Linux Kernel Hardening: Ten Years Deep

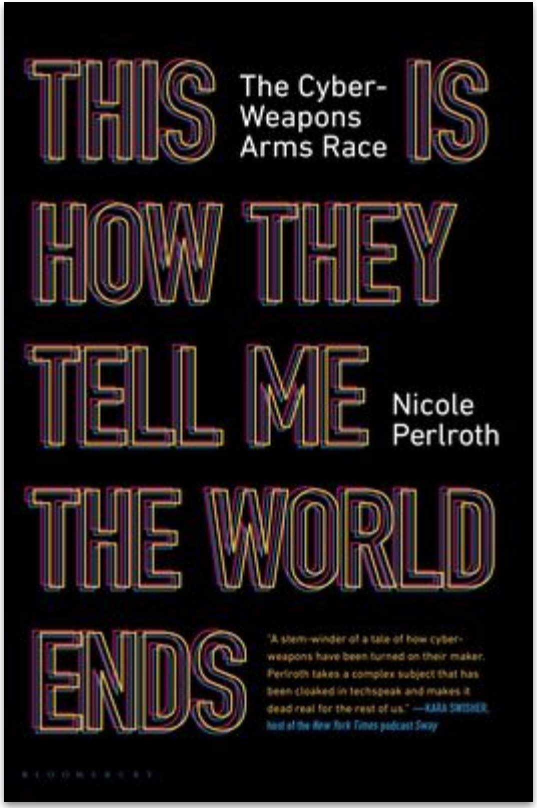
LINUX SECURITY SUMMIT
NORTH AMERICA

June 26, 2025

Kees ("Case") Cook
https://fosstodon.org/@kees
kees@kernel.org

https://outflux.net/slides/2025/lss/kspp-decade.pdf

"The most likely way for the world to be destroyed, most experts agree, is by accident.



