
Yet, we still need code hardening

- Heap-buffer-overflow or heap-use-after-free may overwrite VPTRs, function pointers, array sizes, etc
  - Hijacked VPTR in Chromium: Pwn2Own 2013 (CVE-2013-0912)

- Stack-buffer-overflow or stack-use-after-return may also overwrite return addresses

- Running ASan in production costs 2x CPU/RAM -- infeasible
  - ASan can be bypassed anyway
CFI (Control Flow Integrity)

- Compile with `-fsanitize=cfi-vcall -flto` (LTO!)
- Every disjoint class hierarchy is handled separately
  - Assumes the class hierarchy is a closed system; ok for Chrome
- Layout the vtables for the entire class hierarchy as a contiguous array
  - Align every vtable by the same power-of-2
- For every virtual function call site
  - Compile-time: compute the strict set of allowed functions
  - Run-time: perform a range check, alignment check, and a bitset lookup
- Optimizations:
  - A bitset of <= 64 bits requires no memory loads
  - No check if the bitset contains all ones
  - Optimize the layouts to minimize the bitset sizes
- Chrome: builds, runs, catches real bugs, costs < 1% CPU (Linux)
## CFI: generated x86_64 assembler

### # All ones

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>$0x4008f0,%ecx</td>
</tr>
<tr>
<td>mov</td>
<td>%rax,%rdx</td>
</tr>
<tr>
<td>sub</td>
<td>%rcx,%rdx</td>
</tr>
<tr>
<td>rol</td>
<td>$0x3b,%rdx</td>
</tr>
<tr>
<td>cmp</td>
<td>$0x2,%rdx</td>
</tr>
<tr>
<td>jae</td>
<td>CRASH</td>
</tr>
<tr>
<td>mov</td>
<td>%rbx,%rdi</td>
</tr>
<tr>
<td>callq</td>
<td>*(%rax)</td>
</tr>
</tbody>
</table>

...  

CRASH: ud2

### # <= 64 bits

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>$0x400e20,%edx</td>
</tr>
<tr>
<td>mov</td>
<td>%rax,%rcx</td>
</tr>
<tr>
<td>sub</td>
<td>%rdx,%rcx</td>
</tr>
<tr>
<td>rol</td>
<td>$0x3b,%rcx</td>
</tr>
<tr>
<td>cmp</td>
<td>$0xe,%rcx</td>
</tr>
<tr>
<td>ja</td>
<td>CRASH</td>
</tr>
<tr>
<td>mov</td>
<td>%rbx,%rdi</td>
</tr>
<tr>
<td>callq</td>
<td>*(%rax)</td>
</tr>
</tbody>
</table>

...  

CRASH: ud2

### # Full check

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>$0x401810,%edx</td>
</tr>
<tr>
<td>mov</td>
<td>%rax,%rcx</td>
</tr>
<tr>
<td>sub</td>
<td>%rdx,%rcx</td>
</tr>
<tr>
<td>rol</td>
<td>$0x3b,%rcx</td>
</tr>
<tr>
<td>cmp</td>
<td>$0x40,%rcx</td>
</tr>
<tr>
<td>ja</td>
<td>CRASH</td>
</tr>
<tr>
<td>testb</td>
<td>$0x402140(%rcx)</td>
</tr>
<tr>
<td>je</td>
<td>CRASH</td>
</tr>
<tr>
<td>mov</td>
<td>%rbx,%rdi</td>
</tr>
<tr>
<td>callq</td>
<td>*(%rax)</td>
</tr>
</tbody>
</table>

...  

CRASH: ud2
More CFI

- Non-virtual member calls, indirect calls
  - -fsanitize=cfi-nvcall, -fsanitize=cfi-icall
- Casts (for polymorphic types)
  - -fsanitize=cfi-derived-cast, -fsanitize=cfi-unrelated-cast
- Do not require LTO??
- Allow class hierarchies to cross the DSO boundaries
  - Maybe not a great idea??
  - Control Flow Guard (/d2guard4 + /Guard:cf)
- More platforms
  - Coming soon: Android, OSX, Windows
SafeStack

- Place local variables on a separate stack (separately mmaped region)
  - `-fsanitize=safe-stack`
  - Linux, FreeBSD, OSX

- `stack-buffer-overflow/use-after-return` can’t touch the return addresses

- VTPRs and function pointers can still be affected
  - Combine with `-fsanitize=cfi`

- Chromium: costs < 1% CPU
SafeStack: code example

push    %r14
push    %rbx
push    %rax
mov     0x207d0d(%rip),%r14
mov     %fs:(%r14),%rbx  # Get unsafe_stack_ptr
lea     -0x10(%rbx),%rax # Update unsafe_stack_ptr
mov     %rax,%fs:(%r14)  # Store unsafe_stack_ptr
lea     -0x4(%rbx),%rdi
movl    $0x123456,-0x4(%rbx)
callq   40f2c0 <_Z3barPi>
mov     %rbx,%fs:(%r14)  # Restore unsafe_stack_ptr
xor     %eax,%eax
add     $0x8,%eax
pop     %rbx
pop     %r14
retq

int main() {
    int local_var = 0x123456;
    bar(&local_var);
}
The community can break the monopoly!

● First, make everything build with “another” compiler
  ○ Kernel, GLIBC, Distros

● Setup contiguous builds, don’t let it regress, ever

● Do not switch to “another” compiler completely, continue to use both

● Wait for 3-rd and 4-th compilers to appear and let them compete

● Profit!
Conclusions

- **ASAN, TSAN, MSAN, UBSAN**
  - Like a toothbrush: use them or risk losing your teeth

- **Guided fuzzing is an extremely powerful yet under-utilized technique**
  - Use it with Sanitizers
  - libFuzzer makes it easy

- **Bugs will still slip into production -- use hardening**
  - CFI for virtual calls, other calls, and casts
  - SafeStack

- The arms race continues, we are not done yet