Google

Exciting compiler flags for kernel security

Florent Revest

December 2025 - *LPC Tokyo*

Agenda

01 **SelSAN**
Profile-Guided Sanitization

02 **Heap isolation**
Compiler hints for SLAB

Google

01

SelSAN

The idea: “Profile-guided sanitization”

Some sanitizers can be expensive.

💡 Applying them to cold paths only can result in wide coverage for a small fraction of the cost

Unproven intuition: cold code is probably less maintained and more likely to have vulnerabilities

Credits to: Vitaly Buka for the LLVM work

-lower-allow-check-percentile-cutoff-hot

Sets a percentile cutoff to distinguish between cold code and hot code in the FDO profile

Compiler builtin resolves to `1` or `0` at compile time depending on where it's called and the FDO profile:

```
bool __builtin_allow_runtime_check(const char* kind)
```

Eg: Used by UBSAN Bounds to optimize away bound checks from hot code

Pros: easy to plumb into an existing FDO setup that is already used, for example, for PGO
Cons: effectiveness depends on the representivity of the FDO payload. Hard to get right for the kernel

`-fsanitize-ignorelist`

Points to a file that contains rules specifying what files/functions should and shouldn't be sanitized:

```
[array-bounds]
src:*
src:block/*=sanitize
```

Pros: can give extra confidence in excluding hot parts from sanitization (***eg***: `kernel/sched/` etc…).
rules can be informed by continuous profiling of the actual production use

Cons: needs manual maintenance to stay in sync with the codebase

Google

02

Heap isolation

The idea: “Heap isolation”

Attackers typically have to turn small memory corruptions into stronger *exploitation primitives*

“*Heap feng-shui*” is the art of calling various `kmalloc()` & `kfree()` to create a favorable heap layout

Eg: allocating a struct with a function pointer where a struct with a write-after-free vuln was located

💡 Isolating objects into various memory areas based on properties of their types can make this harder

Credits to: Marco Elver for both the LLVM and kernel work

Allocation tokens

Clang >22 can infer allocation types by parsing C idioms such as:

```
void *b = kmalloc(sizeof(struct blah), GFP_KERNEL);
struct blah *b = kmalloc(20, GFP_KERNEL);
```

And expose a type **token** based on properties of that type

```
__builtin_infer_alloc_token(expression)
```

Currently supports “type contains a pointer ?” isolation (`typehashpointersplit`)
Could be extended with more complex logic like XNU’s `kmalloc_type()`

CONFIG_TYPED_KMALLOC_CACHES

SLAB (the allocator behind `kmalloc()`) can use this **type token** to isolate objects into different areas

Currently up to 16 areas but could be made configurable (more is more secure but less performant)

Warning: for this to be effective, it needs a solution against cross-cache attacks like `SLAB_VIRTUAL`

Thank You!

Google