No Tux Given

Diving Into Contemporary Linux Kernel Exploitation

Sam Page, #TyphoonCon23

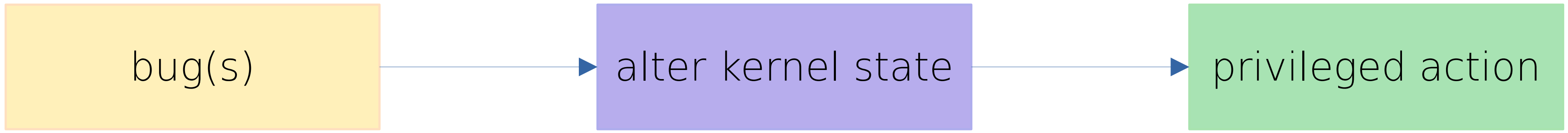
About Me

- Sam (@sam4k1)
- Background in VR and exploit dev
- I like Linux, security, games & food

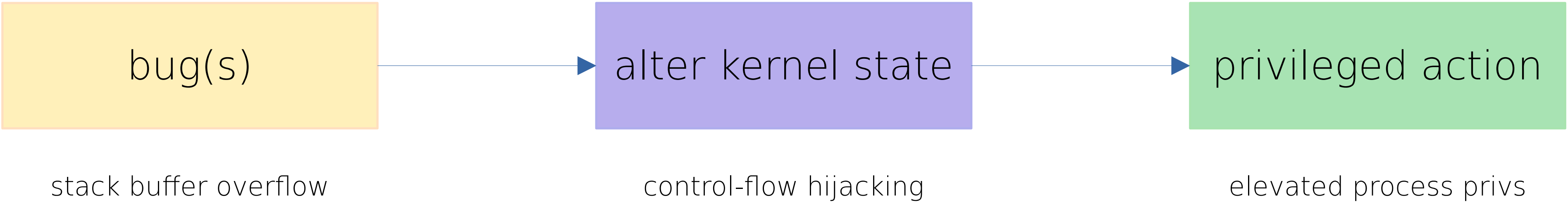
What Are We Doing Here?

- Exploring the past, present & future of kernel security & xdev
- Hopefully making an increasingly complex topic more accessible
- Do we need any more reasons??? This stuff is awesome!

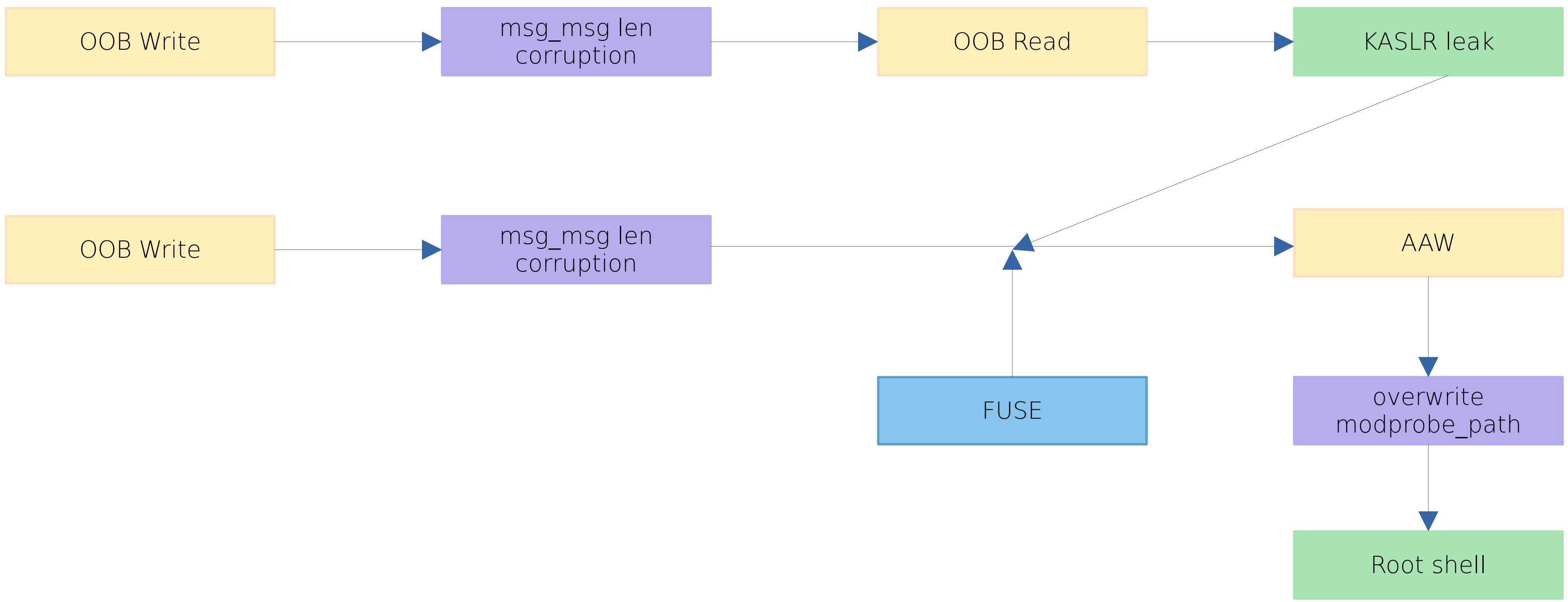
Tl;dr kernel exploits??



Tl;dr kernel exploits??



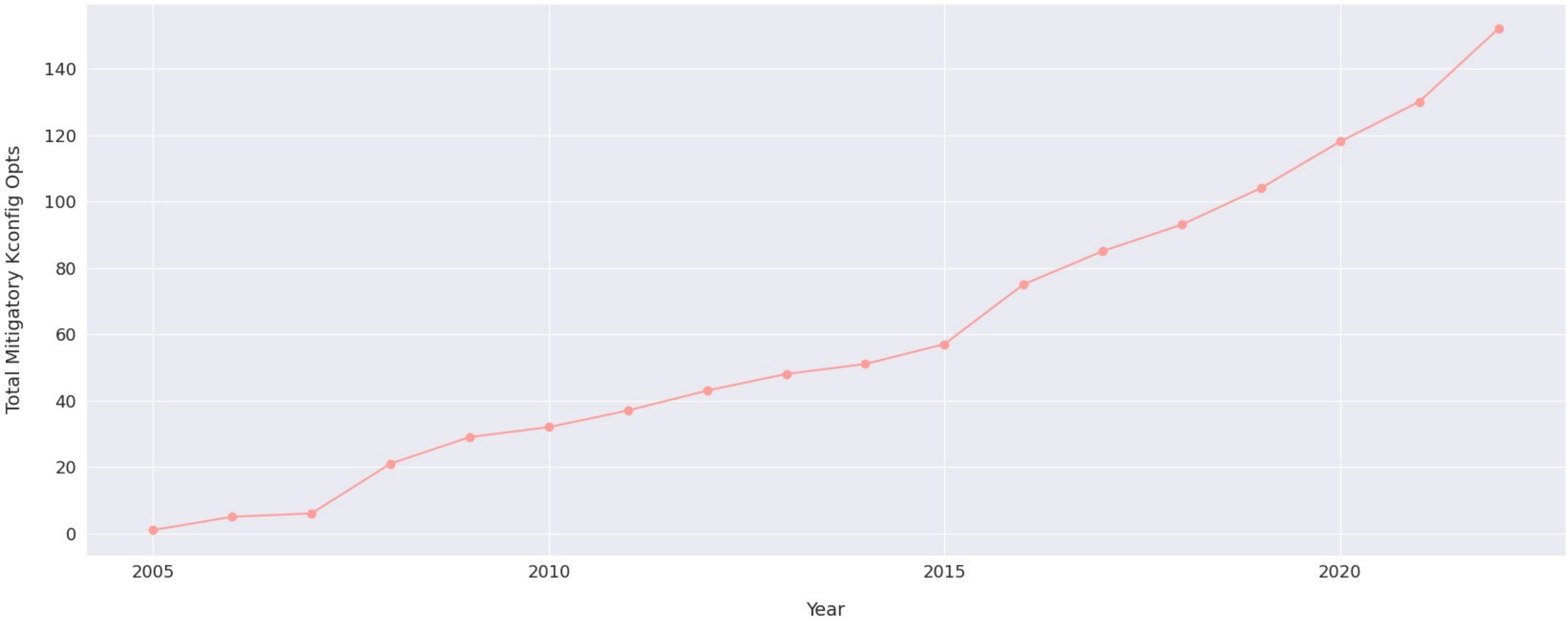
Tl;dr kernel exploits??



