Navigating the MTE Landscape:

iOS Memory Protection Deep Dive

OffensiveCon 2026

Navigating the MTE Landscape

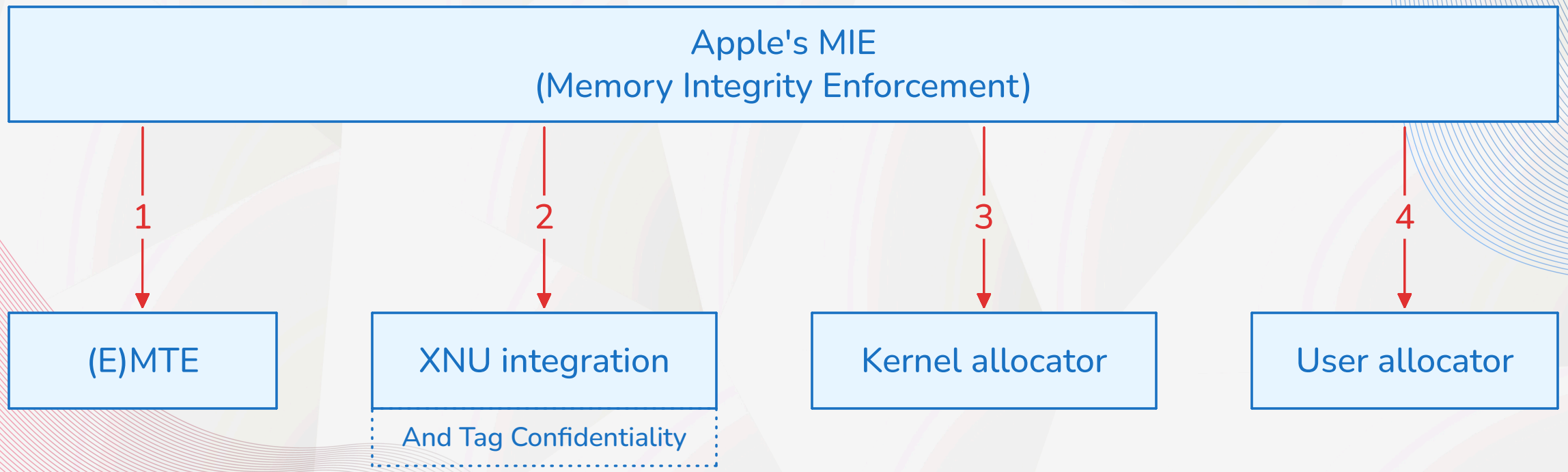
Who Are We?

🪪	Atlan PINABEL
𝕏	_Noiche (mostly inactive)
⛳	Used to play around w/ @MedusHack
💼	iOS Security Researcher and Team Lead @FuzzingLabs

🪪	Patrick VENTUZELO
𝕏	Pat_Ventuzelo
📢	Speaker & trainer at Black Hat, REcon, OffensiveCon, PoC, Pwn2Own 2025
💼	CEO @FuzzingLabs

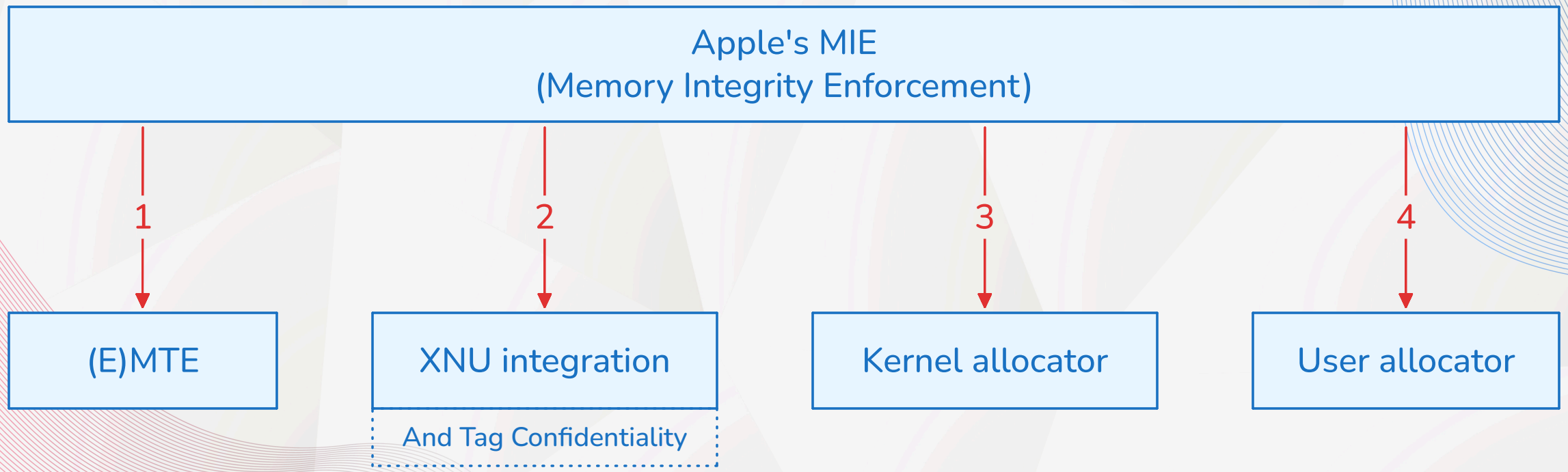
Navigating the MTE Landscape

Outline



Navigating the MTE Landscape

Outline



Disclaimer: There is a lot more to say about MTE and allocators than a 30min talk allows. We will release more material in few days.

Navigating the MTE Landscape

(E)MTE In a Nutshell

(E)MTE In a Nutshell

What is MTE?

- **ARM Memory Tagging Extension** (ARMv8.5, 2019)
  - Hardware-enforced memory corruption mitigation

(E)MTE In a Nutshell

What is MTE?

- **ARM Memory Tagging Extension** (ARMv8.5, 2019)
  - Hardware-enforced memory corruption mitigation
- Associates a **4-bit tag** with every **16-byte granule** of physical memory
  - Associations are stored in a "**tag storage**" (~3% of the DRAM)
  - Using some special ARM **instructions** (IRG, GMI, LDG, STG, ...)
  - Pointers can embed a **4-bit tag** in bits **[59:56]**
  - 15 usable values (0 or F is canonical)

(E)MTE In a Nutshell

What is MTE?

- **ARM Memory Tagging Extension** (ARMv8.5, 2019)
  - Hardware-enforced memory corruption mitigation
- Associates a **4-bit tag** with every **16-byte granule** of physical memory
  - Associations are stored in a "**tag storage**" (~3% of the DRAM)
  - Using some special ARM **instructions** (IRG, GMI, LDG, STG, ...)
  - Pointers can embed a **4-bit tag** in bits **[59:56]**
  - 15 usable values (0 or F is canonical)
- On every tagged memory access, hardware compares **pointer tag** vs **memory tag**
  - Mismatch triggers a **tag check fault** (sync or async)

(E)MTE In a Nutshell

MTE Is Still Being Enhanced

- **ARM Enhanced Memory Tagging Extension** (ARMv8.9, 2022)
  - Extends MTE with new features (i.e., `FEAT_MTE4`)

(E)MTE In a Nutshell

MTE Is Still Being Enhanced

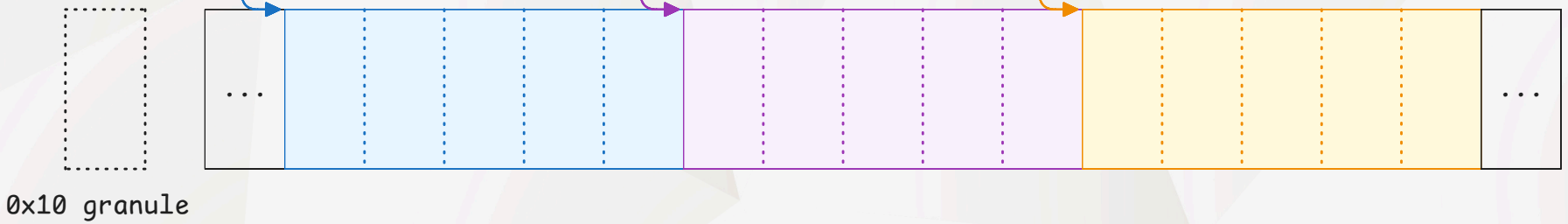
- **ARM Enhanced Memory Tagging Extension** (ARMv8.9, 2022)
  - Extends MTE with new features (i.e., `FEAT_MTE4`)
- Canonical tag checking (can't access **untagged** memory from **tagged** pointers)
  - Will protect global access from tagged pointer
- Tag permission (deny `STG` / `LDG` through untagged **PTEs**)
- Other improvements like Checked Pointer Arithmetic (`FEAT_CPA` , ARMv9.5)

(E)MTE In a Nutshell

Example: MTE Defeating Overflow / OOB

```
__uint128_t *c1, *c2, *c3;

c1 = malloc(0x50);
c2 = malloc(0x50);
c3 = malloc(0x50);
```

