ITMO Advanced OS 2024

Aleksei Romanovskii, PhD,
SPb Research Center (CBG OS Lab)

Lesson 2024.11.06

Sanitizing the Linux kernel

- What are and why (address) sanitizers are needed?
  - Sanitizers by Google
- KFENCE
  - D.U.M.A
- KASAN

What are and why sanitizers are needed?

- An address/memory/thread/etc/ Sanitizer is a programming tool that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer (use-after-free), data race bugs, etc.
  - Dozens of memory error detection tools are available
    - AddressSanitizer (ASan), LeakSanitizer (LSan), ThreadSanitizer (TSan), UndefinedBehaviorSanitizer (UBSsan), MemorySanitizer (MSan)
- Sanitizers are based on compiler instrumentation. Currently implemented in
  - Clang (version 3.1+),
  - GCC (version 4.8+),
  - Xcode (version 7.0+) and
  - MSVC (version 16.9+).
- On average, the instrumentation and sanitizers increase processing time by about 73% and memory usage by 240%
  - They helped to find over 300 previously unknown bugs in the Chromium browser and many bugs in other software.

Sanitizers by Google: ASan

- Heap-, stack-, and global buffer overflow
- Use-after-free (dangling pointer dereference)
- Use-after-scope -fsanitize-address-use-after-scope
- Use-after-return (pass detect_stack_use_after_return=1 to ASAN_OPTIONS)
- Double free, invalid free
- Initialization order bugs
- ASan-ified binaries may consume 20TB of virtual memory

Sanitizers by Google: ASan example

- int global_array[100] = {-1};

- int main(int argc, char **argv) {
-     return global_array[argc + 100]; // global buffer overflow
- }

- When built with -fsanitize=address -fno-omit-frame-pointer -O1 flags, this program will exit with a non-zero code due to the global buffer overflow detected by ASan

Sanitizers by Google: Leak sanitizer (LSan)

- It is memory leak detector.
- In a stand-alone mode, this Sanitizer is a run-time tool that does not require compiler instrumentation.
- However, LSan is also integrated into AddressSanitizer, so you can combine them to get both memory errors and leak detection.
- To run LSan only (and avoid the ASan's slowdown), use -fsanitize=leak instead of -fsanitize=address
- int main(){
    - int *x = new int(10); // leaked
    - return 0;
- }

Sanitizers by Google: Thread sanitizer (TSan)

- It detects
  - Normal data races
  - Races on C++ object vptr
  - Use after free races
  - Races on mutexes
  - Races on file descriptors
  - Races on pthread_barrier_t
  - Destruction of a locked mutex
  - Leaked threads
  - Signal-unsafe malloc/free calls in signal handlers
  - Signal handler spoils errno
  - Potential deadlocks (lock order inversions)

- Data races occur when multiple threads access the same memory without synchronization and at least one access is a write.

- TSan in Valgrind: 5x–30x slowdown due to the complex race detection algorithm; on heavy web applications the slowdowns were even greater (50x and more)

- TSan in LLVM is much faster than in Valgrind

- To use TSan compile with -fsanitize=thread -fPIE -pie -g

Sanitizers by Google: TSan example

```c
#include <pthread.h>
#include <stdio.h>

int Global;

void *Thread1(void *x) {
    Global++;
    return NULL;
}

void *Thread2(void *x) {
    Global--;
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    pthread_create(&t[1], NULL, Thread2, NULL);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
}
```

Sanitizers by Google: UndefinedBehavoir (UBSan)

- It is a runtime checker for undefined behavior, which is a result of any operation with unspecified semantics, such as
  - dividing by zero,
  - null pointer dereference,
  - usage of an uninitialized non-static variable,
  - etc., see the full list at clang.llvm.org

- One can turn the checks on one by one, or use flags for check groups -fsanitize=undefined, -fsanitize=integer, and -fsanitize=nullability

- int main() {
-     int i = 2048;
-     i <<= 28;
-     return 0;
- }

Sanitizers by Google: Memory Sanitizer (MSan)

- It is a detector of uninitialized memory reads.
- MSan Finds the cases when stack- or heap-allocated memory is read before it is written.
- MSan is also capable of tracking uninitialized bits in a bitfield

- MSan can track back the origins of an uninitialized value to where it was created and report this information.

