**What is a Kernel Address Sanitizer?**

The Kernel Address Sanitizer or KASAN is a fast and efficient memory error detector designed by developers. It is heavily based on compiler optimization and has been very effective in reporting bugs in the Linux Kernel.

This Sanitizer will help detect a lot of memory errors that otherwise would be hard to detect.

**Design of KASAN and how it works**

Kernel Address Sanitizer works by instrumenting all the memory accesses and having a separate "shadow buffer" to keep track of all the addresses that are legitimate and accessible and complains (Very Descriptively!!) when the kernel reads/writes elsewhere.

The basic idea behind Kernel ASan is to set aside a map/buffer where **each byte in the kernel is represented by using a bit**. This means the size of the buffer would be 1/8th of the total memory accessible by the kernel. In amd64(also x86\_64) this would mean setting aside **16TB of memory to handle a total of 128TB** of kernel memory.

**Implementation Outline**

Kernel Address Sanitizer is useful in finding bugs/coding errors in the kernel such as :

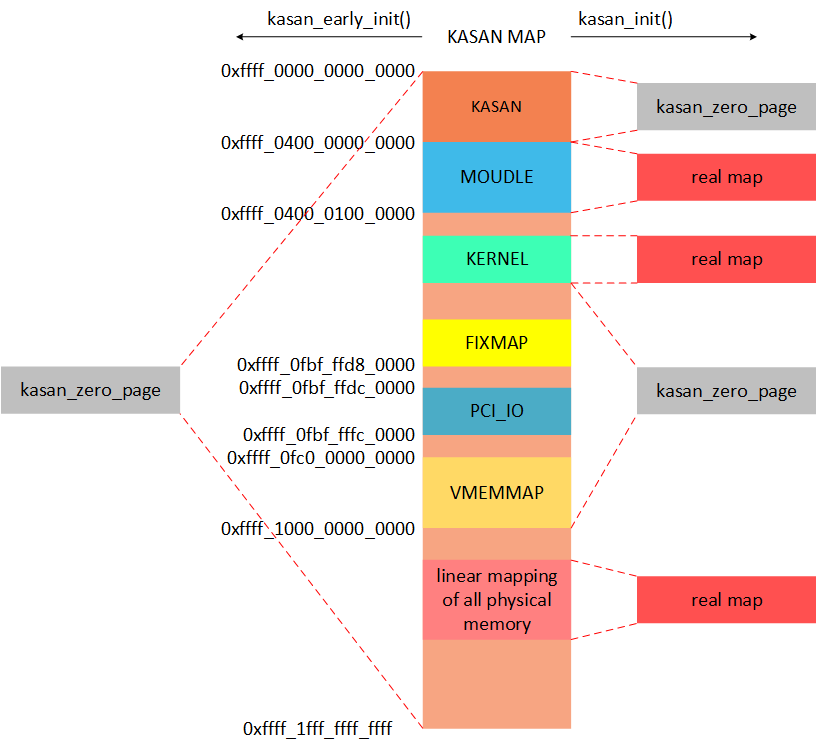
* Use - after - free
* Stack, heap and global buffer overflows
* Double free
* Use - after - scope

The design makes it faster than other tools such as kmemcheck etc. The average slowdown is expected to be around ~2x times or less.

**KASAN Initialisation**

KASAN initialization happens in two stages -

* early in the boot stage, it sets each page entry of the entire shadow region to zero\_page (*early\_kasan\_init*)
* after the physical memory has been mapped and the pmap(9) has been bootstrapped during kernel startup, the zero\_pages are unmapped and the real pages are allocated and mapped (*kasan\_init*).



Below is a short description of what kasan\_init() does in Linux code :

* It loads the kernel boot time page table and clears all the page table entries for the shadow buffer region which had been populated with zero\_pages during early\_kasan\_init.
* It marks shadow buffer offsets of parts of kernel memory; which we don't want to track or are prohibited, by populating them using *kasan\_populate\_zero\_shadow* which iterates through all the page tables.