Tux’s Security Past

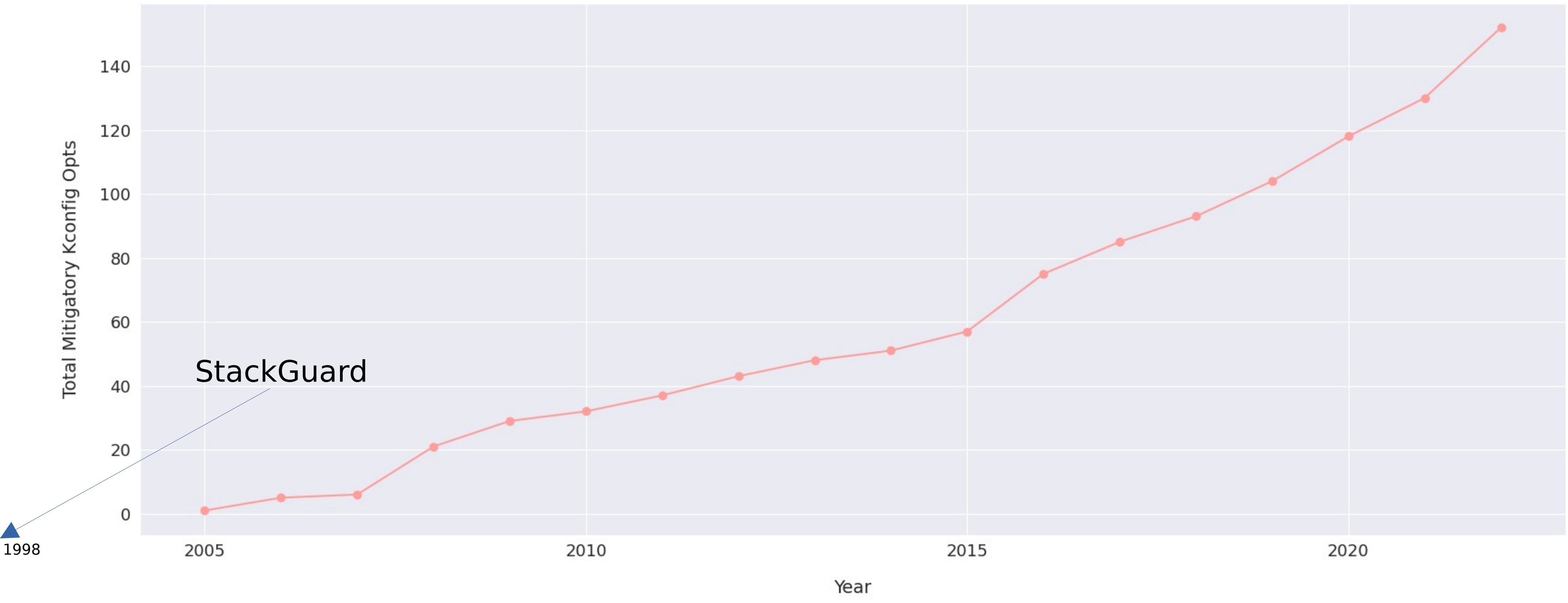
Examining Historical Kernel Exploitation Trends

Mitigations



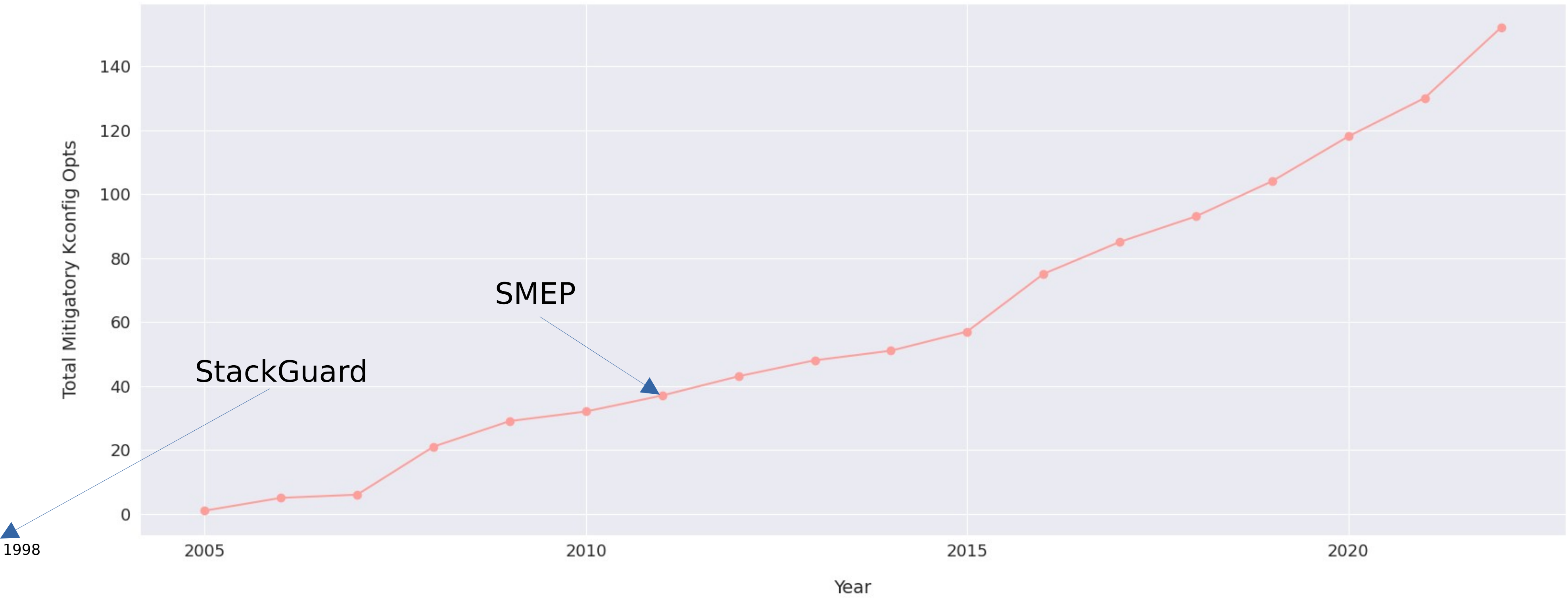
Data: kconfig-hardened-check[1], LKDDb[2]