- Pass the -fsanitize-memory-track-origins flag to enable this functionality.

- To efficiently use MSan, compile your program with -fsanitize=memory -fPIE -pie -fno-omit-frame-pointer -g, add -fno-optimize-sibling-calls and -O1

```
• int main(int argc, char** argv) {
•     int* a = new int[10];
•     a[5] = 0;
•     if (a[argc])
•         std::cout << a[3];
•     return 0;
• }
```

EFENCE

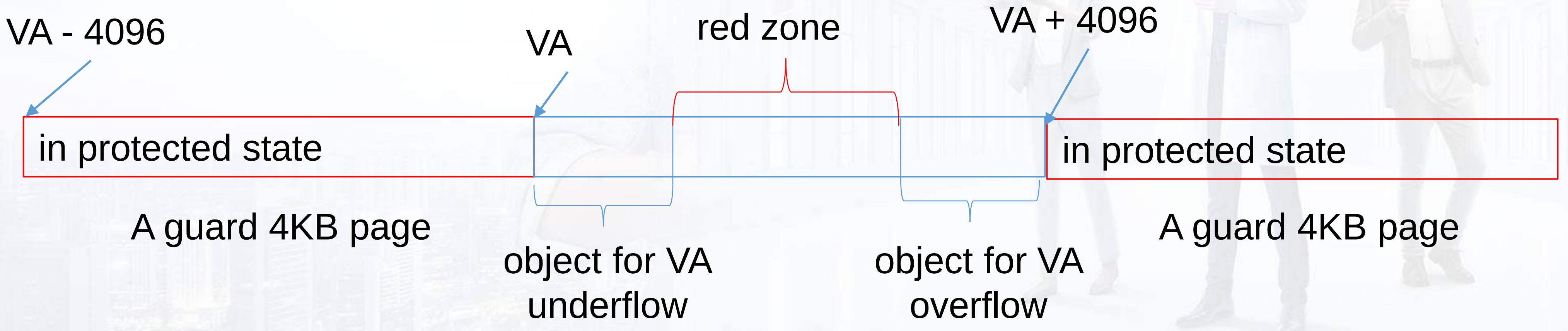
- Electric Fence Malloc Debugger (1987-1999, by Bruce Perens)
  - Bruce Perens is an American computer programmer and advocate in the free software movement.
  - He created The Open Source Definition and published the first formal announcement and manifesto of open source.
- Electric Fence detects two common programming bugs:
  - software that overruns the boundaries of a malloc() memory allocation
  - software that touches a memory allocation that has been released by free()
- Unlike other malloc() debuggers, Electric Fence will detect read accesses as well as writes, and it will pinpoint the exact instruction that causes an error.
- Electric Fence uses the virtual memory hardware (mmu) to place an inaccessible memory page immediately after or before, at the user's option each memory allocation.
- When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction.
- Simply link your application with libefence.a

KFENCE 1 of 4

- KFENCE is a low-overhead sampling-based memory safety error detector of heap
- use-after-free, invalid-free, and out-of-bounds access errors.
- Since Linux kernel 5.12 KFENCE exists for the x86 and arm64 architectures, KFENCE hooks are in SLAB and SLUB allocators.
- KFENCE is inspired by GWP-ASan (**GNU** WP-ASan **W**ill **P**rovide **A**llocation **SAN**ity), a userspace tool with similar properties.
- The name "KFENCE" is a homage to the EFENCE
- KFENCE is designed to be enabled in production kernels, and has near zero performance overhead.
- Compared to KASAN, KFENCE trades performance for precision.
- The main motivation behind KFENCE's design, is that with enough total uptime KFENCE will detect bugs in code paths not typically exercised by non-production test workloads.
- One way to quickly achieve a large enough total uptime is when the tool is deployed across a large fleet of machines.

