NAME

KMSAN - Kernel Memory SANitizer

SYNOPSIS

The *GENERIC-KMSAN* kernel configuration can be used to compile a KMSAN-enabled kernel using *GENERIC* as a base configuration. Alternately, to compile KMSAN into the kernel, place the following line in your kernel configuration file:

options KMSAN

#include <sys/msan.h>

void
kmsan_mark(*const void *addr, size_t size, uint8_t code*);

void
kmsan_orig(*const void *addr, size_t size, int type, uintptr_t pc*);

void
kmsan_check(*const void *addr, size_t size, const char *descr*);

void
kmsan_check_bio(*const struct bio *, const char *descr*);

void
kmsan_check_ccb(*const union ccb *, const char *descr*);

void
kmsan_check_mbuf(*const struct mbuf *, const char *descr*);

DESCRIPTION

KMSAN is a subsystem which leverages compiler instrumentation to detect uses of uninitialized memory in the kernel. Currently it is implemented only on the amd64 platform.

When **KMSAN** is compiled into the kernel, the compiler is configured to emit function calls preceding memory accesses. The functions are implemented by the **KMSAN** runtime component and use hidden, byte-granular shadow state to determine whether the source operand has been initialized. When uninitialized memory is used as a source operand in certain operations, such as control flow expressions or memory accesses, the runtime reports an error. Otherwise, the shadow state is propagated to destination operand. For example, a variable assignment or a **memcpy**() call which copies uninitialized

memory will cause the destination buffer or variable to be marked uninitialized.

To report an error, the **KMSAN** runtime will either trigger a kernel panic or print a message to the console, depending on the value of the **debug.kmsan.panic_on_violation** sysctl. In both cases, a stack trace and information about the origin of the uninitialized memory is included.

In addition to compiler-detected uses of uninitialized memory, various kernel I/O "exit points", such as copyout(9), perform validation of the input's shadow state and will raise an error if any uninitialized bytes are detected.

The **KMSAN** option imposes a significant performance penalty. Kernel code typically runs two or three times slower, and each byte mapped in the kernel map requires two bytes of shadow state. As a result, **KMSAN** should be used only for kernel testing and development. It is not recommended to enable **KMSAN** in systems with less than 8GB of physical RAM.

FUNCTIONS

The **kmsan_mark**() and **kmsan_orig**() functions update **KMSAN** shadow state. **kmsan_mark**() marks an address range as valid or invalid according to the value of the *code* parameter. The valid values for this parameter are KMSAN_STATE_INITED and KMSAN_STATE_UNINIT, which mark the range as initialized and uninitialized, respectively. For example, when a piece of memory is freed to a kernel allocator, it will typically have been marked initialized; before the memory is reused for a new allocation, the allocator should mark it as uninitialized. As another example, writes to host memory performed by devices, e.g., via DMA, are not intercepted by the sanitizer; to avoid false positives, drivers should mark device-written memory as initialized. For many drivers this is handled internally by the busdma(9) subsystem.

The **kmsan_orig**() function updates "origin" shadow state. In particular, it associates a given uninitialized buffer with a memory type and code address. This is used by the **KMSAN** runtime to track the source of uninitialized memory and is only for debugging purposes. See *IMPLEMENTATION NOTES* for more details.

The **kmsan_check**() function and its sub-typed siblings validate the shadow state of the region(s) of kernel memory passed as input parameters. If any byte of the input is marked as uninitialized, the runtime will generate a report. These functions are useful during debugging, as they can be strategically inserted into code paths to narrow down the source of uninitialized memory. They are also used to perform validation in various kernel I/O paths, helping ensure that, for example, packets transmitted over a network do not contain uninitialized kernel memory. **kmsan_check**() and related functions also take a *descr* parameter which is inserted into any reports raised by the check.

IMPLEMENTATION NOTES

Shadow Maps

The **KMSAN** runtime makes use of two shadows of the kernel map. Each address in the kernel map has a linear mapping to addresses in the two shadows. The first, simply called the shadow map, tracks the state of the corresponding kernel memory. A non-zero byte in the shadow map indicates that the corresponding byte of kernel memory is uninitialized. The **KMSAN** instrumentation automatically propagates shadow state as the contents of kernel memory are transformed and copied.

The second shadow is called the origin map, and exists only to help debug reports from the sanitizer. To avoid false positives, **KMSAN** does not raise reports for certain operations on uninitialized memory, such as copying or arithmetic. Thus, operations on uninitialized state which raise a report may be far removed from the source of the bug, complicating debugging. The origin map contains information which can help pinpoint the root cause of a particular **KMSAN** report; when generating a report, the runtime uses state from the origin map to provide extra details.