THIS IS HOW THEY TELL ME THE WORLD ENDS

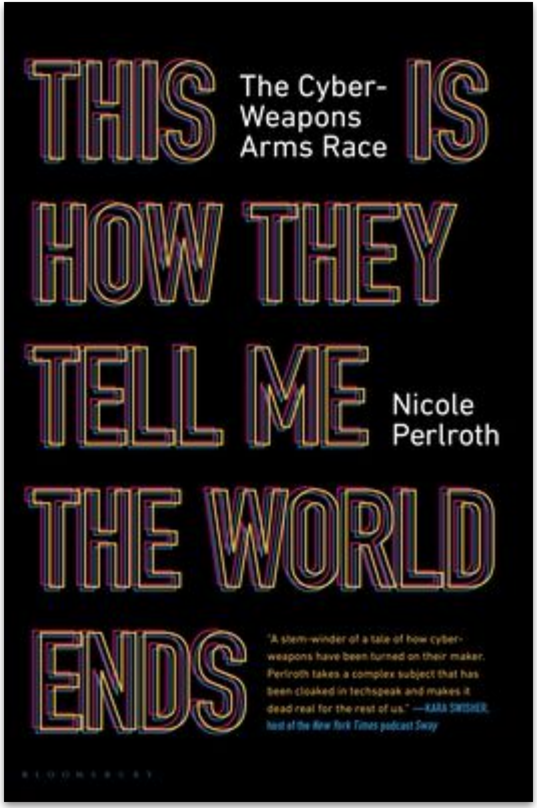
The Cyber-Weapons Arms Race

Nicole Perlroth

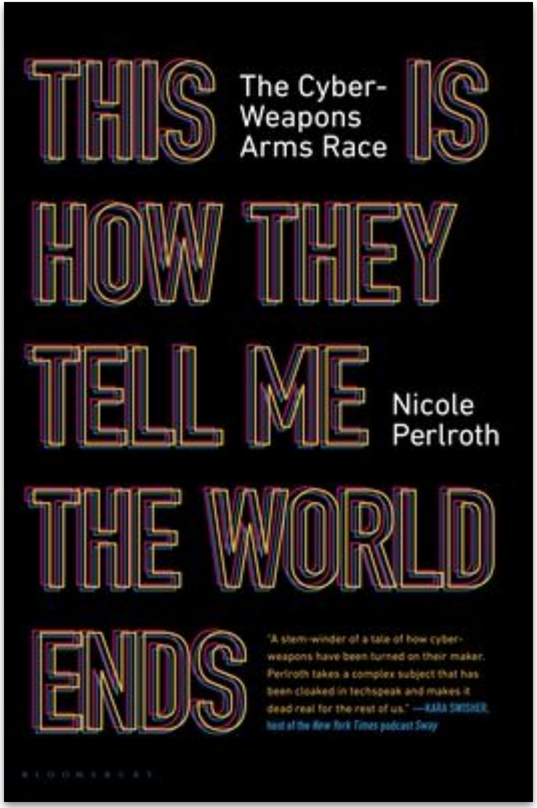
"A stem-winder of a tale of how cyber-weapons have been turned on their maker. Perlroth takes a complex subject that has been cloaked in techspeak and makes it dead real for the rest of us." —KARA SWISHER, host of the *New York Times* podcast *Sway*

BLOOMSBURY

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals.


THIS IS HOW THEY TELL ME THE WORLD ENDS
The Cyber-Weapons Arms Race
Nicole Perlroth
"A stem-winder of a tale of how cyber-weapons have been turned on their maker. Perlroth takes a complex subject that has been cloaked in techspeak and makes it dead real for the rest of us." —KARA SWISHER, host of the New York Times podcast Sway
BLOOMSBURY

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals.
We cause accidents."



THIS IS HOW THEY TELL ME THE WORLD ENDS

The Cyber-
Weapons
Arms Race

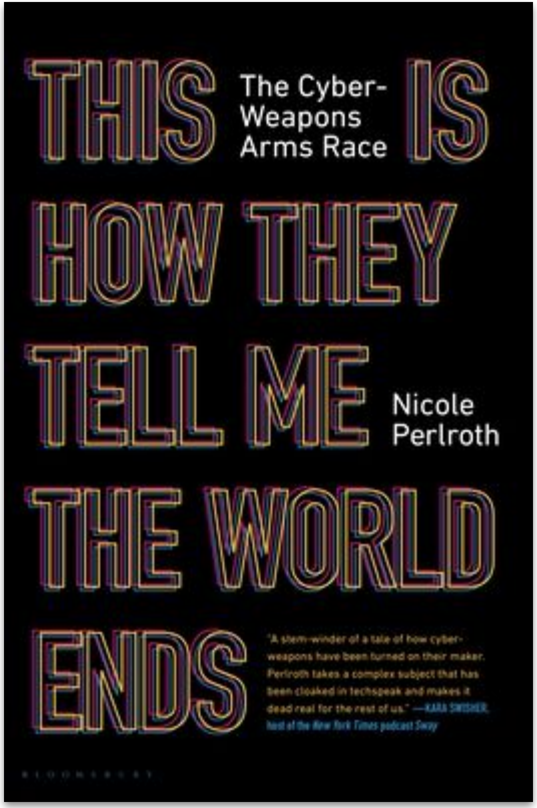
Nicole
Perlroth

"A stem-winder of a tale of how cyber-weapons have been turned on their maker. Perlroth takes a complex subject that has been cloaked in techspeak and makes it dead real for the rest of us." —KARA SWISHER, host of the *New York Times* podcast *Sway*

BLOOMSBURY

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals.
We cause accidents."

– Nathaniel Borenstein, MIME creator (as attributed by Nicole Perlroth)


THIS IS HOW THEY TELL ME THE WORLD ENDS
The Cyber-Weapons Arms Race
Nicole Perlroth
"A stem-winder of a tale of how cyber-weapons have been turned on their maker. Perlroth takes a complex subject that has been cloaked in techspeak and makes it dead real for the rest of us." —KARA SWISHER, host of the New York Times podcast Sway
BLOOMSBURY

Long lifetime of ~~accidents~~security flaws

- In 2010, Jon Corbet found the average security flaw lifetime was 5 years.
- In 2015, I got basically the same (if not worse) conclusion, even separated by severity:
  - 2 critical: 3.3 years
  - 31 high: 6.3 years
  - 297 medium: 4.9 years
  - 172 low: 5.1 years

- So, beyond just to large ***volume*** of flaws, they also had a long ***lifetime***, meaning attackers had a huge window of opportunity for any given system.