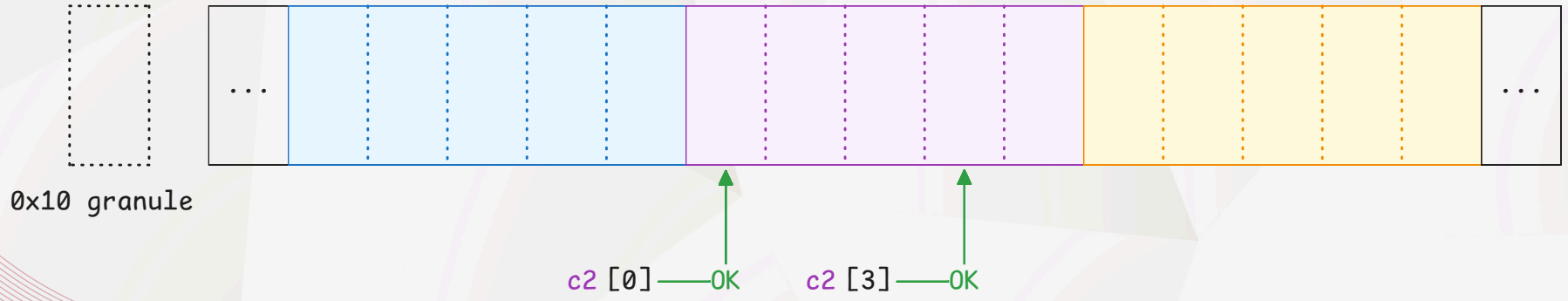


Tag storage
(tag:granule associations, 1:1 mapping)

(E)MTE In a Nutshell

Example: MTE Defeating Overflow / OOB

```
__uint128_t *c1, *c2, *c3;
```

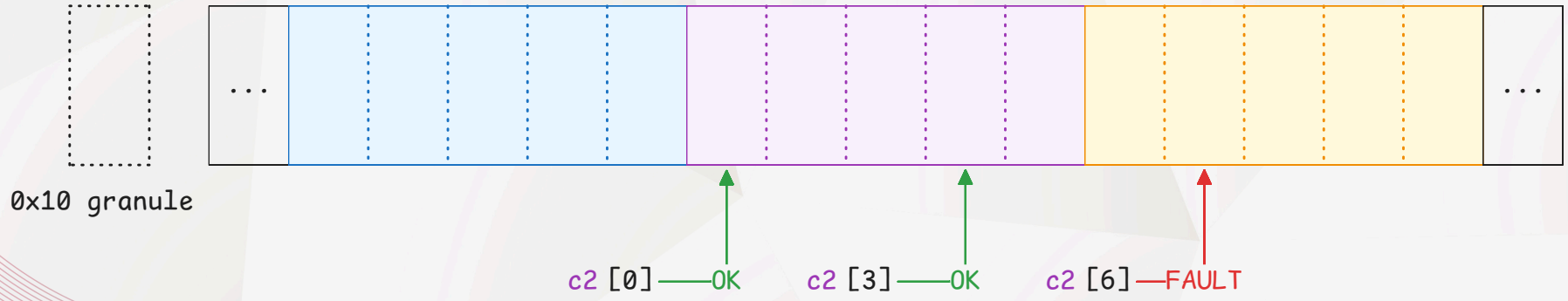


Tag storage
(tag:granule associations, 1:1 mapping)

(E)MTE In a Nutshell

Example: MTE Defeating Overflow / OOB

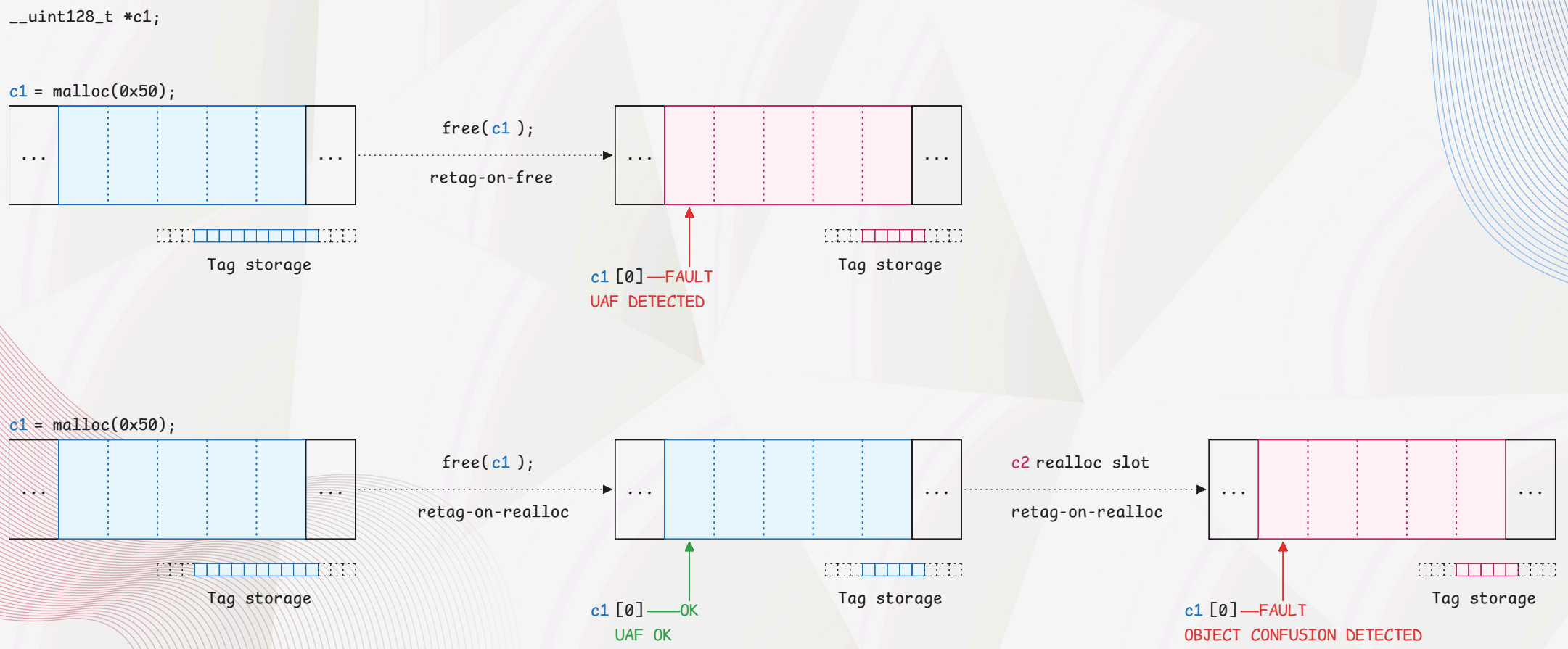
```
__uint128_t *c1, *c2, *c3;
```



Tag storage
(tag:granule associations, 1:1 mapping)

(E)MTE In a Nutshell

Example: MTE Defeating UAF / Type Confusion



Navigating the MTE Landscape

MTE Integration in XNU

MTE Integration in XNU

Apple's Memory Integrity Enforcement

- Shipped with **iPhone 17** (A19) and **M5** macs (EMTE in **synchronous mode only**)

MTE Integration in XNU

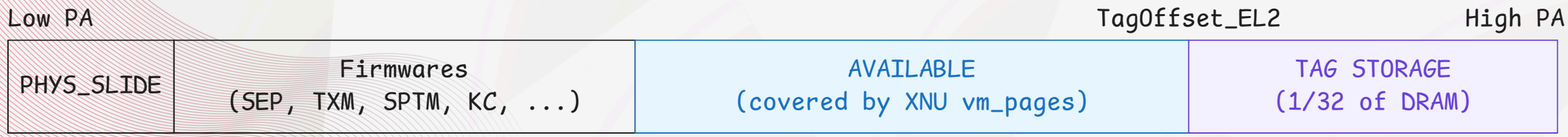
Apple's Memory Integrity Enforcement

- Shipped with **iPhone 17** (A19) and **M5** macs (EMTE in **synchronous mode only**)
- Integrated into memory allocators

MTE Integration in XNU

Apple's Memory Integrity Enforcement

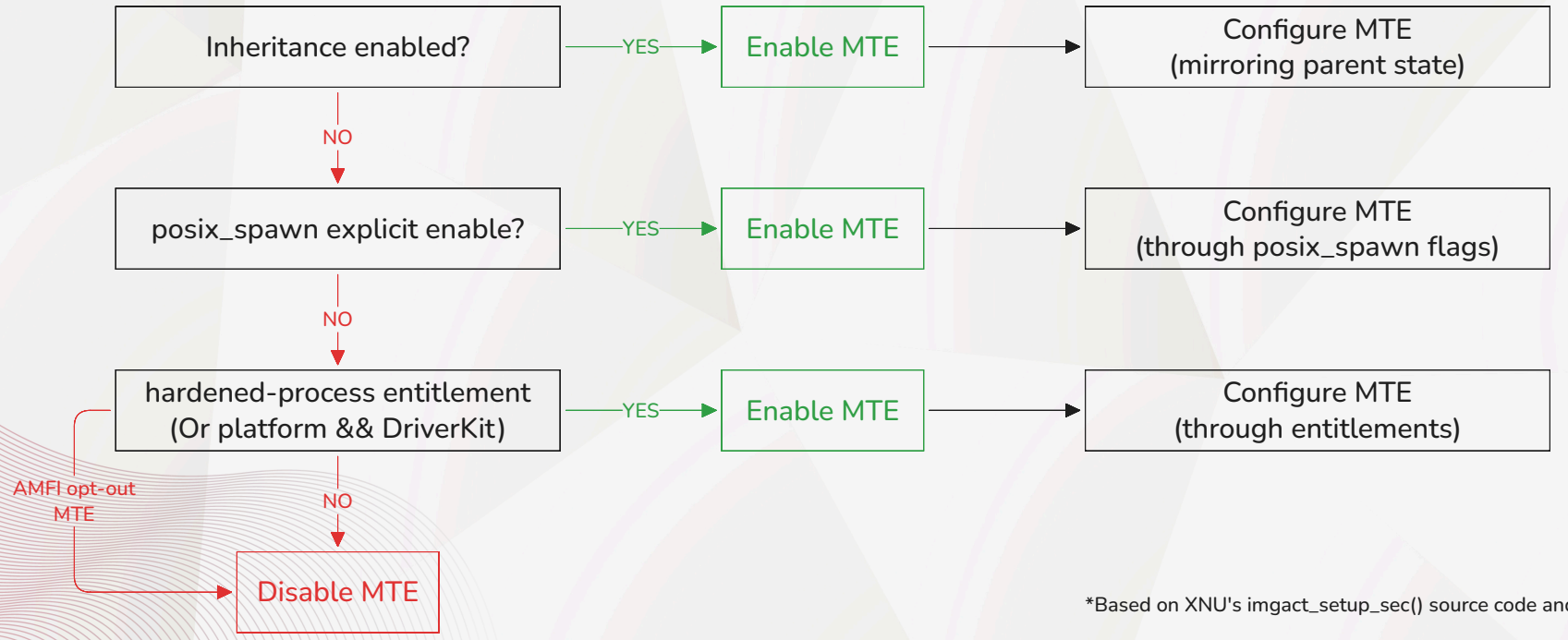
- Shipped with **iPhone 17** (A19) and **M5** macs (EMTE in **synchronous mode only**)
- Integrated into memory allocators
- **Tag Confidentiality Enforcement**
  - Tag generation PRNG reseeded at every context switch (`PACGA_IRG_RESEED` macro)
  - Tag storage is protected by **SPTM** (typed as `XNU_TAG_STORAGE`)
  - SoC designed to be side-channel resistant (TikTag, StickyTag, Spectre V1, ...)



