MemorySanitizer

Evgeniy Stepanov, Kostya Serebryany

Apr 29, 2013
Agenda

- How it works
- What are the challenges
- Random notes
int main(int argc, char **argv) {
    int x[10];
    x[0] = 1;
    if (x[argc]) return 1;
    ...
%
c clang ... stack_umr.c && ./a.out

WARNING: Use of uninitialized value
      #0 0x7f1c31f16d10 in main stack_umr.c:4
Uninitialized value was created by an
allocation of 'x' in the stack frame of
function 'main'
Shadow memory

- 1 application bit => 1 shadow bit
  - 1 = poisoned (uninitialized)
  - 0 = clean (initialized)

- Alternative: 8 bits => 2 bits (Valgrind)
  - 0 = all ok; 1 = all poisoned; 2 = not addressable
  - 3 = partially poisoned (use secondary 1:1 shadow)
  - Slower to extract (VG is slow anyway)
  - Racy updates (VG is single-threaded)
  - More important if combined with redzones
    - VG, but not MSan
Direct 1:1 shadow mapping

Shadow = Addr - 0x400000000000;
Shadow propagation

Reporting every load of uninitialized data is too noisy.

```c
struct {
    char x;
    // 3-byte padding
    int y;
}
```

It's OK to copy uninitialized data around.

Uninit calculations are OK, too, as long as the result is discarded. People do it.
Shadow propagation

- Assign shadow temps to app IR temps.
- Propagate shadow values through expressions
  - $A = \text{op} B, C \Rightarrow A' = \text{op}' B, C, B', C'$
- Propagate shadow through function calls: arguments & return values.
- Report UMR only on some uses (branch, syscall, etc)
  - PC is poisoned (a conditional branch)
  - Syscall argument is poisoned (a side-effect)
Shadow propagation

- \( A = \text{const}: \quad A' = 0 \)
- \( A = \text{load } B: \quad A' = \text{load } B \land \text{ShadowMask} \)
- \( \text{store } B, A: \quad \text{store } B \land \text{ShadowMask}, A' \)
- \( A = B \ll C: \quad A' = B' \ll C \)
- \( A = B \land C: \quad A' = (B' \land C') \lor (B \land C') \lor (B' \land C) \)
- \( A = (B == C): \)
  - \( D = B \xor C; \quad D' = B' \lor C'; \quad \text{now } A = (D == 0) \)
  - \( A' = !(D \land \lnot D') \land D' \)
  - Exact.
- Vector types: easy!
Approximate propagation

\[ A = B + C: \quad A' = B' \mid C' \]

Exact propagation logic is way too complex. This is faster than test-and-report.

Bitwise OR is common propagation logic.
- Never makes a value "less poisoned".
- Never makes a poisoned value clean.
Relational comparison

\[ A = (B > C) : A' = (B' \mid C' \neq 0) \]

```c
struct S { int a : 3;  int b : 5; };  
bool f(S *s) { return s->b;  }

%tobool = \texttt{icmp ugt i16} %bf.load, 7
```

False positive when \texttt{a} is uninitialized.
Relational comparison

\[ A = (B > C) : A' = ? \]

\[ B = \text{xxxxxx???} \]
\[ c = 000000111 \]

Is \( B > C \)?
1. Yes
2. No
3. Maybe
Relational comparison

A = (B > C) : A' = ?

- \( B_{\text{min}} = \text{MinValue}(B, B') \); \( B_{\text{max}} = \text{MaxValue}(B, B') \)
- \( C_{\text{min}} = \text{MinValue}(C, C') \); \( C_{\text{max}} = \text{MaxValue}(C, C') \)
- \( A' = ( (B_{\text{max}} > C_{\text{min}}) \neq (C_{\text{max}} > B_{\text{min}}) ) \)
- Slow! Up to 50% performance degradation on specs.

Current solution:
- Exact propagation if B or C is a constant.
- \( A' = B' \mid C' \) otherwise.
Tracking origins

- Where was the poisoned memory allocated?
  ```c
  a = malloc() ...
  b = malloc() ...
  c = *a + *b ...
  if (c) ... // UMR. Is 'a' guilty or 'b'?  
  ```

- Valgrind --track-origins: propagate the origin of the poisoned memory alongside the shadow

- MSan: secondary shadow
  - Origin-ID is 4 bytes, 1:1 mapping
  - 2x additional slowdown
Secondary shadow (origin)

