KASan in a Bare-Metal Hypervisor

Alexander Popov

LinuxCon Japan
July 13, 2016

Motivation

- C and C++ are not memory safe
- Buffer overflow and use-after-free bugs can be maliciously exploited
- We want to get rid of such bugs in our C code
- KASan is a great technology, let's use it for PT hypervisor!

Agenda

- Basic ideas behind KASan
- What is a bare-metal hypervisor
- Porting KASan to a bare-metal hypervisor:
  - Main steps
  - Pitfalls
  - How to make KASan checks more strict and multi-purposed
- Bonus

KASan (Kernel Address Sanitizer)

- KASan is a **dynamic** memory error detector for Linux kernel
- Based on work by Andrey Konovalov and other great people at **AddressSanitizer** project, came to kernel from Andrey Ryabinin
- **Trophies:** more than 65 memory errors found in Linux kernel
- **Low penalty:** ~1.5x slowdown, ~2x memory usage
- KASan is a **debugging tool** giving maximum profit with fuzzing
- Can be used in **bare-metal** software

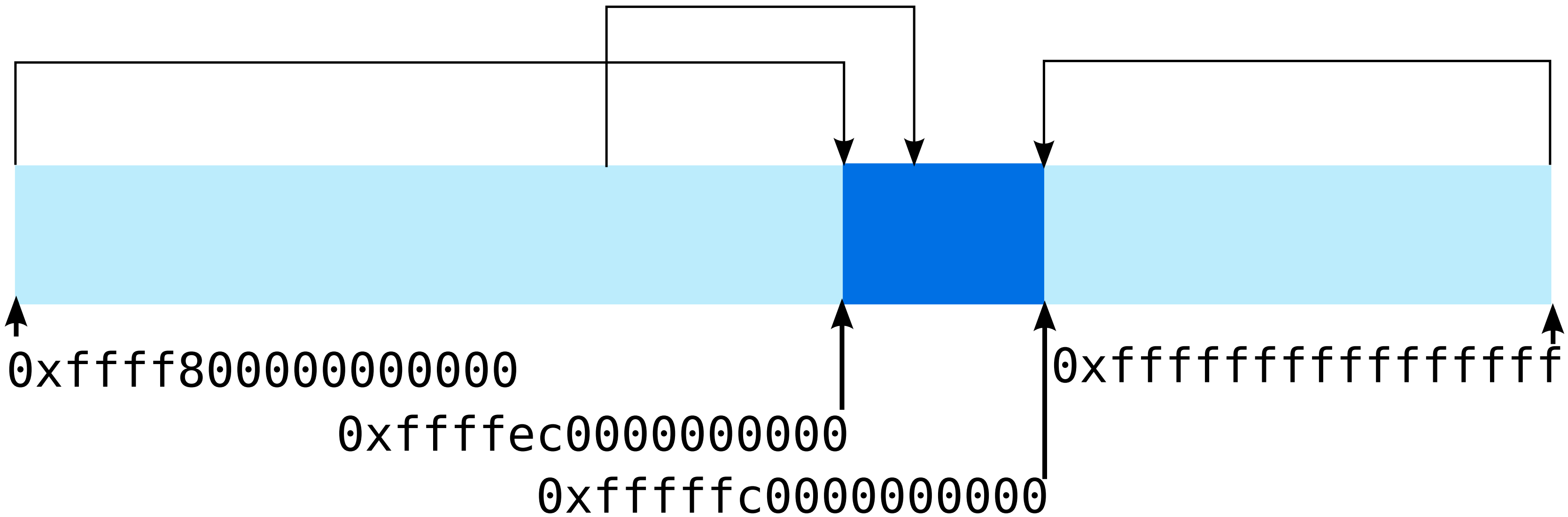
KASan shadow memory legend

Every aligned 8 bytes can have 9 states.