Mitigations



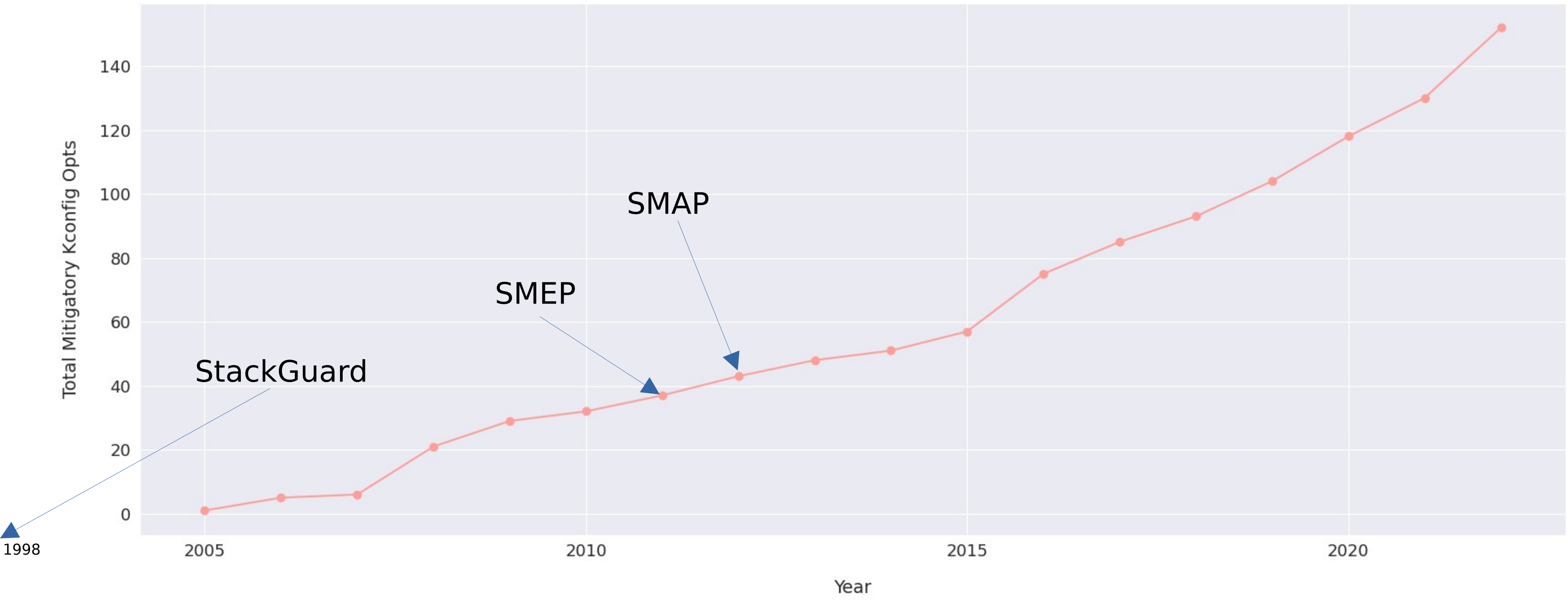
Data: kconfig-hardened-check[1], LKDDb[2]

Mitigations



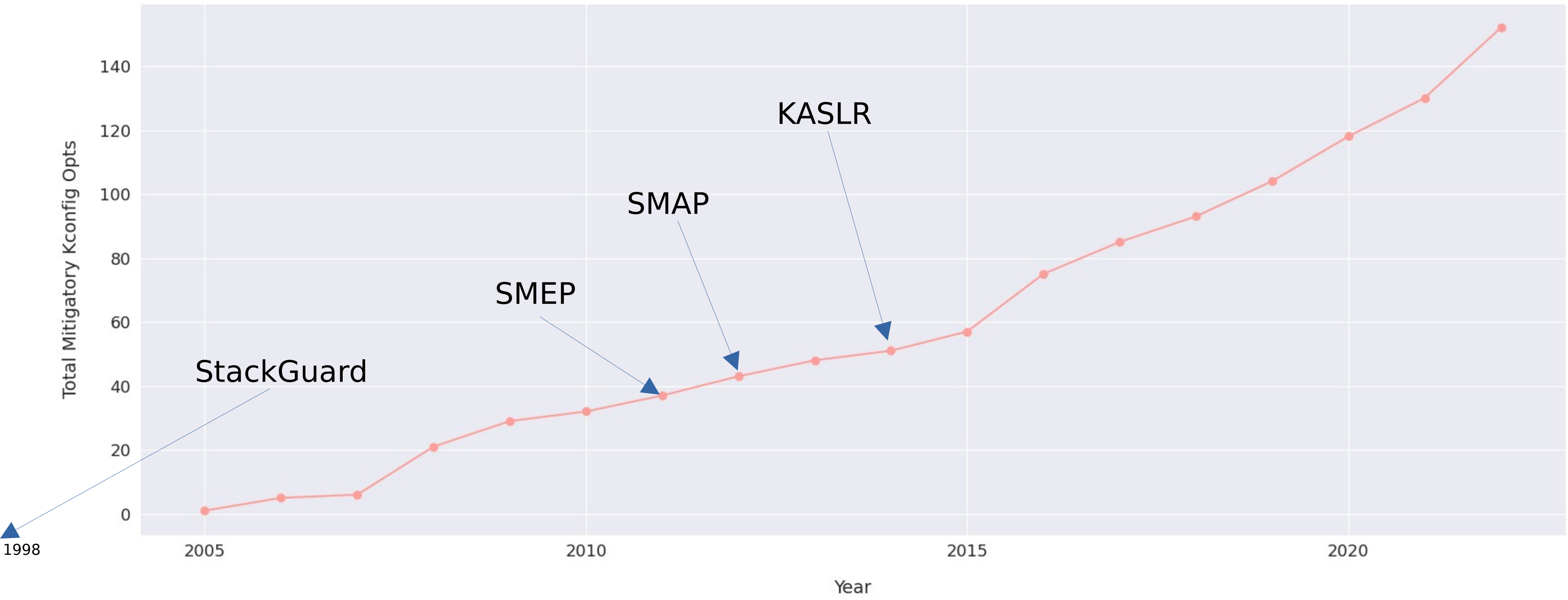
Data: kconfig-hardened-check[1], LKDDb[2]