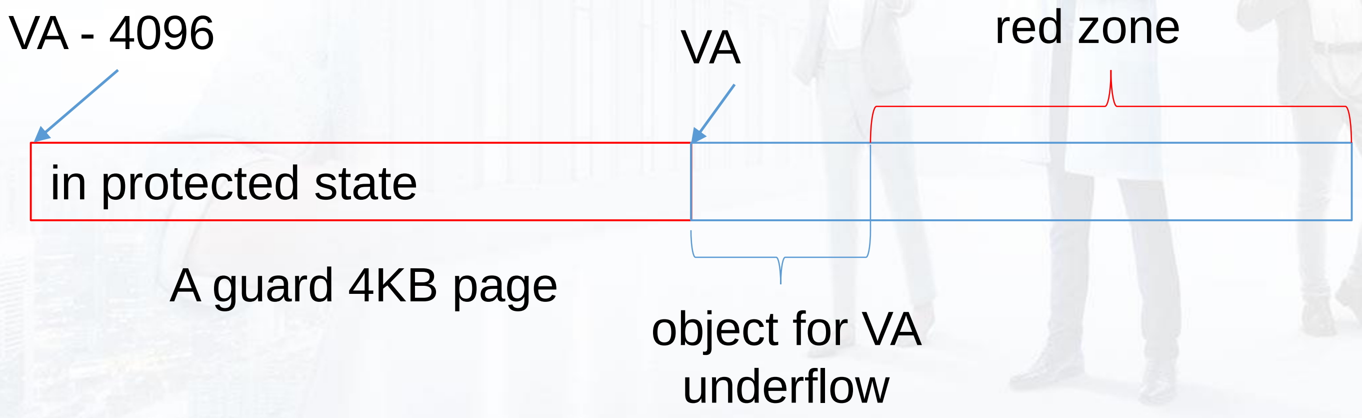
KFENCE 2 of 4

- KFENCE objects each reside on a dedicated page, at either the left or
- right page boundaries.
- The pages to the left and right of the object page are "guard pages", whose attributes are changed to a protected state, and cause page faults on any attempted access to them.
- Such page faults are then intercepted by KFENCE, which handles the fault
- gracefully by reporting a memory access error.
- Each object requires 2 pages, one for the object itself and the other one used as a guard page
- On architectures that support huge pages, KFENCE will ensure that the pool is using pages of size PAGE_SIZE. This will result in additional page tables being allocated.



KFENCE 3 of 4

- To detect out-of-bounds writes to memory within the object's page itself, KFENCE also uses pattern-based redzones.
- For each object page, a redzone is set up for all non-object memory.
- For typical alignments, the redzone is only required on the unguarded side of an object.
- Because KFENCE must honor the cache's requested alignment, special alignments may result in unprotected gaps on either side of an object, all of which are redzoned.
- Upon deallocation of a KFENCE object, the object's page is again protected and the object is marked as freed.
- Any further access to the object causes a fault and KFENCE reports a use-after-free access.
- Freed objects are inserted at the tail of KFENCE's free list, so that the least recently freed objects are reused first, and the chances of detecting use-after-frees of recently freed objects is increased