KASan shadow encoding:

- **0** if access to all 8 bytes is valid
- **N** if access only to first N bytes is valid (1 <= N <= 7)
- **Negative value (poison)** if access to all 8 bytes is invalid

Mapping to KASan shadow (x86-64)



Kernel address space (47 bits, 128 TB)

KASan shadow memory (44 bits, 16 TB)

Mappinng:

```
shadow_addr = KASAN_SHADOW_OFFSET + (addr >> 3)
```

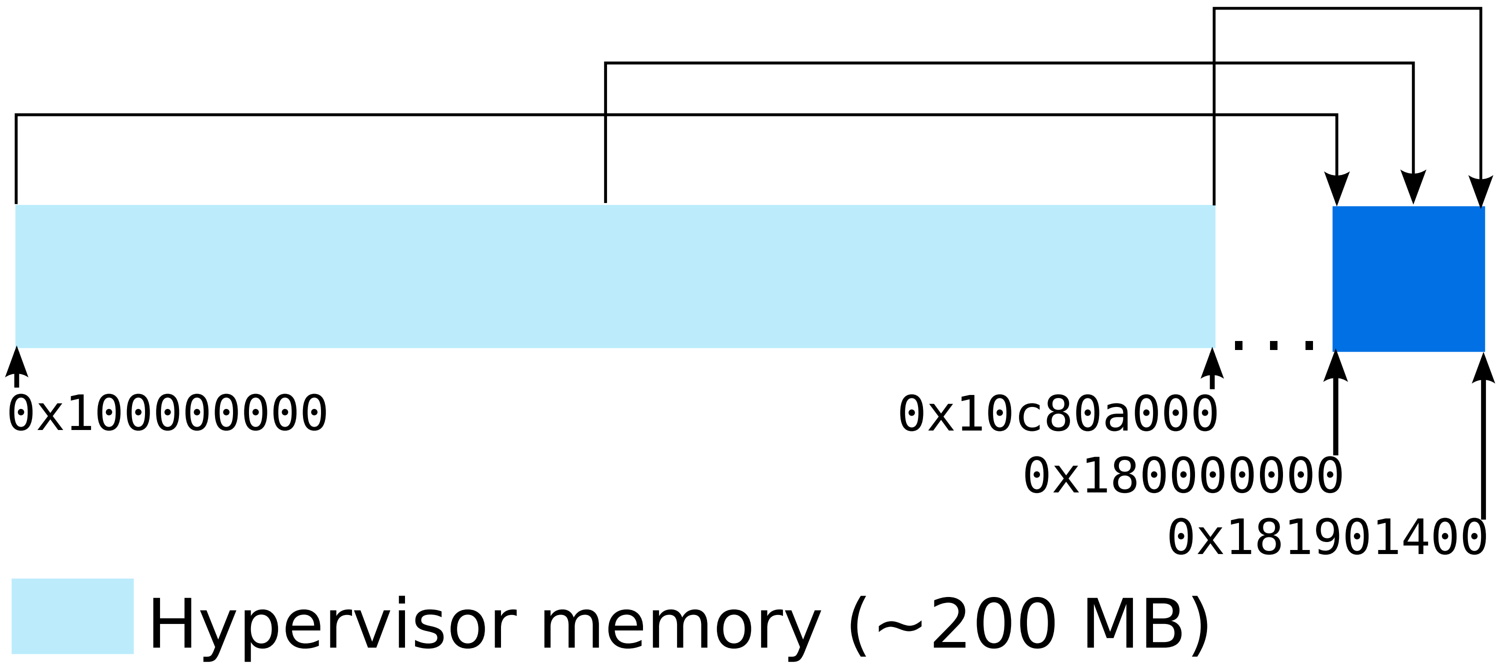
Compile-time instrumentation

- gcc adds calling of `__asan_load##size()` or `__asan_store##size()` before memory access
- gcc adds redzones around stack buffers and globals

A bare-metal hypervisor

- What is a hypervisor
- What does “bare-metal” mean
- How does it work with memory

Step 1: Page tables for shadow



N.B. Choosing KASAN_SHADOW_OFFSET is tricky

N.B. Ability to check whether hypervisor code touches foreign memory

Step 2: Sanitize a single source file

- Specify these gcc flags:
  `-fsanitize=kernel-address`
  `-fasan-shadow-offset=...`
  `--param asan-instrumentation-with-call-threshold=0`