**Allocating the shadow buffer**

Instead of iterating through the page table entries as Linux preferred to do, it helps to use our low-level kernel memory allocators to do the job for us. This helped in reducing the code complexity and allowed us to reduce the size of the code by a significant amount.

One may ask then does that allocator need to be sanitized? We propose to add a kasan\_inited variable which would help the sanitization to occur after the initialization.

**Shadow translation (Address Sanitizer Algorithm)**

The translation from a memory address to the corresponding shadow offset must be done pretty fast since it happens during every memory read/write. This is implemented similar to the below code  
  
shadow\_address = KmemToShadow(address);  
void \* KmemToShadow(void \* addr) {  
return (addr >> Shadow\_scale) + Shadow\_buffer\_start;  
}

The reverse shadow offsets to kernel memory addresses function is also similar to this.

The shadow translation functions have already been implemented and can be found in kasan.h

**Error Detection**

Every read/write is instrumented to have a check which would decide if the memory access was legitimate or not. This would be done in the manner shown below.  
  
shadow\_address = KmemToShadow(address);  
if (IsPoisoned(shadow\_address)) {  
ReportError(address, Size, IsWrite);  
}

The actual implementation of the Error detection is a bit more complex since we have to include the mapping aspect as well.

Each byte of shadow buffer memory maps to a qword(8 bytes) of kernel memory. Because of which poisoned memory(*\*shadow\_address*) values have only 3 possibilities :

* The value can be 0 ( Meaning that all 8 bytes are unpoisoned )
* The value can be -ve ( Meaning that all 8 bytes are poisoned )
* The value can have first k bits unpoisoned and the rest (8 - k) poisoned

Therefore we can use the value also to help assist us while doing Error detection.

**Basic Bug Report**

The information about each bug is stored in struct kasan\_access\_info which is then used to determine the following information