KFENCE 4 of 4

- To enable KFENCE, configure the kernel with:
  - CONFIG_KFENCE=y
- To build a kernel with KFENCE support, but disabled by default (to enable, set kfence.sample_interval to non-zero value), configure the kernel with:
  - CONFIG_KFENCE=y
  - CONFIG_KFENCE_SAMPLE_INTERVAL=0
- The most important parameter is KFENCE's sample interval, which can be set via the kernel boot parameter kfence.sample_interval in milliseconds.
- The sample interval determines the frequency with which heap allocations will be guarded by KFENCE.
- The sample interval controls a timer that sets up KFENCE allocations.
- By default, to keep the real sample interval predictable, the normal timer also causes CPU wake-ups when the system is completely idle. This may be undesirable on power-constrained systems.
- The KFENCE memory pool is of fixed size, and if the pool is exhausted, no further KFENCE allocations occur.
  - KFENCE objects/pages live in a separate page range and are not to be intermixed with regular heap objects (e.g. KFENCE objects must never be added to the allocator freelists).
- With CONFIG_KFENCE_NUM_OBJECTS (default 255), the number of available guarded objects can be controlled.

KFENCE API

- bool is_kfence_address(const void *addr)
- void kfence_shutdown_cache(struct kmem_cache *s)
- void *kfence_alloc(struct kmem_cache *s, size_t size, gfp_t flags)
- size_t kfence_ksize(const void *addr)
- void *kfence_object_start(const void *addr)
- void __kfence_free(void *addr) - release a KFENCE heap object to KFENCE pool
- bool kfence_free(void *addr) - try to release an arbitrary heap object to KFENCE pool
- bool kfence_handle_page_fault(unsigned long addr, bool is_write, struct pt_regs *regs)

KASAN Intro 1 of 2

- KernelAddressSANitizer (KASAN) is a dynamic memory error detector (ASan ported to kernel)
- It provides a fast and comprehensive solution for finding use-after-free and out-of-bounds bugs.
- KASAN uses compile-time instrumentation for checking every memory access.
  - need a GCC version 4.9.2 or later.
  - GCC 5.0 or later is required for detection of out-of-bounds accesses to stack or global variables.
- To enable KASAN configure kernel with: CONFIG_KASAN = y
- Choose between CONFIG_KASAN_OUTLINE and CONFIG_KASAN_INLINE.
- Outline and inline are compiler instrumentation types. The former produces smaller binary the latter is 1.1 - 2 times faster.
- Inline instrumentation requires a GCC version 5.0 or later.
- KASAN works with both SLUB and SLAB memory allocators.
- For better bug detection and nicer reporting, enable CONFIG_STACKTRACE.