MTE Integration in XNU

Activating MTE — Per-Process Decision

MTE is **per-process**, decided at `exec` time in XNU's `imgact_setup_sec()` :



*Based on XNU's imgact_setup_sec() source code and comment

MTE Integration in XNU

Activating MTE — Task Property & Entitlements

Entitlement	Effect
com.apple.security.hardened-process	Enable **MTE** (platform binaries only)
com.apple.security.hardened-process.checked-allocations	Enable **MTE** explicitly
...checked-allocations.soft-mode	"Soft" mode = non-fatal faults Ignored for platform binaries in **lockdown** mode **Forced** for third-party binaries (until *26.4* kicked in ...)
...checked-allocations.{enable,disable}-pure-data	Tag data-only allocations too (opt-out since *26.4* ...)
...	...

> *There are boot-args (for kernel) and posix_spawn flags equivalent.*

MTE Integration in XNU

Activating MTE — Task Property

- When **activating MTE**, XNU will:
  - Set `task->security_config.sec` to `true`
  - Set `task->task_sec_policy` to represent MTE configuration
  - Inject `has_sec_transition=1` as Apple boot arg string
- When needed, XNU can then check a task's **MTE configuration**
  - Using `task_has_sec_*()`

MTE Integration in XNU

Activating MTE — VM Layer (user land)

- A **MTE-capable task** is allowed to use `VM_FLAGS_MTE` when asking XNU for memory
  - MTE will be enforced at every level in internal XNU VM structures
  - The associated **PTE** itself will indicate that the page can be tagged

MTE Integration in XNU

Activating MTE — VM Layer (user land)

- A **MTE-capable task** is allowed to use `VM_FLAGS_MTE` when asking XNU for memory
  - MTE will be enforced at every level in internal XNU VM structures
  - The associated **PTE** itself will indicate that the page can be tagged
- Any **MTE-capable memory allocator** will use such pages to carve blocks

MTE Integration in XNU

Activating MTE — VM Layer (user land)

- A **MTE-capable task** is allowed to use `VM_FLAGS_MTE` when asking XNU for memory
  - MTE will be enforced at every level in internal XNU VM structures
  - The associated **PTE** itself will indicate that the page can be tagged
- Any **MTE-capable memory allocator** will use such pages to carve blocks
- Tagged mappings suffer limitations about **CoW** and cross-process **sharing** to prevent tag leakage
  - No CoW allowed for tagged memory
  - Inheritance or OOL mach message will force a copy, tags are stripped

> *But the codebase is still evolving ...*

Kernel Allocators

Activating MTE — VM Layer (kernel land)

Layer	API	Role	Example of caller
VM	`vm_page_grab*()`	Physical page allocation and management *can provide tagged page* (`VM_PAGE_GRAB_MTE`)	PMAP layer, `vm_page_alloc_list`
VM	`vm_page_alloc_list()`	Wraps `vm_page_grab` for batch allocation *can provide tagged page* (`KMA_TAG`)	KMEM, `kernel_memory_populate*`
kmem	`kmem_alloc*()`	Huge wired allocations (page-granular) *can provide tagged page* (`KMA_TAG`)	Every kernel component for allocations whose size > **32KB**

Kernel Allocators

Activating MTE — VM Layer (kernel land)

Layer	API	Role	Example of caller
VM	`vm_page_grab*()`	Physical page allocation and management *can provide tagged page* (`VM_PAGE_GRAB_MTE`)	PMAP layer, `vm_page_alloc_list`
VM	`vm_page_alloc_list()`	Wraps `vm_page_grab` for batch allocation *can provide tagged page* (`KMA_TAG`)	KMEM, `kernel_memory_populate*`
kmem	`kmem_alloc*()`	Huge wired allocations (page-granular) *can provide tagged page* (`KMA_TAG`)	Every kernel component for allocations whose size > **32KB**

`VM_PAGE_GRAB_MTE` / `KMA_TAG` are only used through `exclaves_memory_alloc` , `kalloc_large` and **Zalloc** APIs

Navigating the MTE Landscape

Kernel Allocators

Kernel Allocators

XNU Heap Architecture Overview

Layer	API	Role
zalloc	`zalloc()` , `zalloc_flags()`	Fixed-sized zones (SLAB, up to 32KB), *enforcing MTE policy*
kalloc / kalloc_type	`kalloc_ext()` , `kalloc_type(T)`	Generic kernel subsystem allocations, type segregation
IOKit	`IOMalloc()`	Wraps `kheap_alloc` --> `kalloc_ext`

The **kalloc** APIs will allocate from either a dedicated **zalloc zone** or `kmem_alloc_guard` with `KMA_TAG` .

> *If you don't know about kalloc_type: Towards the next generation of XNU memory safety: kalloc_type*

Kernel Allocators

Zalloc Layer — Design (1/2)

- **Fixed-size slab allocator**: each zone serves a single element size
  - Up to **690** zones
  - Reached by indexing `zone_array[i]`

Kernel Allocators

Zalloc Layer — Design (1/2)

- **Fixed-size slab allocator**: each zone serves a single element size
  - Up to **690** zones
  - Reached by indexing zone_array[i]
- Some are compile-time reserved security-critical zones
  - IPC and VM vital structures, thread, proc, cred, sandbox, ...

Kernel Allocators

Zalloc Layer — Design (1/2)

- **Fixed-size slab allocator**: each zone serves a single element size
  - Up to **690** zones
  - Reached by indexing `zone_array[i]`
- Some are compile-time reserved security-critical zones
  - IPC and VM vital structures, thread, proc, cred, sandbox, ...
- Dynamically registered zones (kalloc / kalloc_type, skmem, ...)

Kernel Allocators

Zalloc Layer — Design (1/2)

- **Fixed-size slab allocator**: each zone serves a single element size
  - Up to **690** zones
  - Reached by indexing zone_array[i]
- Some are compile-time reserved security-critical zones
  - IPC and VM vital structures, thread, proc, cred, sandbox, ...
- Dynamically registered zones (kalloc / kalloc_type, skmem, ...)
- Zalloc is already well **documented** (and hardened)

Kernel Allocators

Zalloc Layer — Design (2/2)

Defense	Targets
External bitmap	No inline metadata to corrupt, double-free detection
Heap separation (DATA / PTR / SHARED)	Cross-type exploitation
VA sequestering	Cross-zone attacks (VA pinned to zone forever)
Sad Feng Shui **Per-CPU anti-LIFO magazines**	Predictable heap feng shui
zone_require()	Validates zone ownership at runtime
Guard pages (random ~25%)	Linear overflows between zone chunks
Type segregation (`kalloc_type`)	Type confusion via heap spraying

Kernel Allocators

Zalloc Layer — Attacker Viewpoint

- **zalloc** hardening + `kalloc_type` = hard to build **reliable early primitives**
  - Layout is difficult to predict and Overflow / OOB might face **guard pages**
  - Turning a **UAF** into a meaningful **Type Confusion** is almost impossible
  - You can't abuse **Double Free** either

Kernel Allocators

Zalloc Layer — Attacker Viewpoint

- **zalloc** hardening + `kalloc_type` = hard to build **reliable early primitives**
  - Layout is difficult to predict and Overflow / OOB might face **guard pages**
  - Turning a **UAF** into a meaningful **Type Confusion** is almost impossible
  - You can't abuse **Double Free** either
- Before **MTE** you could still turn a **UAF** into an **Object Confusion** (same type)
  - Leading to state / resource / indirect confusion
  - Think about memory descriptors, mach ports, ...