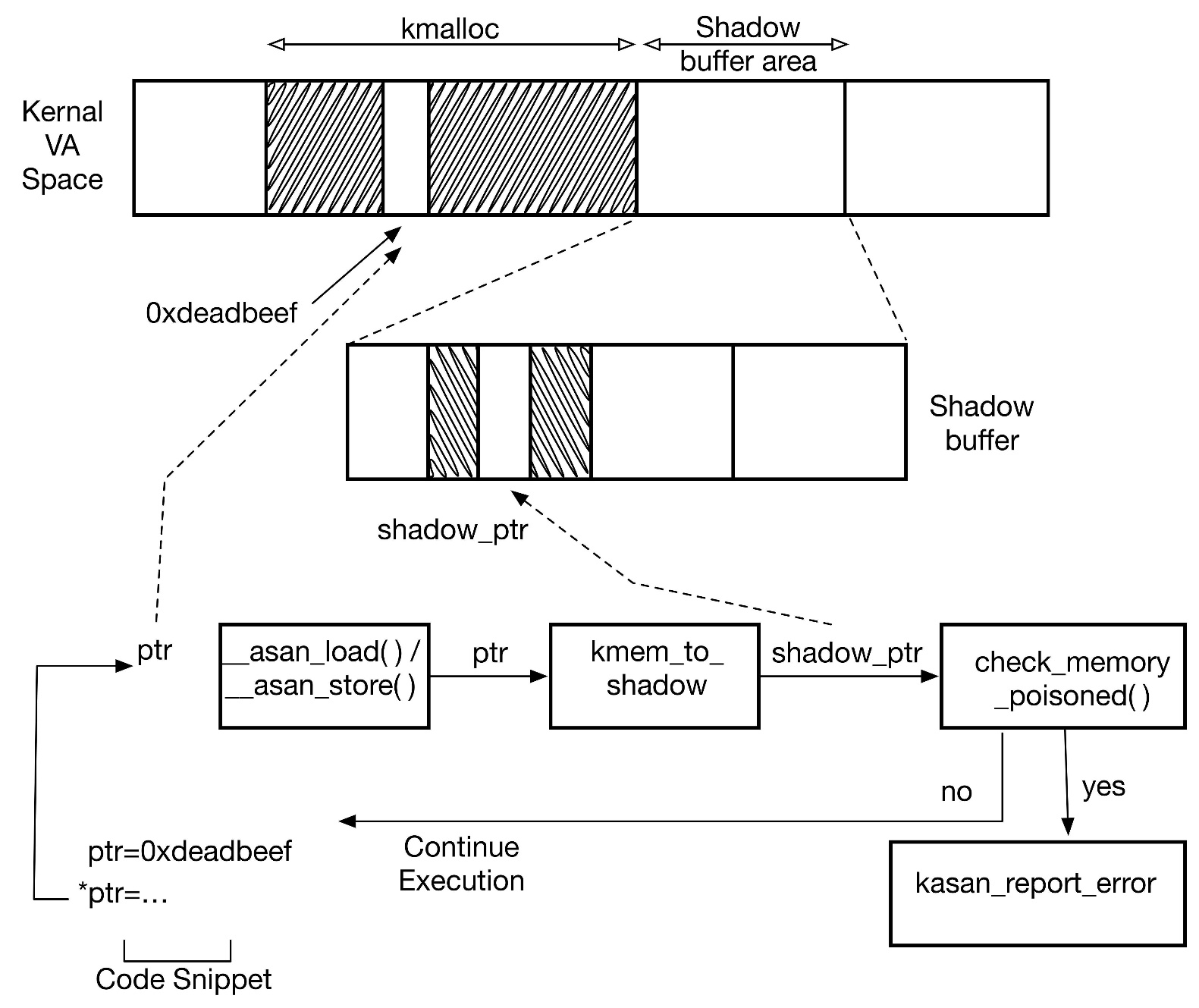
* The kind of bug
* Whether read/write caused it
* Process ID of the task being executed
* The address which caused the error

it also print the stack backtrace which helps in identifying the function with the bug and also helps in finding the execution flow which caused the bug.

One of the best features is that we will be able to use the address where the error occurred to show the poisoning in the shadow buffer. This diagram will be pretty useful for developers trying to fix the bugs found by KASAN.

# Design and principle of KASan

KASan maintains a shadow buffer which is 1/8th the size of the total accessible kernel memory. Each byte of kernel memory is mapped to a bit of shadow memory. Every byte of memory that is allocated and freed is noted in the shadow buffers with the help of the kernel memory allocators.



The above diagram shows how a piece of kernel code gets instrumented by the compiler and the checks that it undergoes.

# Kasan Initialisation

During kernel boot the shadow memory needs to be initialised. This is done after the [pmap(9)](https://man-k.org/man/NetBSD-current/9/pmap?r=1&q=pmap) and the [uvm(9)](https://man-k.org/man/NetBSD-current/9/uvm?r=1&q=uvm) systems have been bootstrapped.

## Linux Initialisation model

Linux initialises the shadow buffer in two step method.

**Kasan\_early\_init** - Function called in main function of the kernel before the MMU is set up.

* Initialises a physical zero page.
* maps the entire top level of the shadow region to a physical zero page.

**Kasan\_init** - Function called in main function of the kernel after the MMU is setup.

* The earlier mapping of the shadow region is cleared.
* Allocation of the shadow buffer takes place
  + Shadow offsets of regions of kernel memory that are not backed up by actual physical memory are mapped to a single zero page.
    - This is done by traversing into the page table and modifying it to point to the zero page.
  + Shadow offsets corresponding to the regions of kernel memory that are backed up by actual physical memory are populated with pages allocated from the same NUMA node
    - This is done by traversing into the page table and allocating memory from the same NUMA node as the original memory by using early alloc.
* The modifications made to the page table are updated.