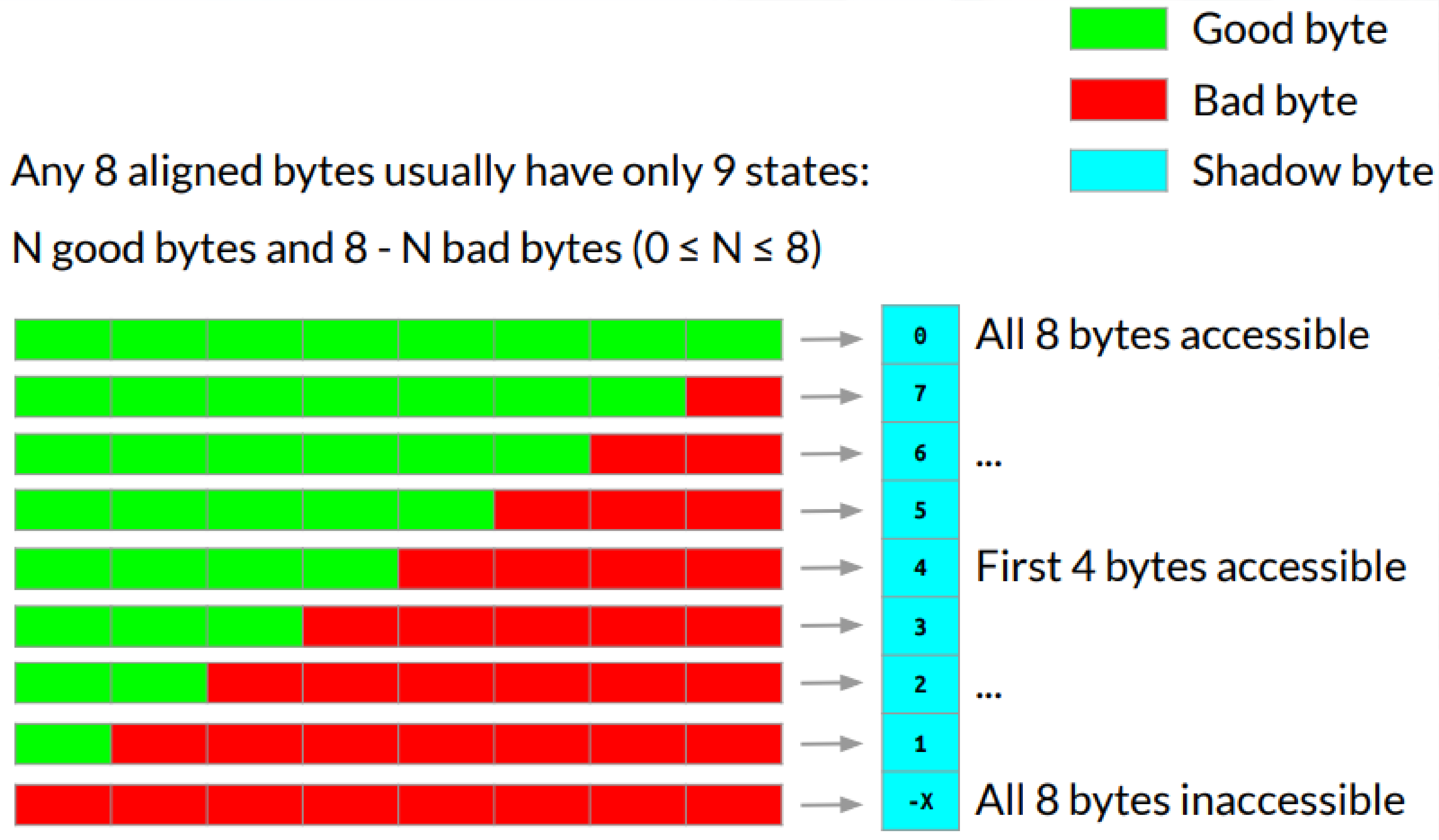
KASAN Intro 2 of 2

- From a high level, our approach to memory error detection is similar to that of kmemcheck:
  - use shadow memory to record whether each byte of memory is safe to access, and use compile-time instrumentation to check shadow memory on each memory access.
- AddressSanitizer dedicates 1/8 of kernel memory to its shadow memory
- (e.g. 16TB to cover 128TB on x86_64)
- It uses mapping with a scale and offset to translate a memory address to its corresponding shadow address.

KASAN parts

- Compiler module (llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp)
  - Instruments memory accesses when building the kernel
  - Inserts redzones for stack and global variables
- Runtime part (mm/kasan/ + include/linux/kasan.h + ...)
  - Maintains shadow memory to track memory state
  - Hooks into kernel allocators to track alloc/free events
  - Prints bug reports