Kernel Allocators

Zalloc Layer — MTE Integration

- Tagging activated via `z_tag` bit in a zone's `zone_security_flags_t` (defaults to 1 except for RO / DATA)
  - Forces a zone to init/expand via `KMA_TAG` and tag blocks

Kernel Allocators

Zalloc Layer — MTE Integration

- Tagging activated via `z_tag` bit in a zone's `zone_security_flags_t` (defaults to **1** except for RO / DATA)
  - Forces a zone to init/expand via `KMA_TAG` and tag blocks
- `z_tag` field replaced by runtime `zone_submap_has_tagging_enabled` since **26.4**
  - Still avoid tagging for **RO** submap, but can enable tagging for **DATA**
  - `KHEAP_ID_DATA_BUFFERS` is now split in `KHEAP_ID_DATA_PRIVATE` and `KHEAP_ID_DATA_SHARED`
  - Private data get tagged, shared don't

Kernel Allocators

Zalloc Layer — MTE Integration

- Tagging activated via `z_tag` bit in a zone's `zone_security_flags_t` (defaults to **1** except for RO / DATA)
  - Forces a zone to init/expand via `KMA_TAG` and tag blocks
- `z_tag` field replaced by runtime `zone_submap_has_tagging_enabled` since **26.4**
  - Still avoid tagging for **RO** submap, but can enable tagging for **DATA**
  - `KHEAP_ID_DATA_BUFFERS` is now split in `KHEAP_ID_DATA_PRIVATE` and `KHEAP_ID_DATA_SHARED`
  - Private data get tagged, shared don't
- **Tag-on-free policy** across the full element lifecycle
  - When you **allocate** a block, it is **tagged already** and ready to serve your request
  - But then you need to **initialize** block tags when a new page is grabbed (init/expand)

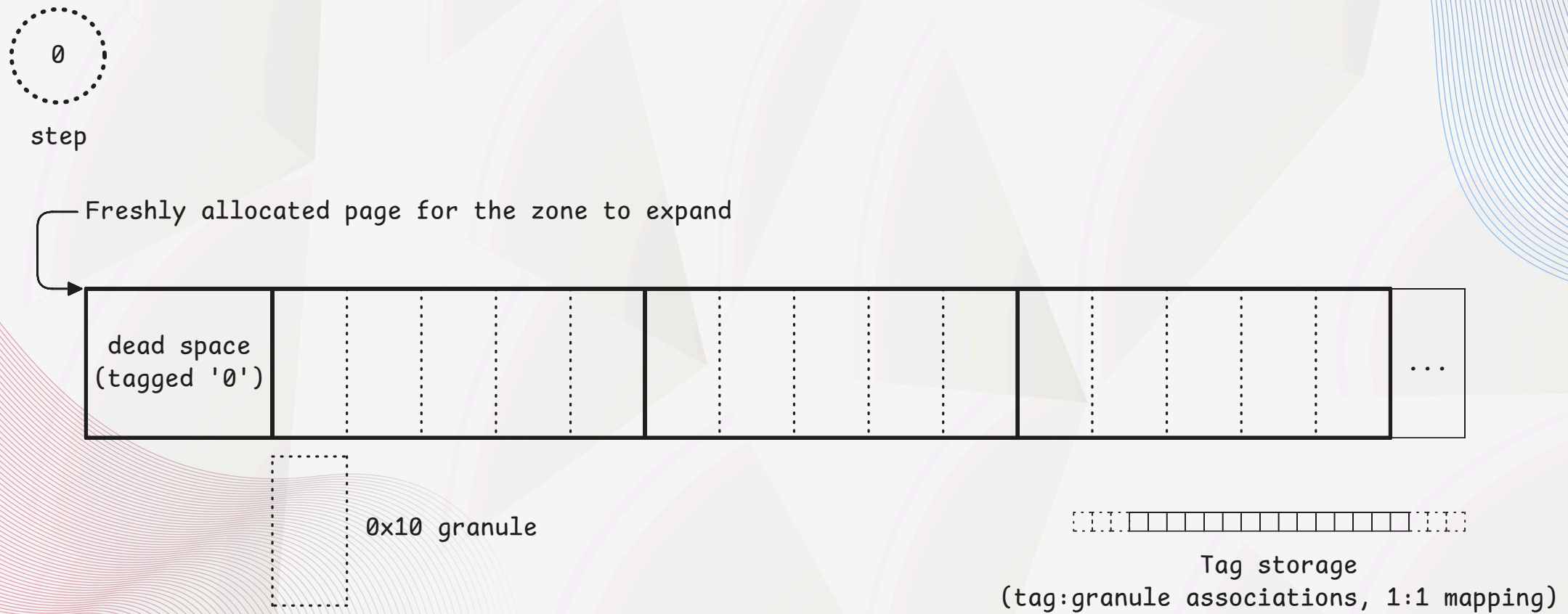
Kernel Allocators

Zalloc Layer — Zone Init & Expansion (since 26.4)

```
static inline void zcram_memtag_init(zone_t zone, vm_offset_t base, uint32_t start, uint32_t end)
{
    if (!zone_submap_has_tagging_enabled(zone_security_config(zone))) { return; }
    /* ... */
    vm_size_t elem_size = zone_elem_outer_size(zone);
    vm_size_t oob_offs  = zone_elem_outer_offs(zone);
    /* ... */
    for (uint32_t i = start; i < end; i++) {
        vm_offset_t addr = base + oob_offs + i * elem_size;
#if HAS_MTE
        addr = (vm_offset_t)mte_generate_and_store_tag((caddr_t)addr,
            elem_size, zone_mte_exclusion_mask((zone->z_chunk_elems - i) & 1));
    /*...*/
    }
}
```

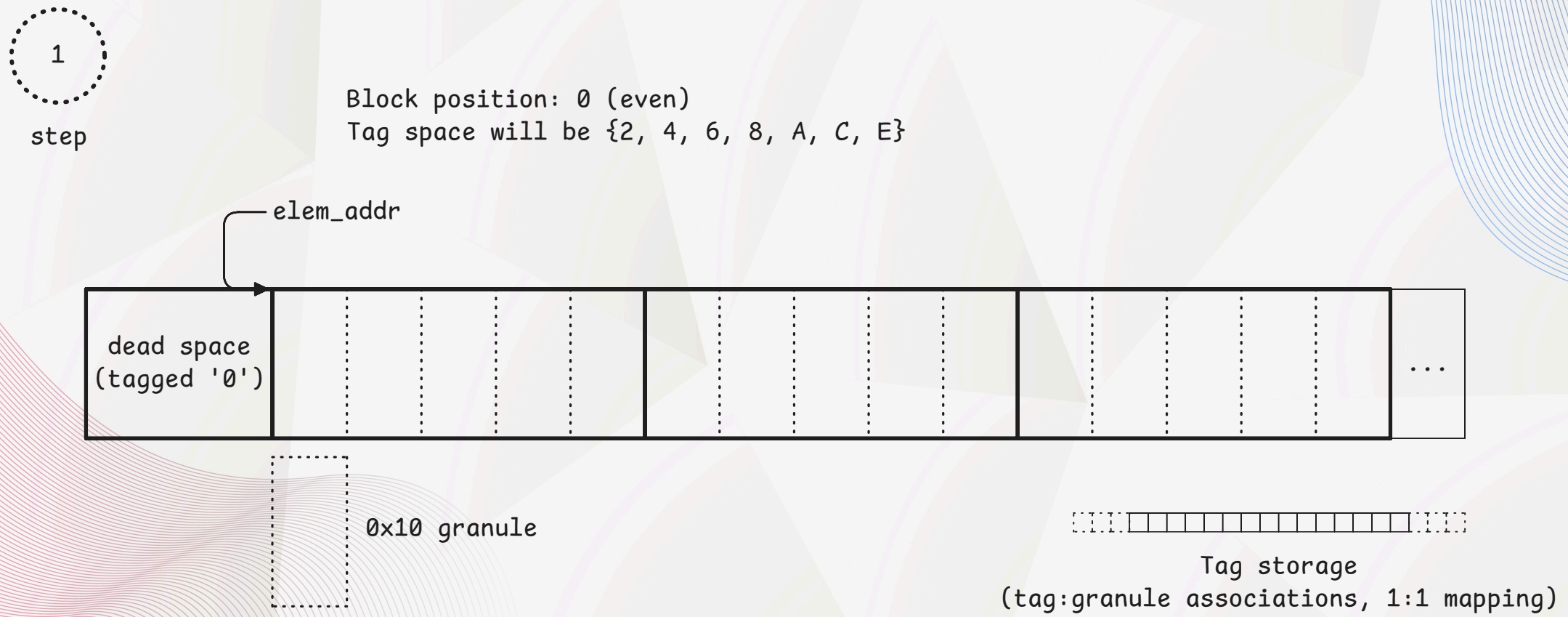
Kernel Allocators

Zalloc Layer — Zone Init & Expansion (since 26.4)



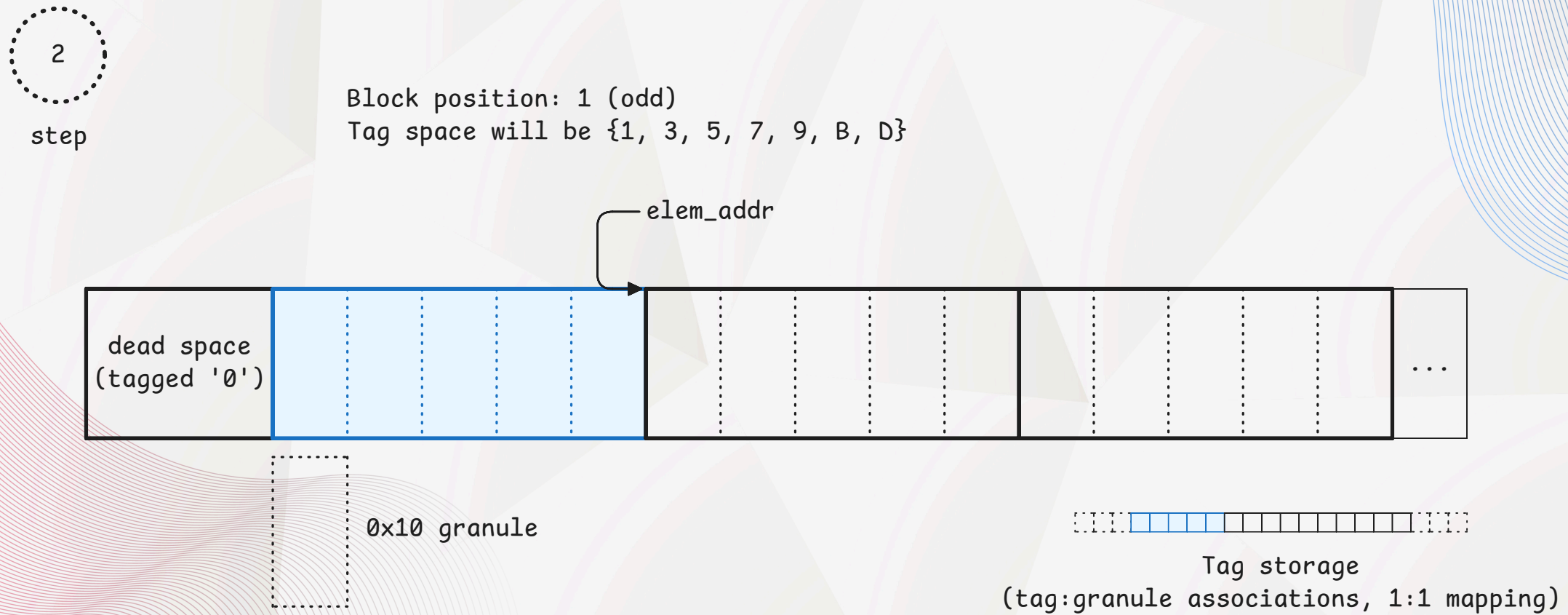
Kernel Allocators

Zalloc Layer — Zone Init & Expansion (since 26.4)