Mitigations



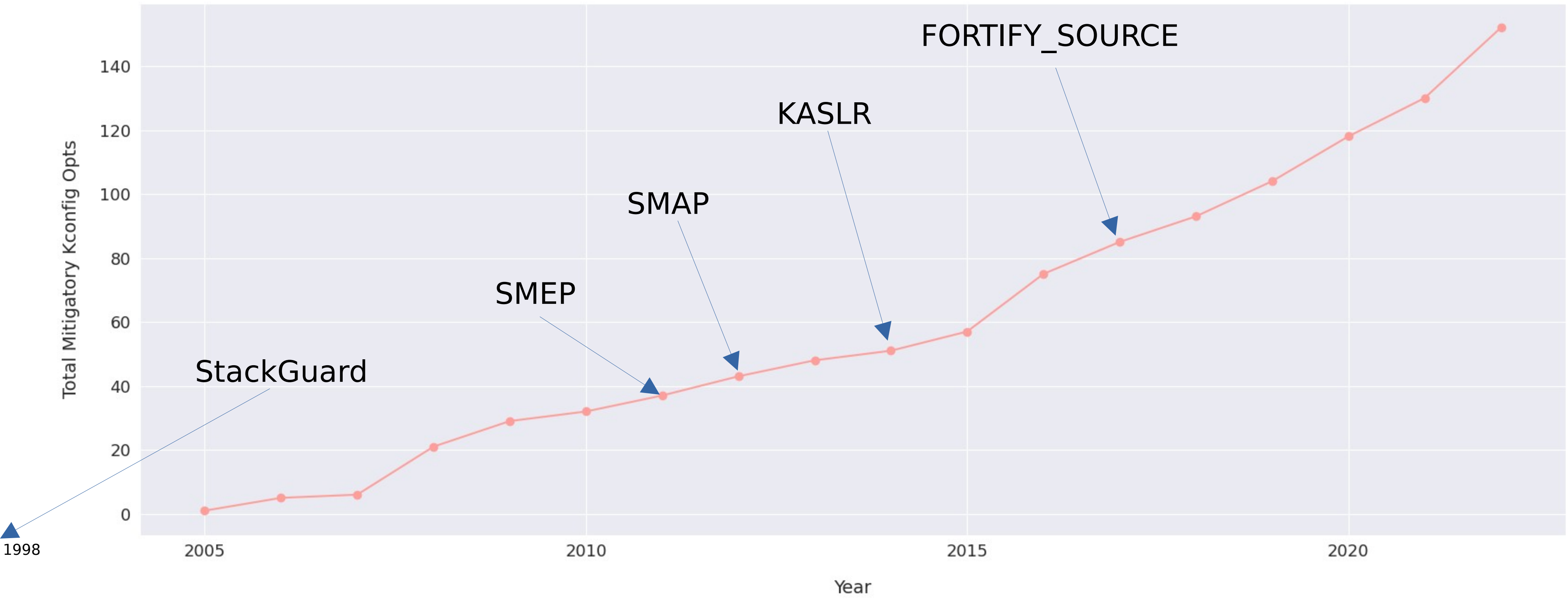
Data: kconfig-hardened-check[1], LKDDb[2]

Mitigations



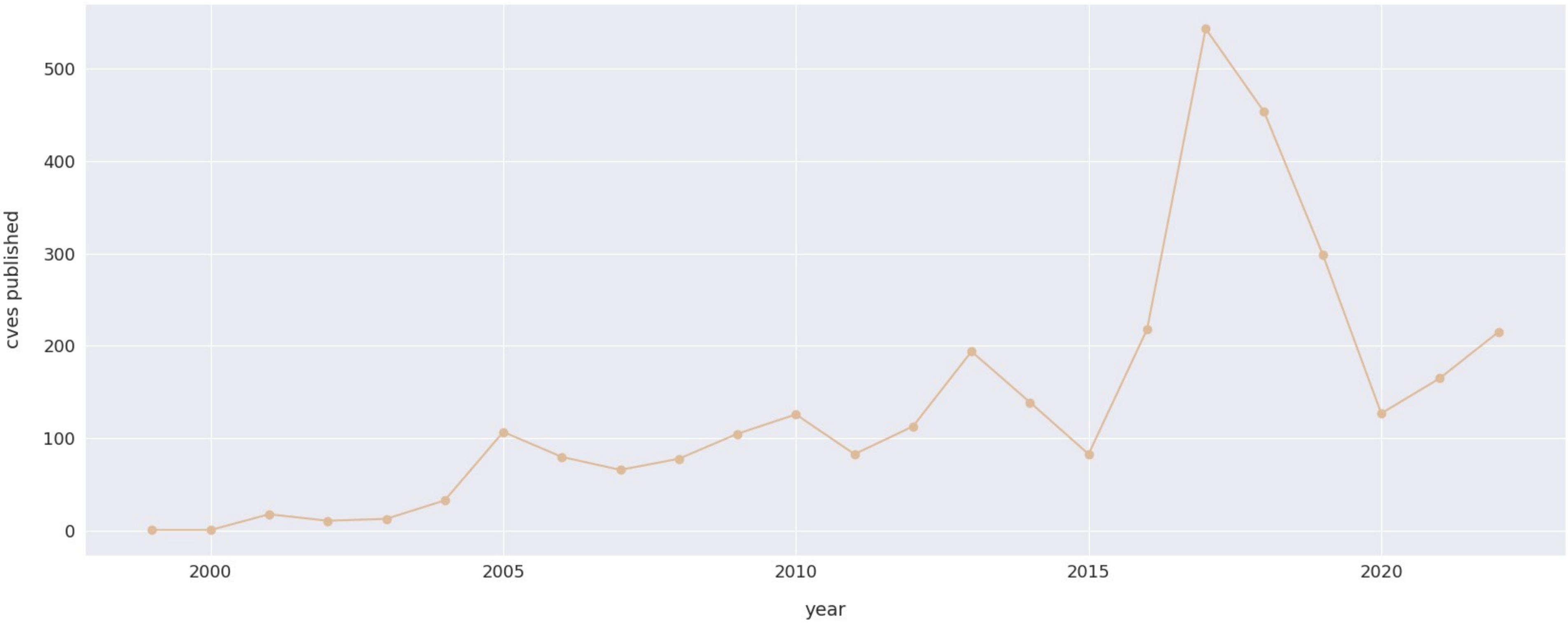
Data: kconfig-hardened-check[1], LKDDb[2]