KASAN shadow memory and shadow byte

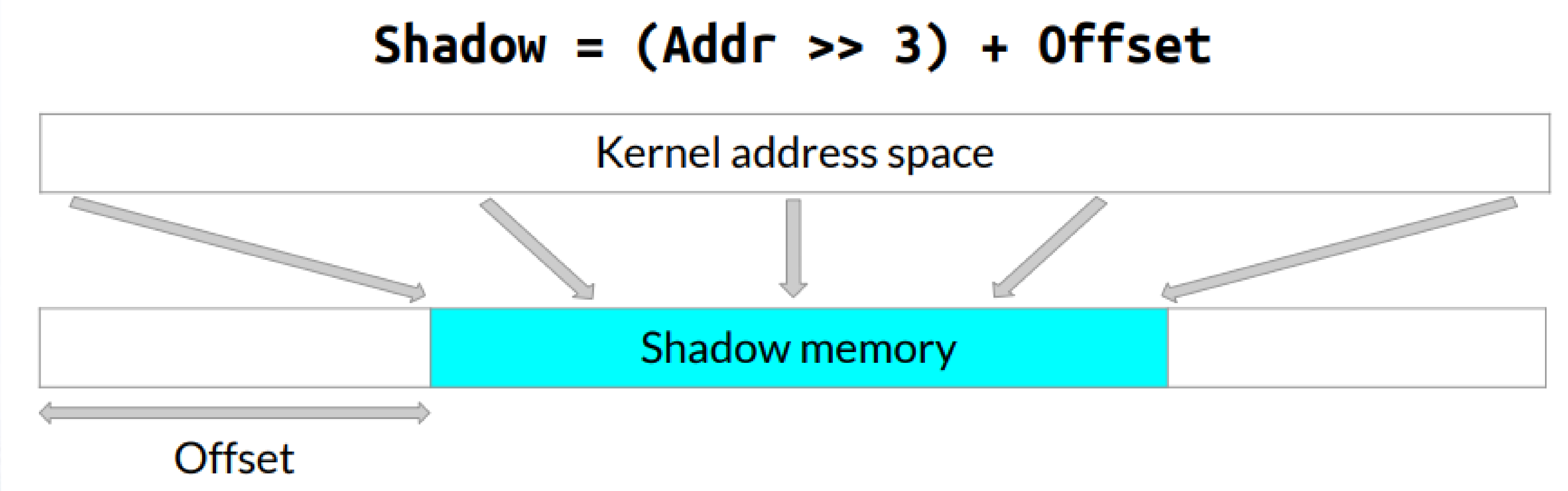


Shadow byte values for inaccessible memory

- #define KASAN_PAGE_FREE 0xFF /* freed page */
- #define KASAN_PAGE_REDZONE 0xFE /* redzone for kmalloc_large allocation */
- #define KASAN_SLAB_REDZONE 0xFC /* redzone for slab object */
- #define KASAN_SLAB_FREE 0xFB /* freed slab object */
- #define KASAN_VMALLOC_INVALID 0xF8 /* inaccessible space in vmap area */
- #define KASAN_SLAB_FREETRACK 0xFA /* freed slab object with free track */
- #define KASAN_GLOBAL_REDZONE 0xF9 /* redzone for global variable */
- #define KASAN_STACK_LEFT 0xF1
- #define KASAN_STACK_MID 0xF2
- #define KASAN_STACK_RIGHT 0xF3
- #define KASAN_STACK_PARTIAL 0xF4

Shadow memory region

- Contains shadow bytes for each mapped region of kernel memory
- Memory-to-shadow mapping scheme:



x86-64 kernel memory layout (4-level page tables)

- ...
- ffff800000000000 | ffff87ffffffffff | 8 TB | ... guard hole, also reserved for hpv.
- ffff880000000000 | ffff887fffffffff | 0.5 TB | LDT remap for PTI
- ffff888000000000 | ffffc87fffffffff | 64 TB | mapping of phys. memory (page_offset_base)
- ffffc88000000000 | ffffc8ffffffffff | 0.5 TB | ... unused hole
- ffffc90000000000 | ffffe8ffffffffff | 32 TB | vmalloc/ioremap space (vmalloc_base)
- ffffe90000000000 | ffffe9ffffffffff | 1 TB | ... unused hole
- ffffea0000000000 | ffffeaffffffffff | 1 TB | virtual memory map (vmemmap_base)
- ffffeb0000000000 | ffffebffffffffff | 1 TB | ... unused hole
- ffffec0000000000 | fffffbffffffffff | 16 TB | KASAN shadow memory

Instrumentation of 8-byte access by compiler

```
*a = ...;
```

```
char *shadow = (a >> 3) + Offset;
if (*shadow)
   kasan_report(a);
*a = ...;
```