Kernel Allocators

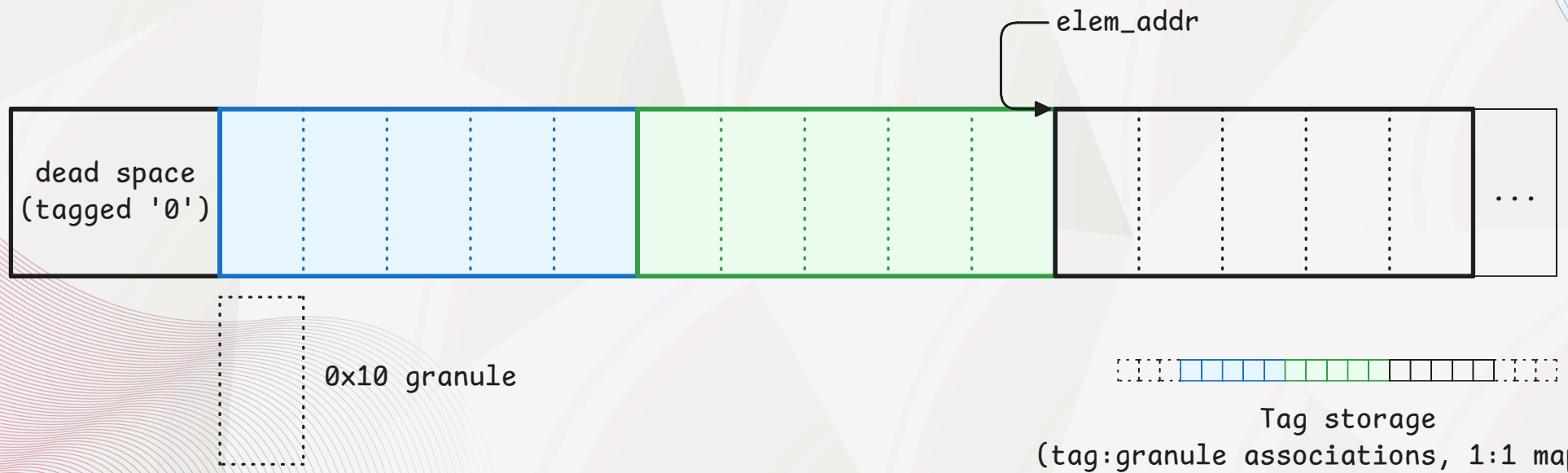
Zalloc Layer — Zone Init & Expansion (since 26.4)



Kernel Allocators

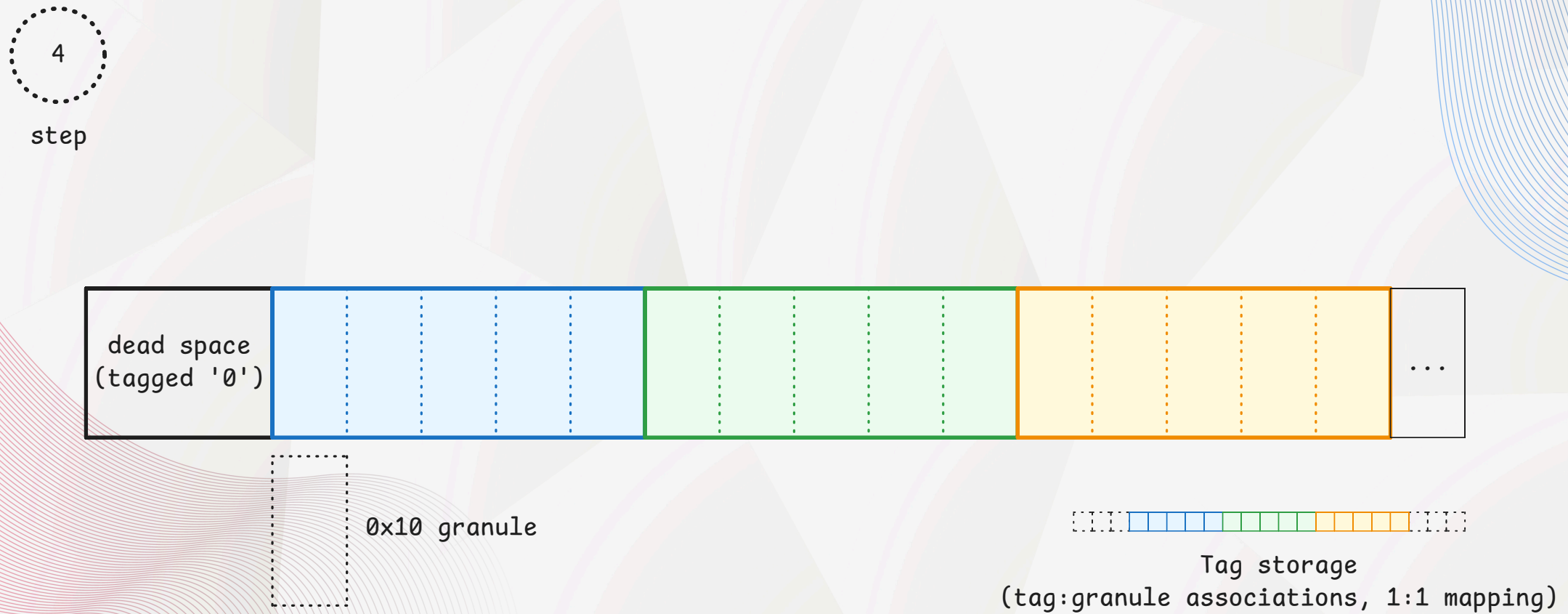
Zalloc Layer — Zone Init & Expansion (since 26.4)

Block position: 2 (even)
Tag space will be {2, 4, 6, 8, A, C, E}



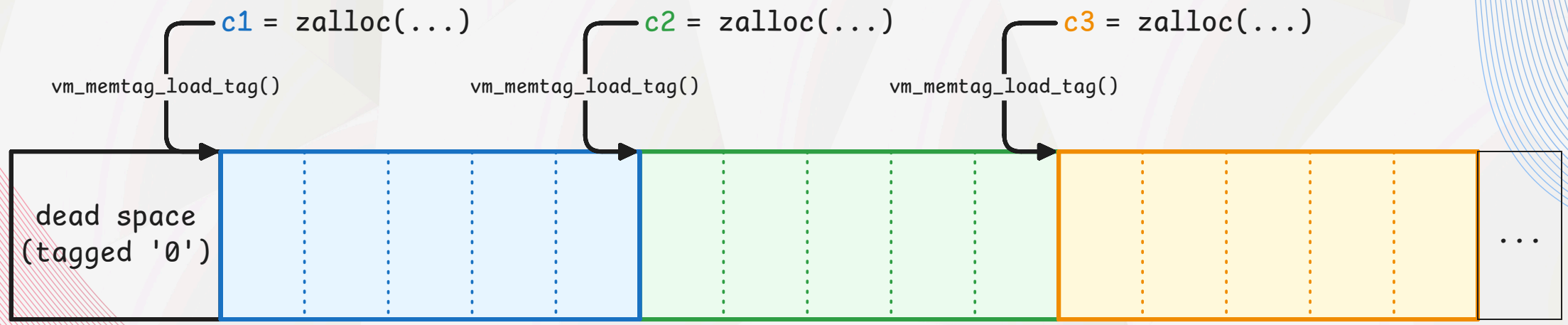
Kernel Allocators

Zalloc Layer — Zone Init & Expansion (since 26.4)