Mitigations



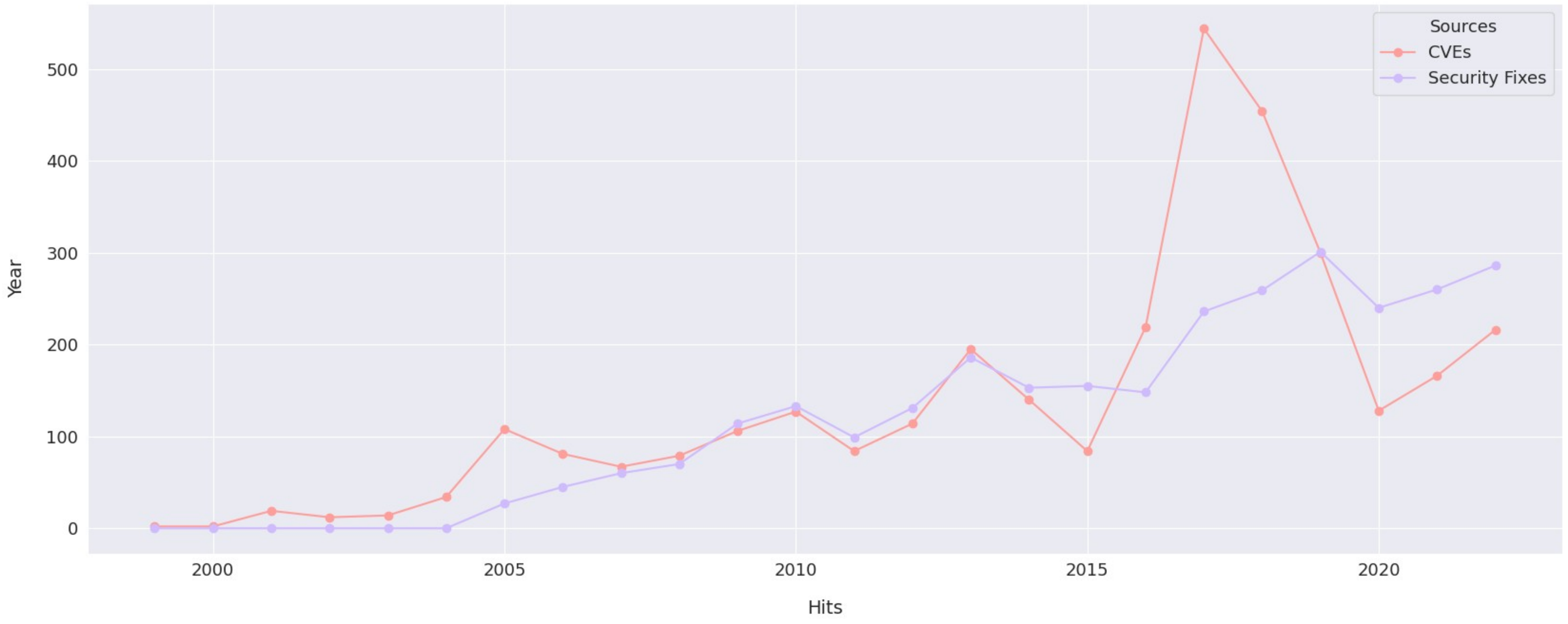
Data: kconfig-hardened-check[1], LKDDb[2]

Bug Trends



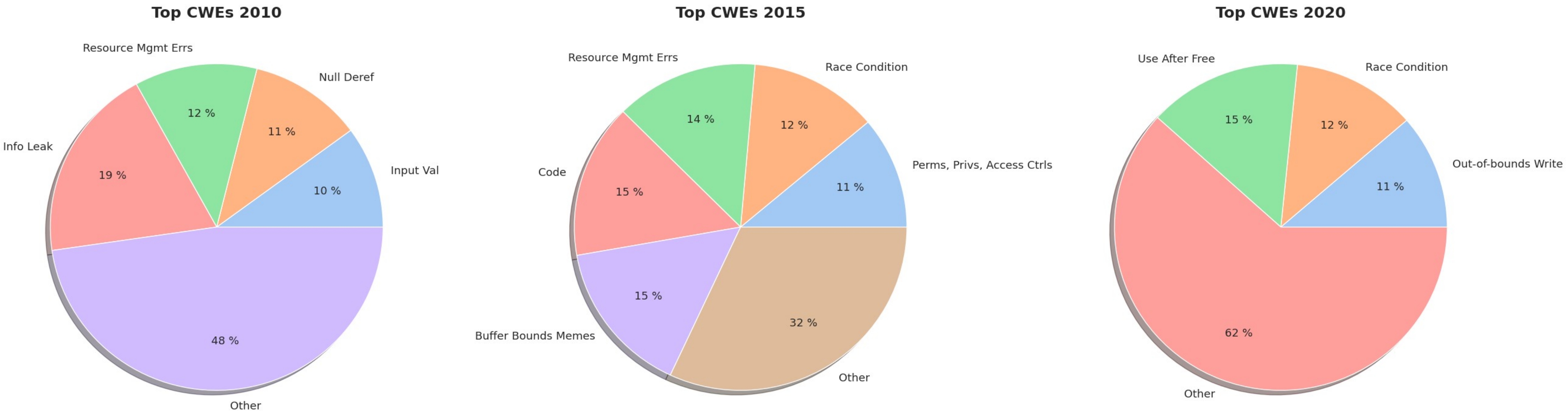
Data: gsd-database[3]

Bug Trends



Data: gsd-database[3], torvalds/linux[4]

Bug Trends



Data: gsd-database[3]

Tux’s Security Present

Looking At Contemporary Kernel Exploitation

Kernel Exploits in 2023 | The Process

The process of getting from bug to privesc has become more complex:

1) Need to understand the attack surface

2) Find yourself some bugs (ezpz right?)

3) Figure out how, and what you need, to exploit it

   - Typically takes knowledge of platform/surface/bug and existing techniques

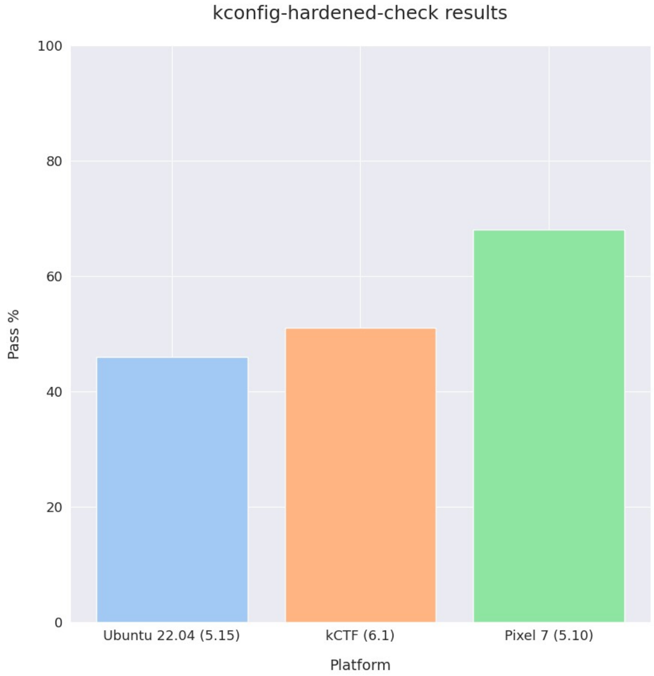
4) Actually get a (reliably??) working proof-of-concept

Kernel Exploits in 2023 | The Mindset

- Curiosity! Ask questions and take the time to understand
- Patience helps too, as sometimes there are no solutions
- Document, document, document! You'll thank yourself
- Opt for generic tooling and techniques where possible, to reuse
- The kernel is unforgiving of mistakes and unexpected behaviour!