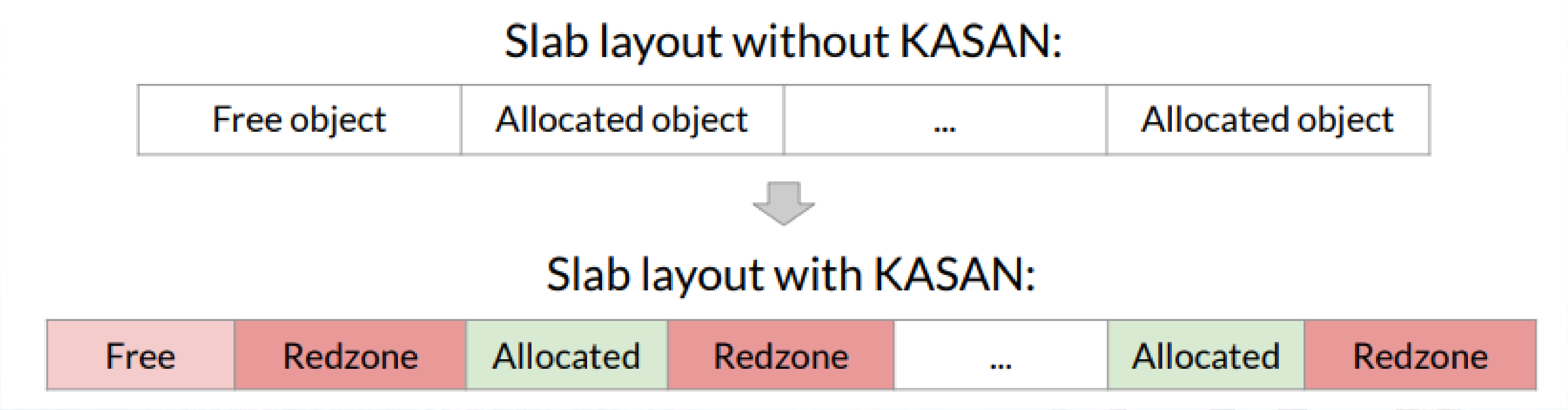
Instrumentation of 1,2,4-byte access by compiler

```
*a = ...;
```

```
char *shadow = (a >> 3) + Offset;
if (*shadow && *shadow < (a & 7) + N)
   kasan_report(a);
*a = ...;
```

Allocation hooks

- KASAN need to keep shadow up-to-date
- This requires tracking of alloc/free events
- KASAN adds hooks to kernel allocators
  - SLUB/SLAB, page_alloc, vmalloc (grep code for "kasan_")



Quarantine for freed memory

- When memory is freed, it's typically immediately reallocated
  - Detecting use-after-free is hard

- KASAN implements quarantine for slab objects
  - Freed objects are not returned to allocator immediately
  - Instead, they are put into a delayed reuse queue
  - Higher chance to detect use-after-free

Compiler instrumentation

- void foo() {
- char x[10];
- <------------- Original function code ------------->
- }

- To

- void foo() {
- char rz1[32];
- char x[10];
- char rz2[22];
- <------------- Original function code ------------->
- }

Generic KASAN summary

- Dynamic memory corruption detector for the Linux kernel
- Finds out-of-bounds, use-after-free, and double/invalid-free bugs
- Supports slab, page_alloc, vmalloc, stack, and global memory
- Requires compiler support: implemented in both Clang and GCC
- google.github.io/kernel-sanitizers/KASAN
- Relatively fast: ~x2 slowdown
- RAM impact: shadow (1/8 RAM) + quarantine (1/32 RAM) + ~x1.5 for slab
- Basic usage: enable and run tests

More sanitizers

- KMSAN — Kernel Memory Sanitizer (un-init mem)
- KCSAN — Kernel Concurrency Sanitizer
- UBSAN — [Kernel] Undefined Behavior Sanitizer