Kernel Allocators

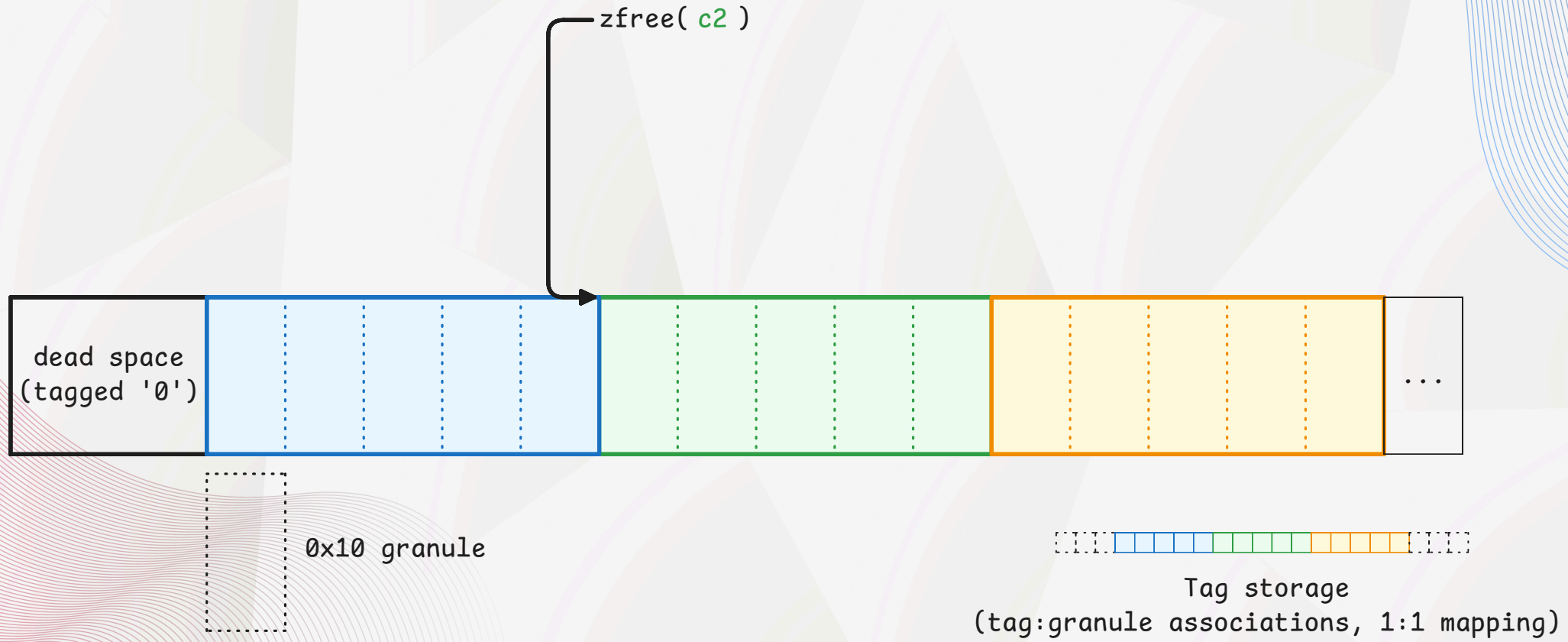
Zalloc Layer — Zone Init & Expansion (since 26.4)



Tag storage
(tag:granule associations, 1:1 mapping)

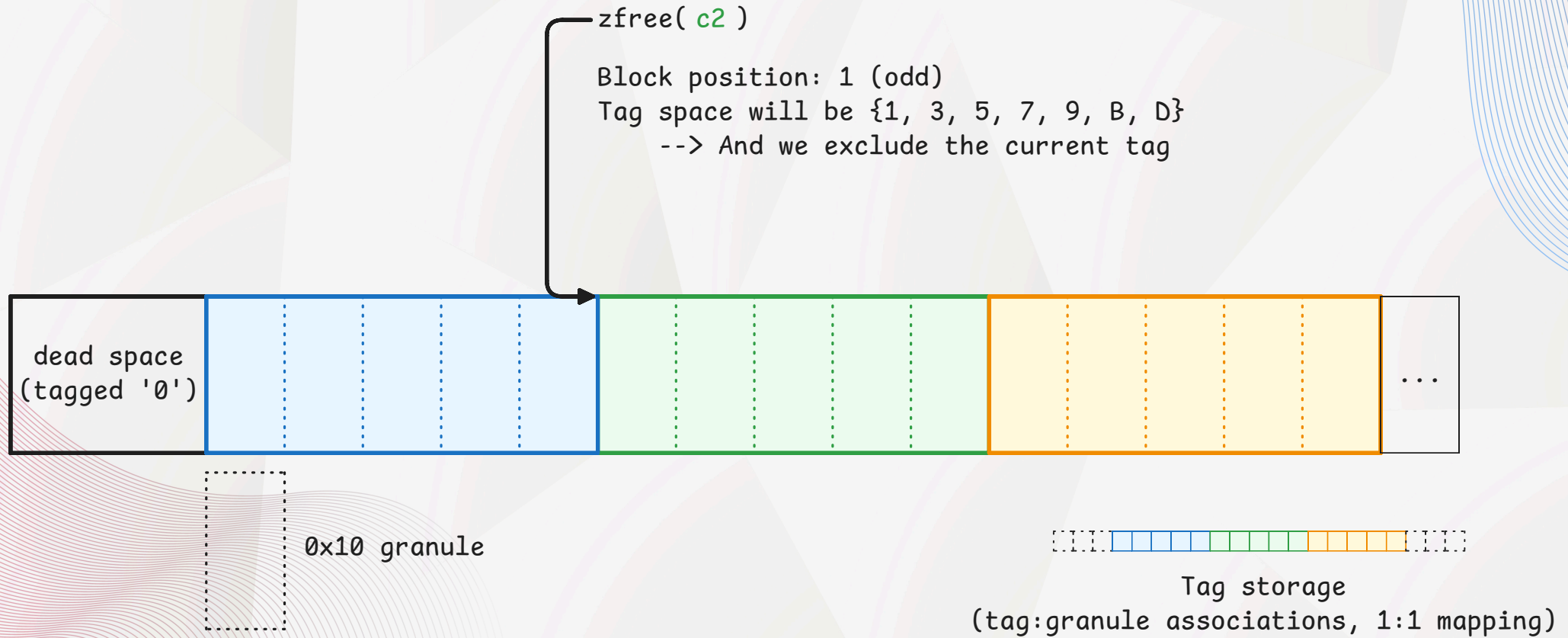
Kernel Allocators

Zalloc Layer — Free (since 26.4)



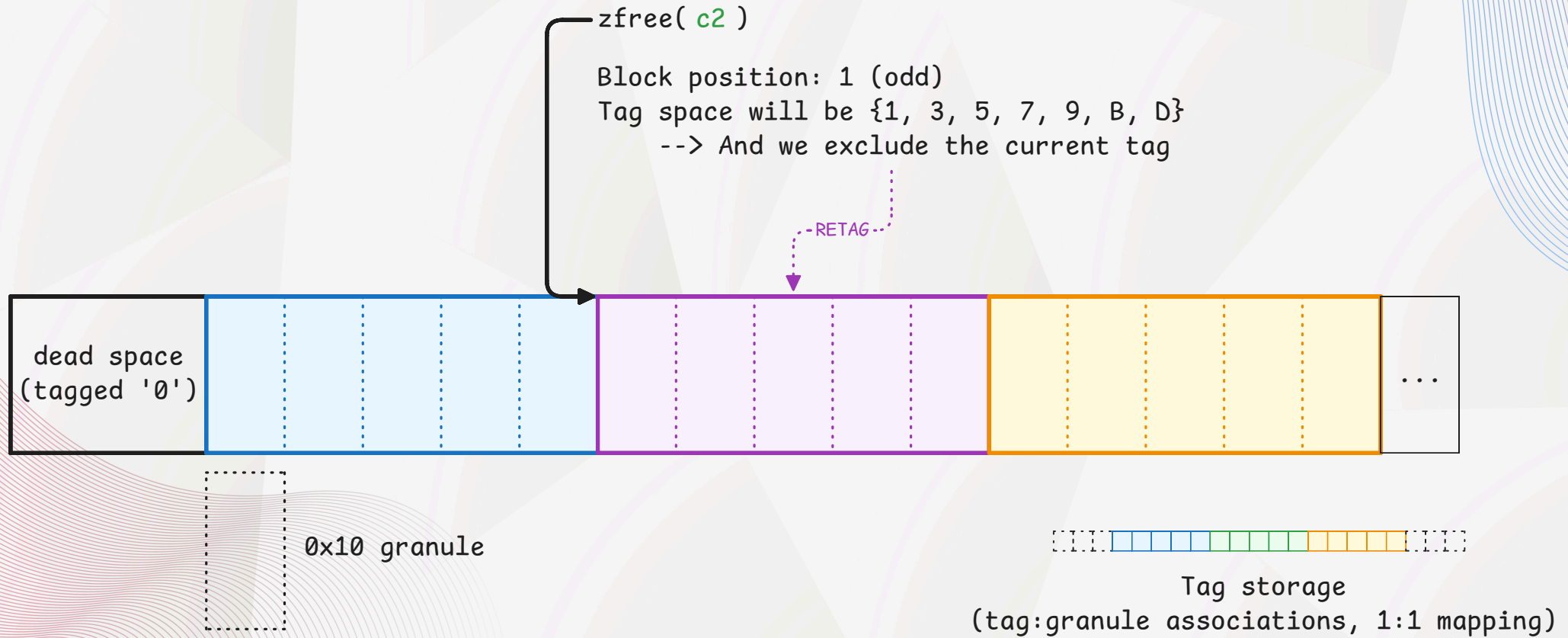
Kernel Allocators

Zalloc Layer — Free (since 26.4)