Understanding The Attack Surface

- Informs where to look for bugs, what to look for and how to exploit them
- Lots of factors to consider: Kconfig, arch, platform specifics, 3rd parties etc.
- Varies greatly across desktop, android, IoT



Finding Some Bugs | Approaches

- Doesn't have to be 0days! Syzbot dashboard, silent fixes, n-days etc.
- QEMU + gdb make it easy to dig deeper and do some dynamic analysis
- Time spent understanding the bug & surface will help going forward
- Factor in surface/mitigations when thinking about what to look for

Finding Some Bugs | Tools & Tips

But if you do want a shiny 0day there's…

- Good ol' fashioned code auditing
- CodeQL to help flag areas of interest or check for specific patterns
- Spin up your own modified syzkaller instance
  - Adding coverage for areas without descriptions (e.g. 3rd party drivers)
  - Extending coverage for more tailored fuzzing using platform knowledge

From Bug To

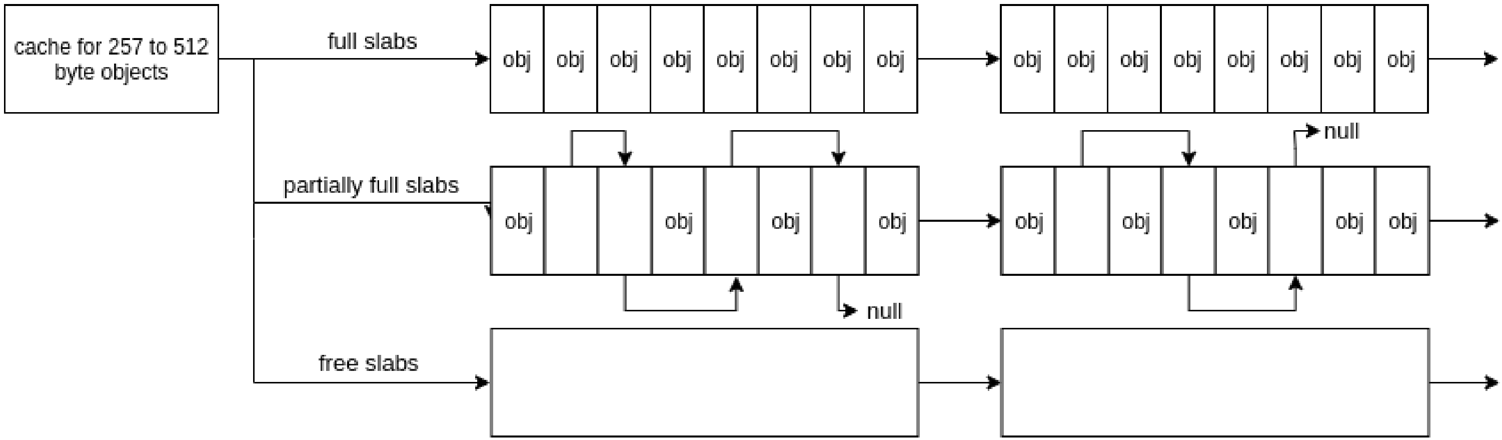
- Bug provides our initial primitive
- Generic techniques & strategies to leveraging particular primitives
- With each surface/bug often having its own nuances & requirements
- Goal is to chain these together to ultimately privesc
  - Typically via elevating our procs privs or executing another bin with privs

Exploiting UAFs | Getting Our Bearings

- Can cause the kernel to do some action(s) on previously freed memory
- So we need to think about how the kernel allocates this memory:

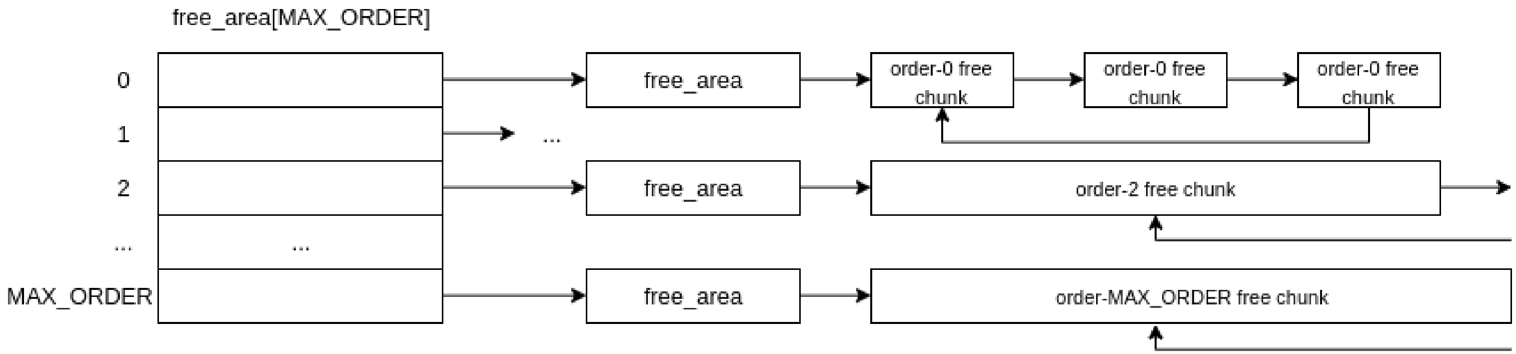
Exploiting UAFs | Getting Our Bearings

- Can cause the kernel to do some action(s) on previously freed memory
- So we need to think about how the kernel allocates this memory:
  - SLUB allocator: used for small, commonly used objects



Exploiting UAFs | Getting Our Bearings

- Can cause the kernel to do some action(s) on previously freed memory
- So we need to think about how the kernel allocates this memory:
  - Page allocator: handles larger, contiguous allocs (including slabs!)



(Where chunk size = 2^{order} * `PAGE_SIZE`)

Exploiting UAFs | Getting Our Bearings

- Can cause the kernel to do some action(s) on previously freed memory
- So we need to think about how the kernel allocates this memory:
  - SLUB allocator: used for small, commonly used objects
  - Page allocator: handles larger, contiguous allocs (including slabs!)
- We also need to consider what actions are done on the freed memory
- As well as how reachable/triggerable the UAF is and any timing issues

Exploiting UAFs | Mitigations

	Ubuntu 22.04 (5.15)	kCTF (6.1)	Pixel 7 (5.10)
`init_on_alloc`	default	default	default
`SLAB_FREELIST_RANDOM`	default	default	default
`SHUFFLE_PAGE_ALLOCATOR`	default	not set	default
`STATIC_USERMODEHELPER`	not set	not set	default
no unpriv `userfaultfd` OR FUSE	FUSE	neither enabled	neither unpriv*
`slab_nomerge`	not set	default	default

Exploiting UAFs | Realising Our Goal

- Need an object to replace our freed one
  - Such that actions done on it give us further kernel primitives/priv esc
  - CodeQL is a useful tool here to query specific obj criteria (size, offsets etc.)
- Then we need to make sure our object(s) ends up where its supposed to…
  - i.e. we need to understand how to control the memory layout