Unlike the shadow map, the origin map is not byte-granular, but consists of 4-byte "cells". Each cell describes the corresponding four bytes of mapped kernel memory and holds a type and compressed code address. When kernel memory is allocated for some purpose, its origin is initialized either by the compiler instrumentation or by runtime hooks in the allocator. The type indicates the specific allocator, e.g., uma(9), and the address provides the location in the kernel code where the memory was allocated.

Assembly Code

When **KMSAN** is configured, the compiler will only emit instrumentation for C code. Files containing assembly code are left un-instrumented. In some cases this is handled by the sanitizer runtime, which defines wrappers for subroutines implemented in assembly. These wrappers are referred to as interceptors and handle updating shadow state to reflect the operations performed by the original subroutines. In other cases, C code which calls assembly code or is called from assembly code may need to use **kmsan_mark**() to manually update shadow state. This is typically only necessary in machine-dependent code.

Inline assembly is instrumented by the compiler to update shadow state based on the output operands of the code, and thus does not usually require any special handling to avoid false positives.

Interrupts and Exceptions

In addition to the shadow maps, the sanitizer requires some thread-local storage (TLS) to track initialization and origin state for function parameters and return values. The sanitizer instrumentation will automatically fetch, update and verify this state. In particular, this storage block has a layout defined by the sanitizer ABI.

Most kernel code runs in a context where interrupts or exceptions may redirect the CPU to begin execution of unrelated code. To ensure that thread-local sanitizer state remains consistent, the runtime

maintains a stack of TLS blocks for each thread. When machine-dependent interrupt and exception handlers begin execution, they push a new entry onto the stack before calling into any C code, and pop the stack before resuming execution of the interrupted code. These operations are performed by the **kmsan_intr_enter**() and **kmsan_intr_leave**() functions in the sanitizer runtime.

EXAMPLES

The following contrived example demonstrates some of the types of bugs that are automatically detected by **KMSAN**:

```
int
f(size_t osz)
{
        struct {
                uint32_t bar;
                uint16_t baz;
                /* A 2-byte hole is here. */
        } foo;
        char *buf;
        size_t sz;
        int error;

        /*
         * This will raise a report since "sz" is uninitialized
         * here.  If it is initialized, and "osz" was left uninitialized
         * by the caller, a report would also be raised.
         */
        if (sz < osz)
                return (1);

        buf = malloc(32, M_TEMP, M_WAITOK);

        /*
         * This will raise a report since "buf" has not been
         * initialized and contains whatever data is left over from the
         * previous use of that memory.
         */
        for (i = 0; i < 32; i++)
                if (buf[i] != ' ')
                        foo.bar++;
        foo.baz = 0;
```

```
        /*
         * This will raise a report since the pad bytes in "foo" have
         * not been initialized, e.g., by memset(), and this call will
         * thus copy uninitialized kernel stack memory into userspace.
         */
        copyout(&foo, uaddr, sizeof(foo));

        /*
         * This line itself will not raise a report, but may trigger
         * a report in the caller depending on how the return value is
         * used.
         */
        return (error);
}
```

SEE ALSO

build(7), busdma(9), copyout(9), KASAN(9), uma(9)

Evgeniy Stepanov and Konstantin Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++", *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

HISTORY

KMSAN was ported from NetBSD and first appeared in FreeBSD 14.0.

BUGS

Accesses to kernel memory outside of the kernel map are ignored by the **KMSAN** runtime. In particular, memory accesses via the direct map are not validated. When memory is copied from outside the kernel map into the kernel map, that region of the kernel map is marked as initialized. When **KMSAN** is configured, kernel memory allocators are configured to use the kernel map, and filesystems are configured to always map data buffers into the kernel map, so usage of the direct map is minimized. However, some uses of the direct map remain. This is a conservative policy which aims to avoid false positives, but it will mask bug in some kernel subsystems.

On amd64, global variables and the physical page array *vm_page_array* are not sanitized. This is intentional, as it reduces memory usage by avoiding creating shadows of large regions of the kernel map. However, this can allow bugs to go undetected by **KMSAN**.

Some kernel memory allocators provide type-stable objects, and code which uses them frequently depends on object data being preserved across allocations. Such allocations cannot be sanitized by

KMSAN. However, in some cases it may be possible to use **kmsan_mark**() to manually annotate fields which are known to contain invalid data upon allocation.