Linux Kernel Self-Protection Project

I announced the project in November 2015 (as an upstream Linux focus area), trying to gather the many disparate security improvement efforts.

Our two specific goals:

- Remove entire bug classes (stop the whack-a-mole of fixing individual bugs)
- Eliminate exploitation methods (don't make things easy for attackers)

It's been 10 years of cat herding! Beyond the technical accomplishments, we convinced the community that the work was *needed*.

Who has contributed?

A huge span of people have been helping over the years! The last time I could fit everyone on a single page was in 2017. By now, given the breadth of work, it's hard to get anything close to an accurate count, but to give a sense, it includes:

- Vendors (e.g. Intel, ARM, QualComm, IBM, Cisco, HP, Huawei, …)
- Distros (e.g. RedHat, Oracle, Canonical, …)
- Service Providers (e.g. Google, Netflix, Docker, …)
- Integrators (e.g. Linaro, GrapheneOS, Microsoft, …)
- Contractors, Researchers, and Individuals (e.g. Gustavo A.R. Silva, Nathan Chancellor, Andy Lutomirski, Alexander Popov, David Windsor, PaX Team, …)

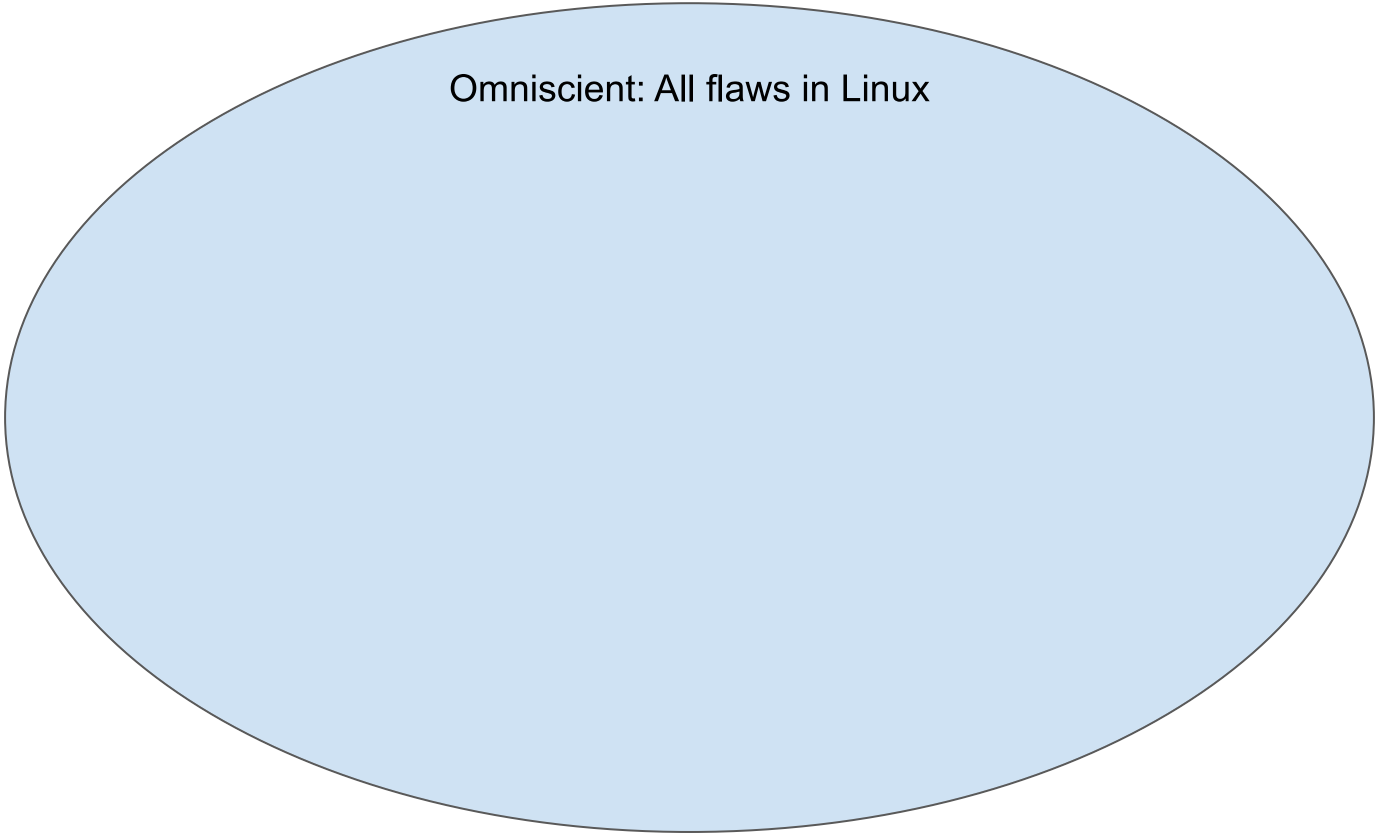
The work has had an impact!