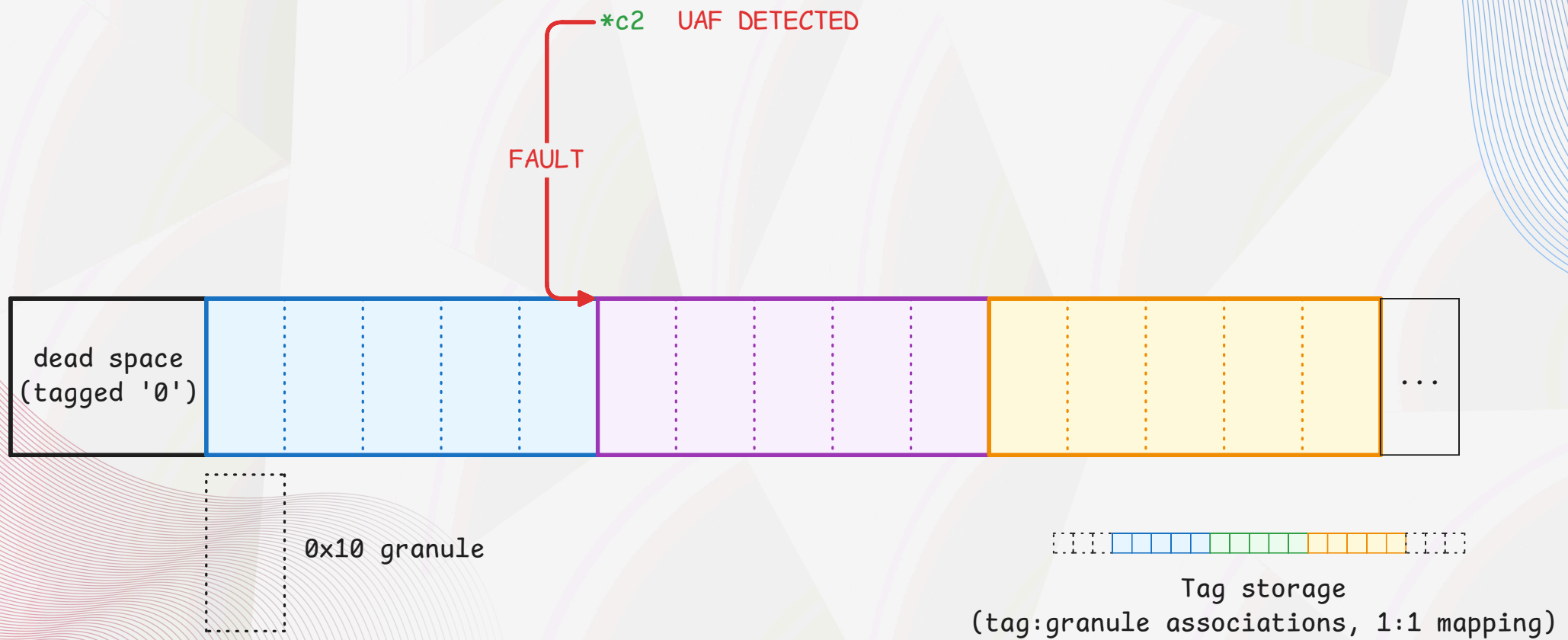
Kernel Allocators

Zalloc Layer — Free (since 26.4)



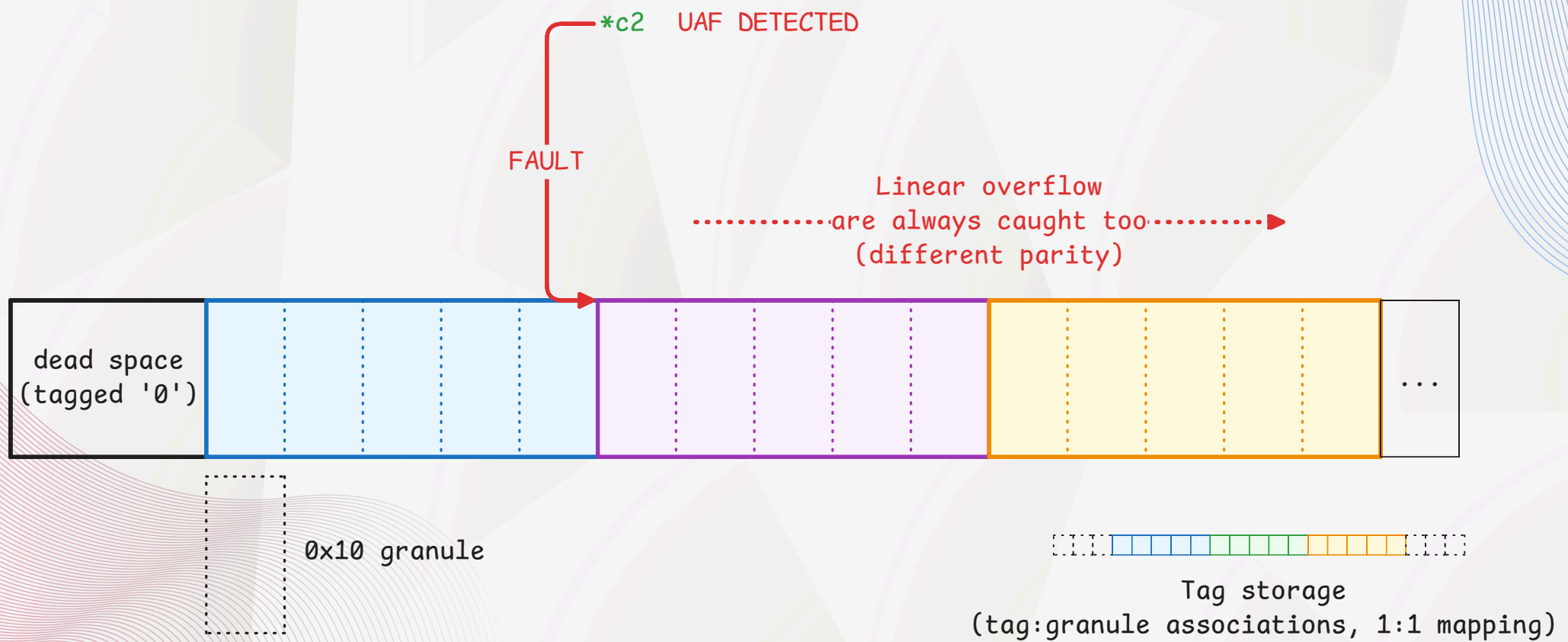
Kernel Allocators

Zalloc Layer — Free (since 26.4)



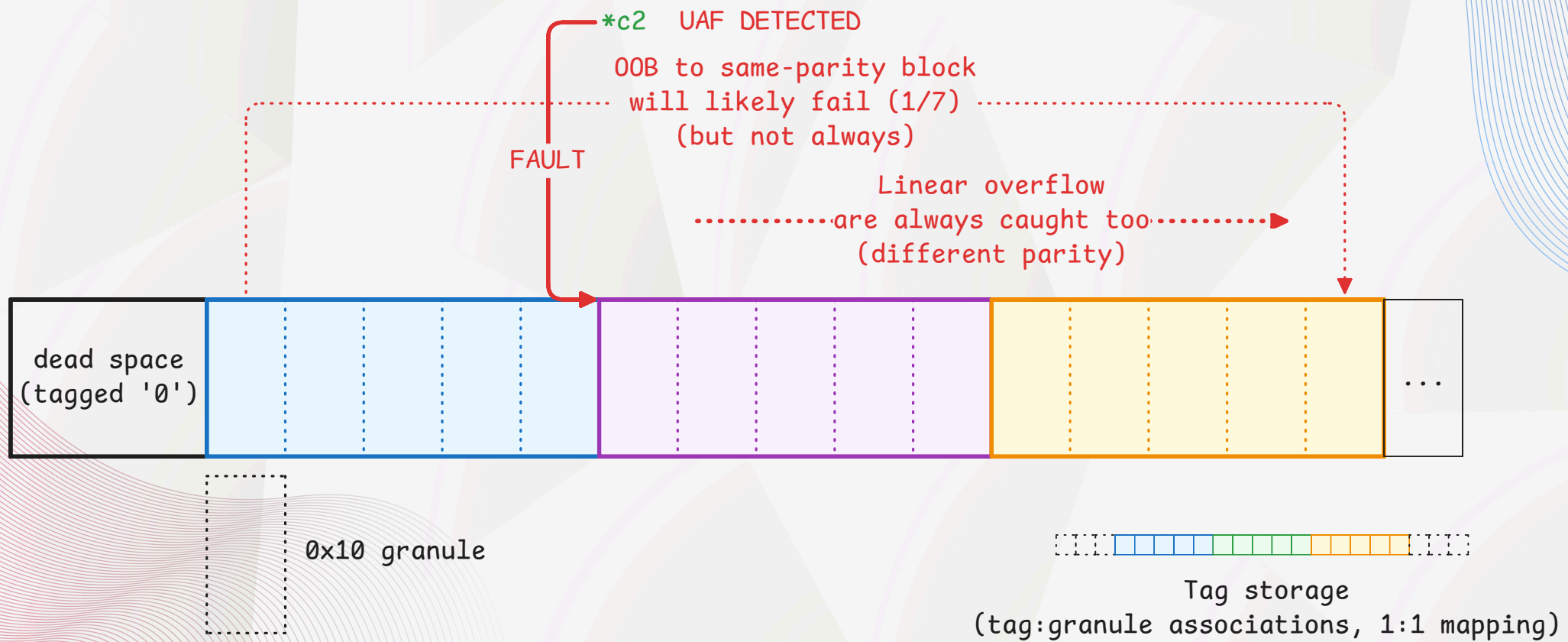
Kernel Allocators

Zalloc Layer — Free (since 26.4)



Kernel Allocators

Zalloc Layer — Free (since 26.4)



Kernel Allocators

What Can We Still Do?

- **MTE** is forcing attackers to either bypass it or change strategy:

Kernel Allocators

What Can We Still Do?

- **MTE** is forcing attackers to either bypass it or change strategy:
  - Attack **untagged allocations** (DATA-only, huge allocations, ...) → *26.4 closed the DATA gap*

Kernel Allocators

What Can We Still Do?

- **MTE** is forcing attackers to either bypass it or change strategy:
  - Attack **untagged allocations** (DATA-only, huge allocations, ...) → *26.4 closed the DATA gap*
  - Fall back to intra-object primitives (then possibly leak tags using it)

Kernel Allocators

What Can We Still Do?

- **MTE** is forcing attackers to either bypass it or change strategy:
  - Attack **untagged allocations** (DATA-only, huge allocations, ...) → *26.4 closed the DATA gap*
  - Fall back to intra-object primitives (then possibly leak tags using it)
  - Play at a lower level (i.e., physical operations like Darksword's LPE)

Kernel Allocators

What Can We Still Do?