Exploiting UAFs | Shaping General Purpose Caches

- Different **`gfp_t flags`** may be allocated into different general purpose caches
  - E.g. **`GFP_KERNEL_ACCOUNT`**, used for objects containing user data
- Elastic objects provide us with a generic approach, usable across cache sizes
- Cache noise is also an important factor in tuning the reliability of your heap spray
- FUSE can open up more allocation possibilities by allowing us to keep more ephemeral object allocations in memory[8]

```
 OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
71744  48601  67%    0.06K   1121       64      4484K kmalloc-64
50624  50624 100%    0.06K    791       64      3164K kmalloc-rcl-64 # __GFP_RECLAIM
46179  45949  99%    0.19K   2199       21      8796K kmalloc-192
46080  44526  96%    0.03K    360      128      1440K kmalloc-32
36352  36010  99%    0.25K   1136       32      9088K kmalloc-256
# ...
10240  10240 100%    0.01K     20      512        80K kmalloc-cg-8   # __GFP_ACCOUNT
# ...
  256      0   0%    0.02K      1      256         4K dma-kmalloc-16 # __GFP_DMA
# ...
```

$ sudo slabtop

```
void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *objp);
```

API example for general purpose allocs

Exploiting UAFs | Shaping Private Caches

```c
struct kmem_cache *kmem_cache_create(const char *name, unsigned int size,
            unsigned int align, slab_flags_t flags,
            void (*ctor)(void *));
void *kmem_cache_alloc(struct kmem_cache *s, gfp_t flags);
void kmem_cache_free(struct kmem_cache *s, void *objp);
```

API example for private cache allocs

```
  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
754992 754536  99%    0.19K  35952       21    143808K dentry
493056 490561  99%    0.03K   3852      128     15408K numa_policy
343280 342993  99%    0.57K  12260       28    196160K radix_tree_node
272944 272913  99%    1.12K   9748       28    311936K btrfs_inode
269640 267748  99%    0.07K   4815       56     19260K vmap_area
265216 134497  50%    0.01K    518      512      2072K lsm_file_cache
206808 198880  96%    0.14K   7386       28     29544K btrfs_extent_map
138560 117011  84%    0.06K   2165       64      8660K dmaengine-unmap-2
123100 122135  99%    0.16K   4924       25     19696K vm_area_struct
 74752  73535  98%    0.06K   1168       64      4672K anon_vma_chain
```

$ sudo slabtop

- Same goal as before, except…
- These caches only contain specified obj
- But… the slabs that make up private and general purpose caches are allocated the same way, by the buddy allocator
- With a bit more work, tuning and luck it's possible to have the slab containing freed private obj to be reallocated as a slab for a general purpose cache
- AKA cross-cache attacks

Exploiting UAFs | Shaping The Buddy (Page) Allocator

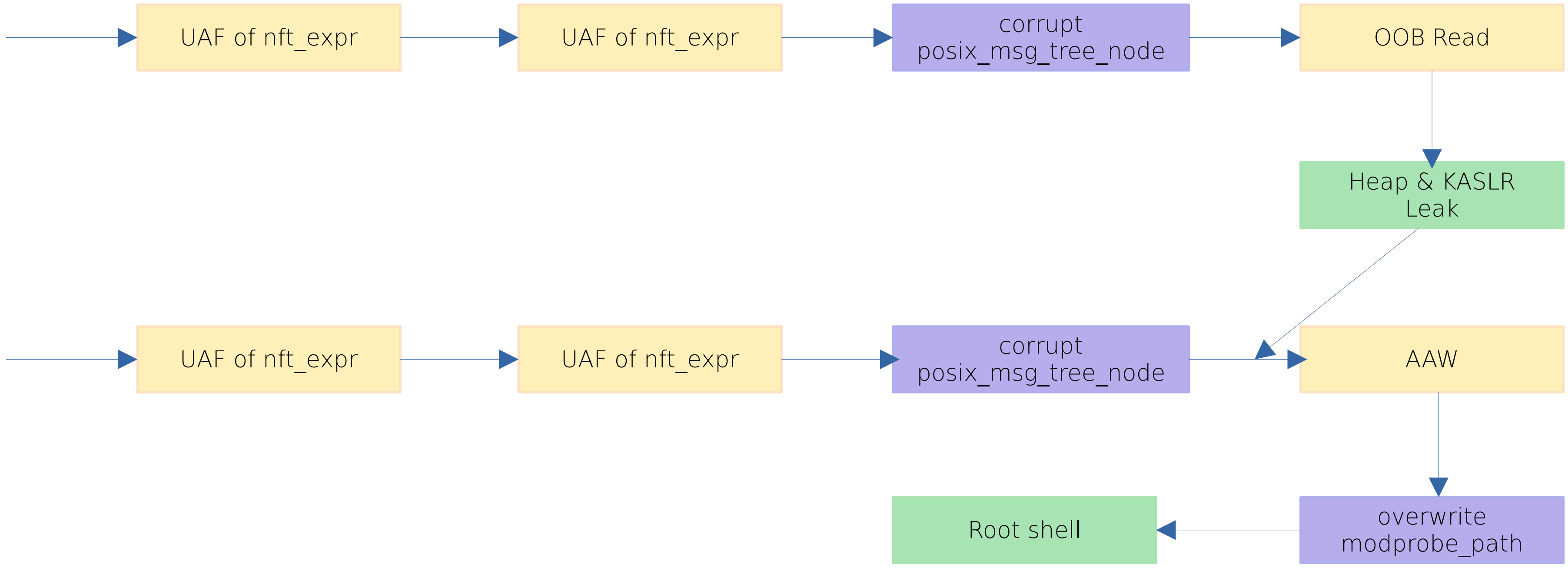
- Goal is the same, just need to remember the different structure!
- Used for large dynamic buffers (GPU, packet ring buffers), *slabs*
- Need to mitigate noise from chunks merging
  - If lower order is empty, chunks are split
  - If higher order is empty, *contiguous* chunks merged
- May also want to ensure contiguity of multiple allocations

```c
struct page *alloc_pages(gfp_t gfp, unsigned int order);
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);
void __free_pages(struct page *page, unsigned int order);
void free_pages(unsigned long addr, unsigned int order);
static void *__kmalloc_large_node(size_t size, gfp_t flags, int node);
```

```
----- zone info ------| 0   | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10
-------------------------------------------------------------------------------------------------
Node 0, zone      DMA      1      1      1      1      1      1      1      1      0      1      2
Node 0, zone    DMA32  37337  28593  15203   5885   1783    703    322    208    145    147    198
Node 0, zone   Normal   4553   1541   3541   1734    952    396    159     78     29     16    107
```

Exploiting UAFs | A Real World Example

- CVE-2022-32250[5] was a UAF in Netfilter:



Exploiting OOB Writes | What Bounds?