Prove it …

But first … Linux kernel flaws and CVEs

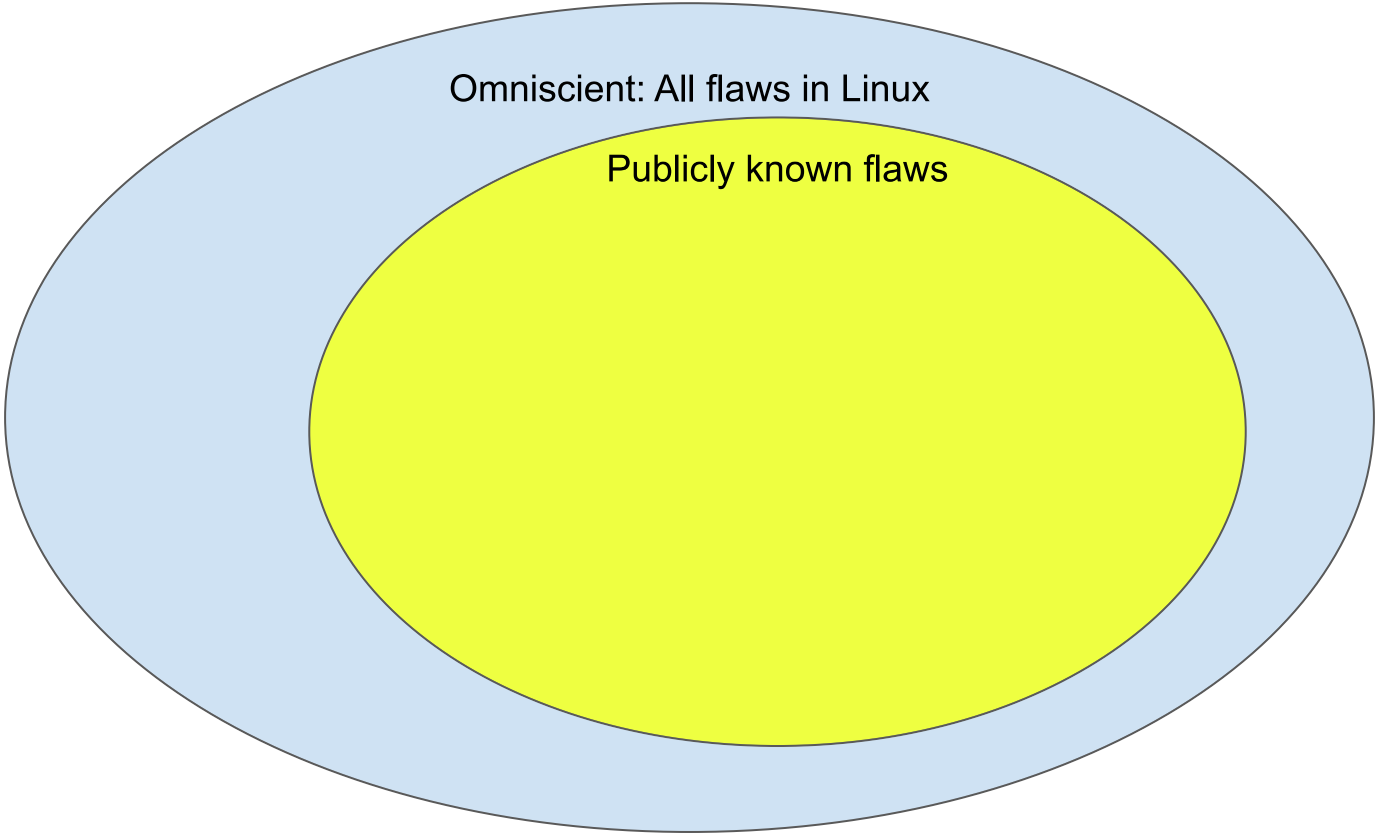
- Common Vulnerability Enumeration (maps vulnerabilities to CVE identifiers)
- Linux Kernel became its own CVE Naming Authority (CNA) in Feb 2024, which changed how CVEs got assigned.
- Prior to that, CVEs were most often assigned by general-purpose distros, and followed their threat models. (And dramatically under-counted flaws in the kernel.)