Origin = Addr - 0x200000000000;
Tracking origins

- Origin propagation

\[ B = \text{op} \ D, \ E \quad B'' = \text{select} \ E', \ E'', \ D'' \]
\[ A = \text{op} \ B, \ C \quad A'' = \text{select} \ C', \ C'', \ B'' \]
Call instrumentation

call void @f(i64 %a, i64 %b)

store i64 %Sa, ... @__msan_param_tls ...
store i64 %Sb, ... @__msan_param_tls ...
call void @f(i64 %a, i64 %b)

__msan_param_tls:

A'    B'
VarArg handling

Problem: va_arg is lowered in the frontend.

```plaintext
%p = alloca [1 x %struct.__va_list_tag], align 16
%arraydecay1 = bitcast [1 x %struct.__va_list_tag]* %p to i8*
call void @llvm.va_start(i8* %arraydecay1)
%gp_offset_p = getelementptr inbounds [1 x %struct.__va_list_tag]* %p, i64 0, i64 0, i32 0
%gp_offset = load i32* %gp_offset_p, align 16
%fits_in_gp = icmp ult i32 %gp_offset, 41
br i1 %fits_in_gp, label %vaarg.in_reg, label %vaarg.in_mem

vaarg.in_reg:
; preds = %entry
%0 = getelementptr inbounds [1 x %struct.__va_list_tag]* %p, i64 0, i64 0, i32 3
%reg_save_area = load i8** %0, align 16
%1 = sext i32 %gp_offset to i64
%2 = getelementptr i8* %reg_save_area, i64 %1
%3 = add i32 %gp_offset, 8
store i32 %3, i32* %gp_offset_p, align 16
br label %vaarg.end

vaarg.in_mem:
; preds = %entry
%overflow_arg_area_p = getelementptr inbounds [1 x %struct.__va_list_tag]* %p, i64 0, i64 0, i32 2
%overflow_arg_area = load i8** %overflow_arg_area_p, align 8
%overflow_arg_area.next = getelementptr i8* %overflow_arg_area, i64 0
store i8* %overflow_arg_area.next, i8** %overflow_arg_area_p, align 8
br label %vaarg.end

vaarg.end:
; preds = %vaarg.in_mem, %vaarg.in_reg
%vaarg.addr.in = phi i8* [%2, %vaarg.in_reg], [%overflow_arg_area, %vaarg.in_mem]
%vaarg.addr = bitcast i8* %vaarg.addr.in to i32*
%4 = load i32* %vaarg.addr, align 4
```

What is %4's shadow?
VarArg handling

Solution (bad): Fill va_list shadow in va_start.
• Platform-dependent.
• Complex and error-prone.
• Works.

Solution (good):
• Emit va_arg in the frontend.
Ret instrumentation

%a = call i64 @f()
%a = call i64 @f()
%Sa = load i64 @__msan_retval_tls

f():
...
store i64 %Sa, @__msan_retval_tls
ret i64 %a

__msan_retval_tls: A'
SIMD intrinsics

Guessing memory effects based on signature and mod/ref behaviour:

- vector store
- vector load
- arithmetic, logic, etc
- special handling for mem*, va_* and bswap.
MSan overhead

- **Without origins:**
  - CPU: 3x
  - RAM: 2x

- **With origins:**
  - CPU: 5x
  - RAM: 3x + malloc stack traces
Optimization

● MemorySanitizer instrumentation inhibits inlining.
  ○ Must be done late.
● Lots of redundant instrumentation.
  ○ Re-run some generic optimization passes.
    ■ 13% perf improvement.

Future ideas.

● App, shadow and origin locations never alias.
● Fast pass origin tracking.
Tricky part :(  

- Missing any write instruction causes false reports  
- Must monitor ALL stores in the program  
  - libc, libstdc++, syscalls, inline asm, JITs, etc
Solution #1: partial

- Use instrumented libc++ or libstdc++
- Wrappers for libc (more than 140 functions)
- Handlers for raw system calls (in-progress)
- Instrument everything else
  - Or isolate uninstrumented parts (ex.: zlib has ~2 interface functions with clear memory effects)
- Works for some real apps:
  - Can bootstrap Clang
- FAST
Solution #2: static + dynamic

- Simple DynamoRIO tool (MSanDr)
  - Instrument stores by cleaning target shadow.
  - Instrument RET and every indirect branch by cleaning function argument shadow.
  - Avoids false positives.

- SLOW, unclear speedup potential
  - Very slow startup
  - Still much faster than Valgrind

- Applicable to all apps
  - Chrome (DumpRenderTree)
MSan summary

- Finds uses of uninitialized memory
- 10x faster than Valgrind
- Provides better warning messages
- Has deployment challenges
Q&A
Why not combine ASan and MSan?

- Slowdowns will add up
  - Bad for interactive or network apps

- Memory overheads will multiply
  - ASan's redzones * MSan's rich shadow

- Not trivial to implement