- Different kinds out-of-bounds writes in the kernel…
  - Array indexes, heap overflows, stack overflows etc.
- However this list may be shorter after we factor in mitigations…

Exploiting OOB Writes | Mitigations

	Ubuntu 22.04 (5.15)	kCTF (6.1)	Pixel 7 (5.10)
FORTIFY_SOURCE	default	default	default
UBSAN	UBSAN_TRAP not set	not set	default
SLAB_FREELIST_HARDENED	default	not set	default
STATIC_USERMODEHELPER	not set	not set	default

Exploiting OOB Writes | Heap Overflows It Is

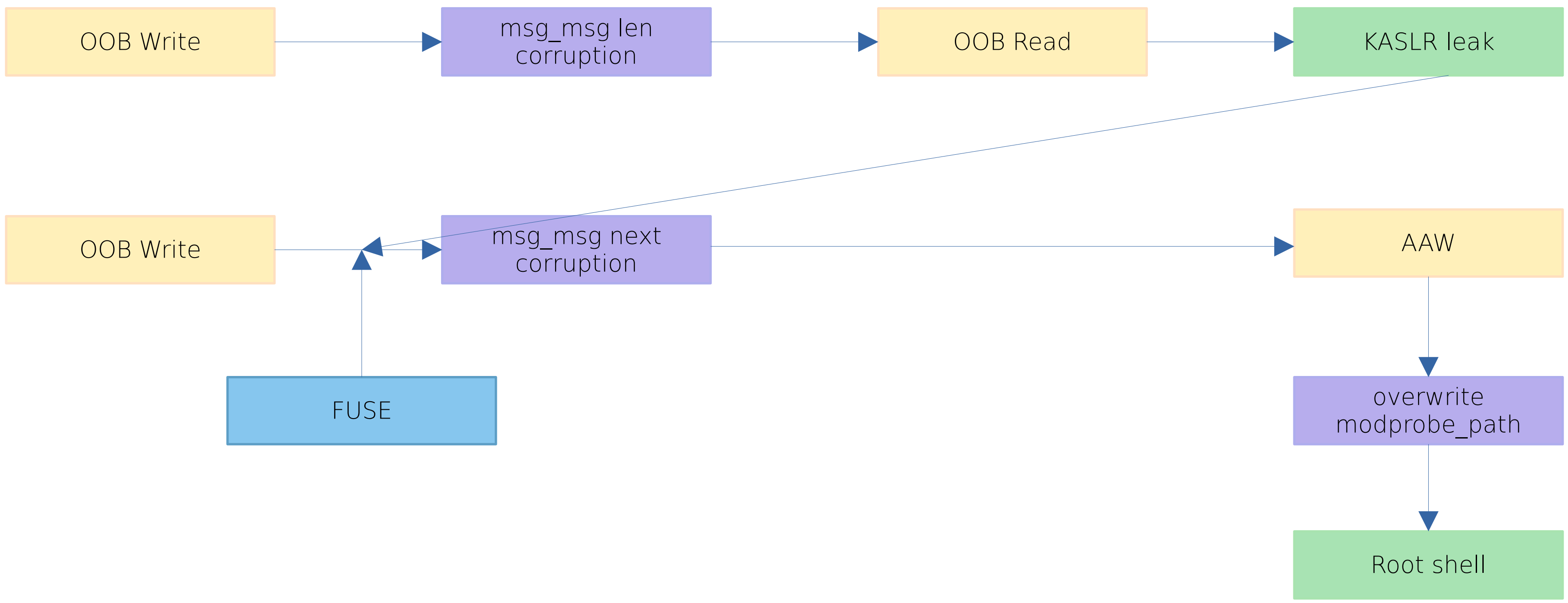
- As we’re dealing with the heap: how is our object allocated?
- Now interested in what’s adjacent to our object:
  - Another object we can corrupt?
  - The freelist pointer for the slab?
  - Another buddy allocated chunk?
- What is the extent of our overflow? Controlled size/data?
- With all this info, we can find a suitable candidate to corrupt

Exploiting OOB Writes | Getting To The Finish Line

- Want to pivot from our initial OOB write primitive
- Elastic objects are a popular target
  - E.g. `msg_msg` can be used to pivot from an OOB write to AAW[6]
- Cross-cache attacks open up possible targets to sensitive, otherwise inaccessible corruption targets
- `modprobe_path` is still an easy target to privesc with an AAW

Exploiting OOB Writes | A Real World Example

- CVE-2022-0185[7] was a heap overflow in fsconfig(2):



Exploiting Race Conditions

- Typically enable other bugs, such as use-after-frees
- But can be hard to debug; how do we know we're even winning the race?!
  - Printk debugging 😎 (or other kernel instrumentation, e.g. sleeps to widen the race)
  - Gdb scripts can also make life easier here
- And if we can win it, what if the odds are super low?
  - FUSE (or userfaultfd on older systems) may be an option for hanging kernel execution
  - Alternatively, user-triggerable interrupts (e.g. timers) can widen race condition too[11]
- Be considerate of the little gotchas
  - Execution contexts? Locks? Who's executing what, when? CPU affinity? etc.

Tux’s Security Future

Some Thoughts On Future Impacts to Kernel Security

Looking Ahead

Looking Ahead | New Mitigations

- kCTF experimental mitigations[10]:
  - `KMALLOC_SPLIT_VARSIZE`: mitigate generic direct object reuse via elastic objects (looking at you `msg_msg`!)
  - `SLAB_VIRTUAL`: mitigate cross-cache attacks by reworking slab mem use
- Worth noting that many proprietary mitigations don't yet have mainline equivalents (e.g. grsec's `AUTOSLAB`)
- Lag between mainline mitigation support & hardware adoption
  - E.g. Intel's CFI (CET) support was introduced in their 11th Gen CPUs (2021)

Looking Ahead | New Technologies (AKA Rust)

- Yep, it's Rust time
- Initial support released in kernel version 6.1
- Memory safety built-in as opposed to being bolted on
- Where 66% of kernel security issues are memory safety related (2019)[9]
- However, Rust is still a tool used by people, and we make mistakes!

Looking Ahead | Attitude to Security

- Finding the balance between performance/usability and security
  - When to include, and default, particular mitigations?
  - Most of the topics mentioned today have mitigations
- Fostering open and accessible environment for security research
  - Public research and sharing can drive innovation and improvements
  - Vs. malicious actors who are happy to keep all this in the shadows
  - Still friction in the handling of security fixes & disclosures

Wrapping Up

Thank You! Feel Free To @ Me Online/Offline

Resources

- https://github.com/xairy/linux-kernel-exploitation
- https://github.com/a13xp0p0v/linux-kernel-defence-map
- https://sam4k.com (any talk updates will be posted here!)
- https://codeql.github.com
- https://github.com/google/syzkaller

Refs

1. https://github.com/a13xp0p0v/kconfig-hardened-check/
2. https://cateee.net/lkddb/web-lkddb/
3. https://github.com/cloudsecurityalliance/gsd-database
4. https://github.com/torvalds/linux
5. https://blog.theori.io/research/CVE-2022-32250-linux-kernel-lpe-2022/
6. https://www.willsroot.io/2021/08/corctf-2021-fire-of-salvation-writeup.html
7. https://www.willsroot.io/2022/01/cve-2022-0185.html
8. https://exploiter.dev/blog/2022/FUSE-exploit.html
9. https://static.sched.com/hosted_files/lssna19/d6/kernel-modules-in-rust-lssna2019.pdf
10. https://github.com/thejh/linux/blob/slub-virtual-v6.1-lts/MITIGATION_README
11. https://googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html