Linux Flaws Venn Diagram of Doom

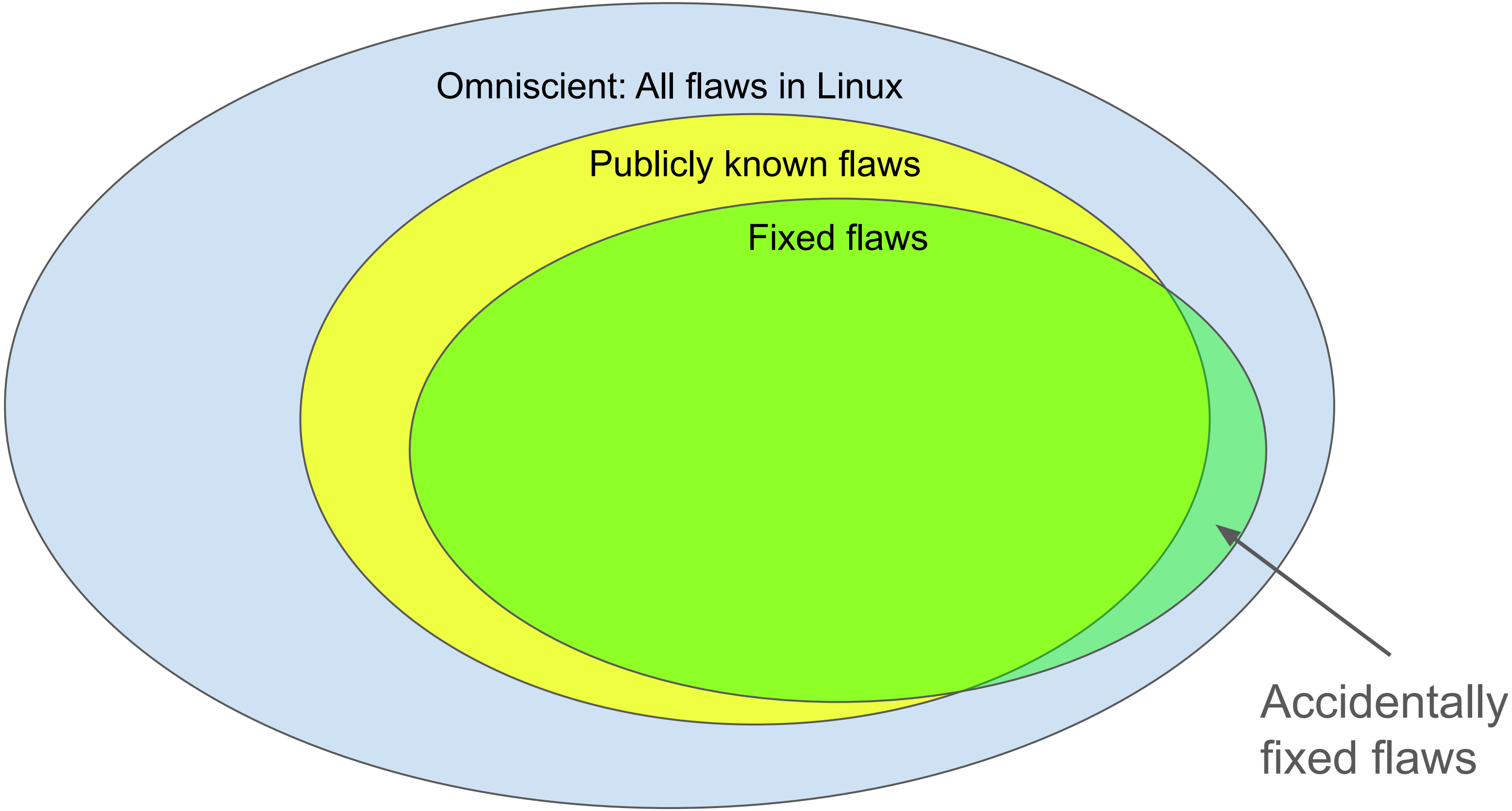


Omniscient: All flaws in Linux

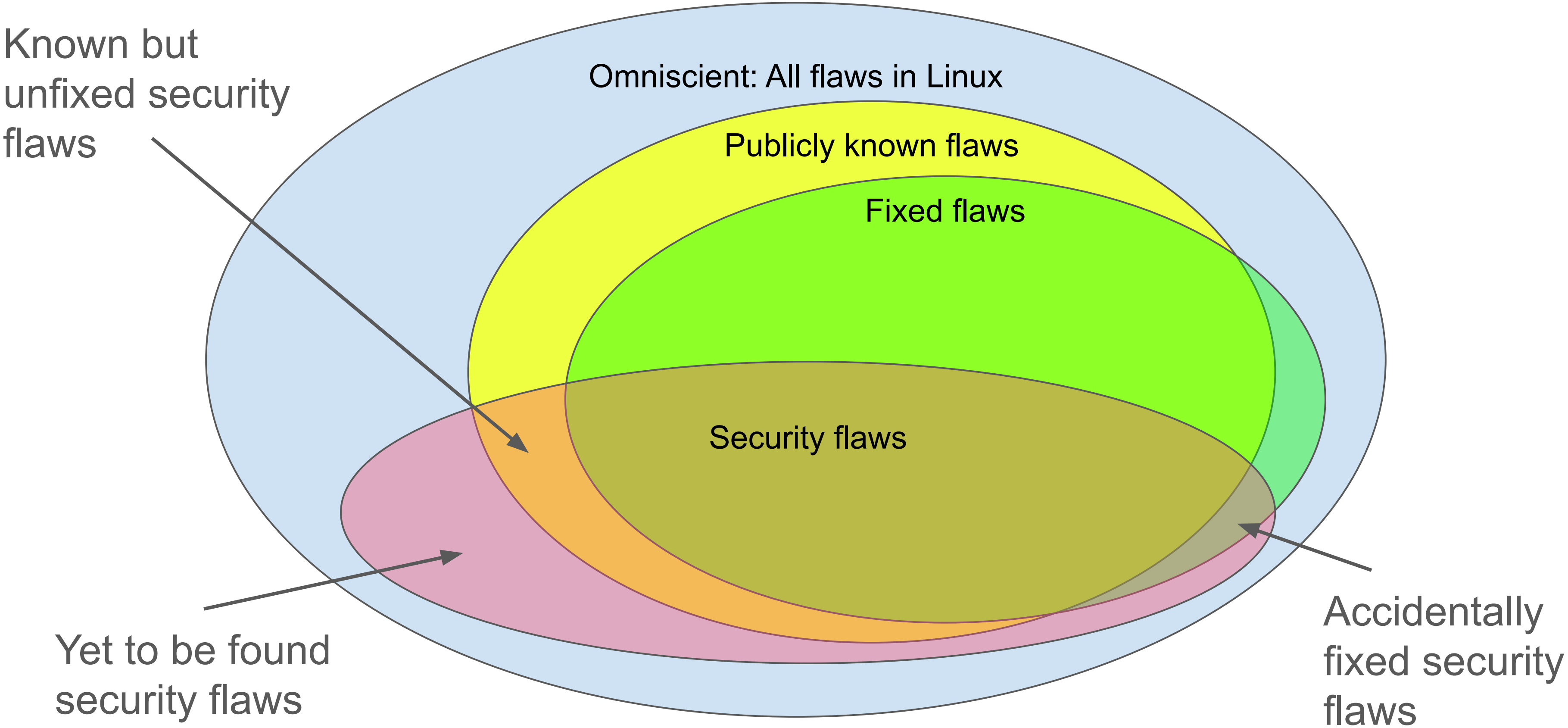