  N.B. The outline instrumentation is easier to start with

  N.B. The build system should support specifying different flags for different source files

- Add KASan implementation from `mm/kasan/kasan.c` little by little (**N.B.** KASan is GPL)

- Experiment till shadow works fine

Step 3: Track global variables

- Additionally specify `--param asan-globals=1`
- Take care of `.ctors` section in the linker script
- Add `do_ctors()` looking at `init/main.c`
- Poison the redzones by negative `KASAN_GLOBAL_REDZONE` in `__asan_register_globals()`
- Use `-fsanitize-sections=...` to instrument globals in all sections
- **N.B.** gcc does not create a KASan constructor for globals declared in assembler

Step 4: Track heap

- Make allocator add redzones around every allocation
- Introduce `kasan_alloc()` which poisons shadow of redzones by `KASAN_HEAP_REDZONE`
- Introduce `kasan_free()` which poisons shadow of freed memory by `KASAN_HEAP_AFTER_FREE`
- If there is a stack of allocators, integrate KASan with each one to find more bugs: **reserved memory != accessible memory**
- Implement delayed freeing, which reduces the probability of missing use-after-free

Step 5: Poison shadow by default

- Fill whole shadow memory by `KASAN_GENERAL_POISON` in `kasan_init()`
- It's a whitelist instead of a blacklist
- A perfectionist sleeps better now :)

Step 6: Track stack

- Additionally specify `--param asan-stack=1`
- When GCC sanitizes stack accesses it works with KASan shadow on its own
- **Pitfall 1:** GCC instruments stack expecting that stack shadow is filled by `0`. A perfectionist is sad.
- **Pitfall 2:** Don't put `kasan_init()` call into a function with local variables.

Step 7: Design a noKASan API

- Allow memory access without KASan checks in:
  - `nokasan_r64()`, `nokasan_w64()` and others
  - `nokasan_memset()`, `nokasan_memcmp()` and others
    - checking the whole region at once
    - avoiding copying the code
    - except `nokasan_snprintf()`, which works with arglist

N.B. Now we can **very carefully** apply this API to the hypervisor code which legitimately works with foreign memory

Steps 8,9,10: Apply to the whole project

- Cover files by KASan gradually
  - Fix memory access bugs
  - Apply noKASan API very carefully

    N.B. Changed memory layout and timings may trigger bugs

    N.B. Thorough code review by the code authors is **vital**
- Move `kasan_init()` as early as possible (not so easy)
- This took me 3 months to do (project size is 55000 SLOC)

Next steps: Continuously support KASan

- Be paranoid, check that KASan is switched on
- Create a test for KASan and run it regularly
- Teach the team how to interpret KASan reports
- Control noKASan API usage

Summary

- KASan has been successfully ported to a bare-metal hypervisor and has found some very tricky memory errors in it
- The new environment allowed to add new features to KASan
- Using KASan in new environments make it better:

  patch to the Linux kernel mainline

  ```
  commit 5d5aa3cfca5cf74cd928daf3674642e6004328d1
  x86/kasan: Fix KASAN shadow region page tables
  ```

- KASan is very helpful for developing

Undefined Behaviour Sanitizer (UBSan)

- UB is a result of executing the code which doesn't have a prescribed behaviour in the language specification
- Why UB is dangerous
- Why UB exists
- The programmers must avoid it, but sometimes they fail
- UBSan can help, even in bare-metal projects!

Porting UBSan to a bare-metal hypervisor

- Specify `-fsanitize=undefined` for a single source file
- Add `__ubsan_handle_*()` stubs
- Experiment with UB and add UBSan implementation little by little looking at `lib/ubsan.c`
- Choose the needed subset of UBSan flags
- Instrument the whole project and run it
- Become **scared** and carefully fix detected UB

Thanks. Questions?

alex.popov@linux.com

alpopov@ptsecurity.com