- **MTE** is forcing attackers to either bypass it or change strategy:
  - Attack **untagged allocations** (DATA-only, huge allocations, ...) → *26.4 closed the DATA gap*
  - Fall back to intra-object primitives (then possibly leak tags using it)
  - Play at a lower level (i.e., physical operations like Darksword's LPE)
  - Hunt for other bug class

Navigating the MTE Landscape

Userland Allocators

Userland Allocators

About libmalloc & libpas

- Apple's **libpas** is used by WebKit and is now supporting **MTE** (but temporarily forced to *soft-mode*)
  - This scope itself could make another 30min talk

Userland Allocators

About libmalloc & libpas

- Apple's **libpas** is used by WebKit and is now supporting **MTE** (but temporarily forced to *soft-mode*)
  - This scope itself could make another 30min talk
- Apple's **libmalloc** is the system default allocator (`malloc()` , `free()` , `calloc()` , ...)
  - Delegates to **zones** implementing the `malloc_zone_t` interface:

Userland Allocators

About libmalloc & libpas

- Apple's **libpas** is used by WebKit and is now supporting **MTE** (but temporarily forced to *soft-mode*)
  - This scope itself could make another 30min talk
- Apple's **libmalloc** is the system default allocator (`malloc()` , `free()` , `calloc()` , ...)
  - Delegates to **zones** implementing the `malloc_zone_t` interface:

Allocator	Size Range	Design	MTE	Shipped
ScalableZone	General-purpose	Generic, scalable and attacker-friendly	No	X Yosemite 10.10
XZone malloc	General-purpose	*XNU-Style zones, modern and secure*	*Yes*	*Tahoe 26.1*
NanoZoneV2	`[0x10:0x100]`	Highly-effective small SLAB	No	Mojave 10.14

Userland Allocators

Zone Selection

- XZone **replaces** ScalableZone based on:
  - Apple boot-arg string `has_sec_transition=1` (*MTE == XZone*)
  - The `hardened-process.hardened-heap` entitlement
  - A name that matches the `security_critical` hardcoded list of process names
  - The feature flag `SecureAllocator_SystemWide` (*true on iOS*)

Userland Allocators

Zone Selection

- XZone **replaces** ScalableZone based on:
  - Apple boot-arg string `has_sec_transition=1` (*MTE == XZone*)
  - The `hardened-process.hardened-heap` entitlement
  - A name that matches the `security_critical` hardcoded list of process names
  - The feature flag `SecureAllocator_SystemWide` (*true on iOS*)
- When XZone is enabled, **NanoZoneV2** is **disabled** by default
  - But it can be activated through the feature flag `SecureAllocator_NanoOnXzone` (*false on iOS*)

Userland Allocators

XZone malloc — Design (1/2)

- **XZone malloc** is a SLAB allocator based on mimalloc and **zalloc** design
- Supporting **typed allocations** and **MTE**
- Respecting **LIFO** ordering
- Overview documentation in libmalloc/doc/xzone_malloc.md

Userland Allocators

XZone malloc — Design (1/2)

- **XZone malloc** is a SLAB allocator based on mimalloc and **zalloc** design
- Supporting **typed allocations** and **MTE**
- Respecting **LIFO** ordering
- Overview documentation in libmalloc/doc/xzone_malloc.md

Size Family	Range	Chunk Size	Allocation Strategy	Retagging Strategy
TINY	16B - 4KB	16KB	Lock-free CAS freelist	*retag-on-free*
SMALL	4KB - 32KB	64/128KB	Bitmap + lock, or lock-free freelist	*retag-on-realloc*
LARGE	32KB - 2MB	2-128 slices	Direct segment allocation / gzone	*no tag* (yet)
HUGE	>2MB	256+ slices	Direct VM allocation	*no tag*

Userland Allocators

XZone Malloc — Design (2/2)

Defense	Targets
Separated metadata	except some **PAC-protected** inplace freelist
Separated Heaps (DATA / PTR)	Cross-type exploitation
VA sequestering	Cross-zone attacks (a chunk is pinned to zone forever)
Type segregation	Type confusion via heap spraying
Guard chunks (random)	Linear overflows between chunks (enabled for `security_critical` processes)
Randomized front index	Allocation predictability
Per-CPU policy	Aims at performance first, but also impacts allocation predictability

*First N allocations are served by **MFM** before XZone malloc kicks in (budget-limited early allocator)*

Userland Allocators

XZone Malloc — Type Segregation (1/2)

- Each allocation is routed to an **xzone** (a slab) based on its **size**/**type descriptor**:
  - `_xzm_xzone_lookup(size, type_desc)`
  - `size` → **bin** (size class), `type_desc` → **bucket** (type identity)
  - xzone index = `bin_offset + bucket`

Userland Allocators

XZone Malloc — Type Segregation (1/2)

- Each allocation is routed to an **xzone** (a slab) based on its **size**/**type descriptor**:
  - `_xzm_xzone_lookup(size, type_desc)`
  - `size` → **bin** (size class), `type_desc` → **bucket** (type identity)
  - xzone index = `bin_offset + bucket`
- **Type descriptors** are inferred at **compilation** time (or derived from the call-site)
  - Based on the object's structure layout (size, fields kinds, fields positioning, ...)
  - Beware: `uintptr_t` is for example considered **DATA** (typed as i64 by LLVM)

Userland Allocators

XZone Malloc — Type Segregation (1/2)

- Each allocation is routed to an **xzone** (a slab) based on its **size**/**type descriptor**:
  - `_xzm_xzone_lookup(size, type_desc)`
  - `size` → **bin** (size class), `type_desc` → **bucket** (type identity)
  - xzone index = `bin_offset + bucket`
- **Type descriptors** are inferred at **compilation** time (or derived from the call-site)
  - Based on the object's structure layout (size, fields kinds, fields positioning, ...)
  - Beware: `uintptr_t` is for example considered **DATA** (typed as i64 by LLVM)
- **Bucketing keys** are:
  - Derived from `executable_boothash` (boot UUID + cdhash)
  - Deterministic per-boot, per-executable

Userland Allocators

XZone Malloc — Type Segregation (2/2)

Bucket	Description
DATA	Pure data, no pointers
OBJC	Objective-C object instances (MALLOC_TYPE_KIND_OBJC)
POINTER 0..3 (4 on macOS)	General pointer-bearing (keyed hash of `type_desc`)

> *Not as many as kalloc_type buckets, but still enough to make attackers' lives harder*

Userland Allocators

XZone malloc — MTE Tagging Policy

- **Tagging decision** per xzone: IF (TINY OR SMALL) AND (!data OR tag_data)
- **VM-level**: segments allocated with **VM_FLAGS_MTE** via mach_vm_map to enable hardware tag storage
- Tagging for **LARGE** is supported but not enforced yet (**TODO** in the codebase)

Bucket	Tagged by Default	Rationale
PTR	**Yes**	Pointer-bearing allocations, hijackable
OBJC	**Yes**	Pointer-bearing allocations, hijackable
DATA	**Yes** (*since 26.4*)	Data is also exploitable (`uintptr_t` alike, states, ...)

Userland Allocators

What Can We Still Do?

- **XZone** is bringing some serious security to userland too, even without **MTE**

Userland Allocators

What Can We Still Do?

- **XZone** is bringing some serious security to userland too, even without **MTE**
- **MTE** is again filling the last gaps:
  - No more **overflow** or **OOB** past the block boundary
  - No more direct **UAF** → **Object Confusion**

Userland Allocators

What Can We Still Do?

- **XZone** is bringing some serious security to userland too, even without **MTE**
- **MTE** is again filling the last gaps:
  - No more **overflow** or **OOB** past the block boundary
  - No more direct **UAF → Object Confusion**
- This is again forcing attackers to either bypass **MTE** or change strategy
  - Same shift as kernel allocations
  - Some platform binaries are still **not using MTE**, but it is becoming more and more adopted
  - Remember that 3rd party binaries are no longer forced to **soft-mode**

Navigating the MTE Landscape

Closing Thoughts

Navigating the MTE Landscape

Closing Thoughts

- Apple's **MIE** is a pretty great protection against memory corruptions, but it is not perfect
  - Powerful primitives like Darksword's LPE are still a good entrypoint (but will need extra-work)
  - Intra-object corruptions too (not caught by design) and it can help leak tags
  - UAFs and OOBs are not the only way to attack a heap anyway

Navigating the MTE Landscape

Closing Thoughts

- Apple's **MIE** is a pretty great protection against memory corruptions, but it is not perfect
  - Powerful primitives like Darksword's LPE are still a good entrypoint (but will need extra-work)
  - Intra-object corruptions too (not caught by design), and it can help leak tags
  - UAFs and OOBs are not the only way to attack a heap anyway
- **MIE** can also benefit attackers using fuzzing to find memory corruptions
  - You can then reproduce on **non-MTE devices** (Obviously not a long-term strategy)
  - Most of the bugs won't be exploitable on **MTE devices**, but some might be meaningful
  - e.g., a caught overflow/OOB might still be a candidate for an intra-object corruption

Navigating the MTE Landscape

Questions?

References

- **Apple Security Blog** - Memory Integrity Enforcement: A complete vision for memory safety in Apple devices
- **Apple Security Blog** - Towards the next generation of XNU memory safety: kalloc_type
- **ARM Architecture** - MTE / EMTE specification
- **Google Project Zero** - MTE As Implemented (Parts 1-3, Mark Brand, 2023)
- **Hexacon 2025** - Keynote by Ivan Krstić
- **XNU source** - apple-oss-distributions/xnu
- **libmalloc source** - apple-oss-distributions/libmalloc
- **Universität Potsdam** - Modern iOS Security Features: A Deep Dive into SPTM, TXM, and Exclaves
- **SPTM firmware reverse engineering**
  - `sptm.t8150.release` (iPhone 17) and `sptm.t8142.release` (MacBook M5)