Linux Flaws Venn Diagram of Doom



Linux Flaws Venn Diagram of Doom



Linux Flaws Venn Diagram of Doom



Known but unfixed security flaws

Omniscient: All flaws in Linux

Publicly known flaws

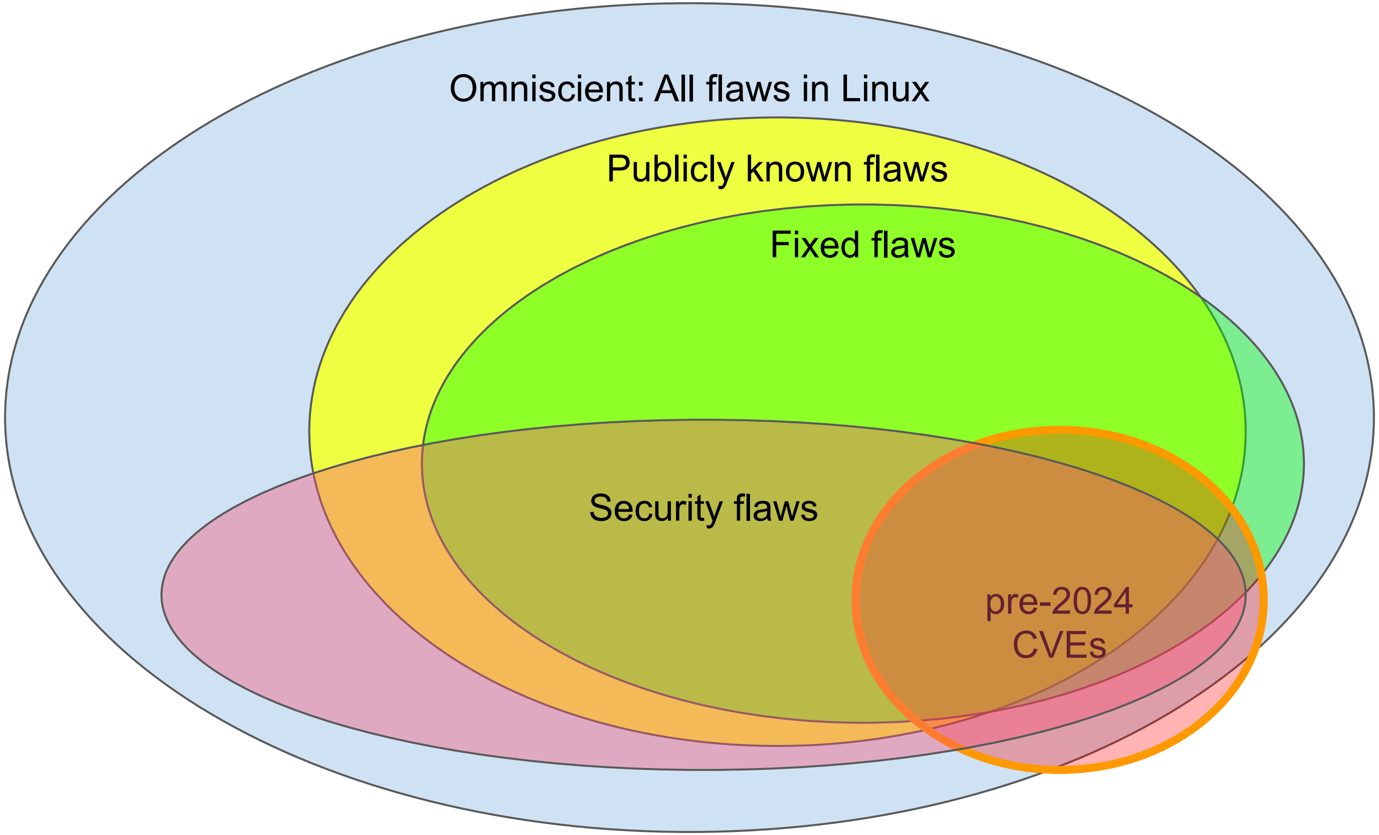
Fixed flaws

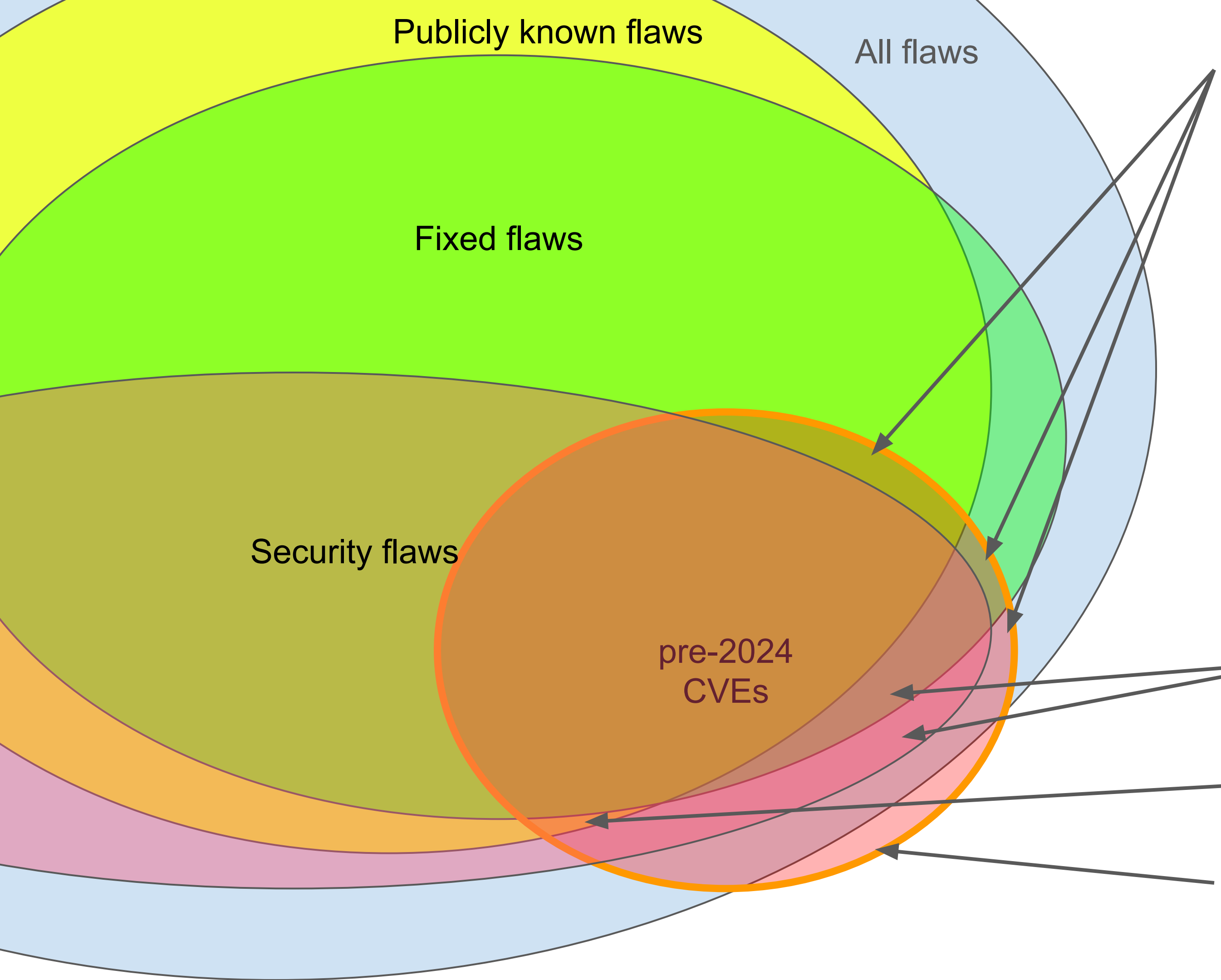
Security flaws

Yet to be found security flaws

Accidentally fixed security flaws

Linux Flaws Venn Diagram of Doom



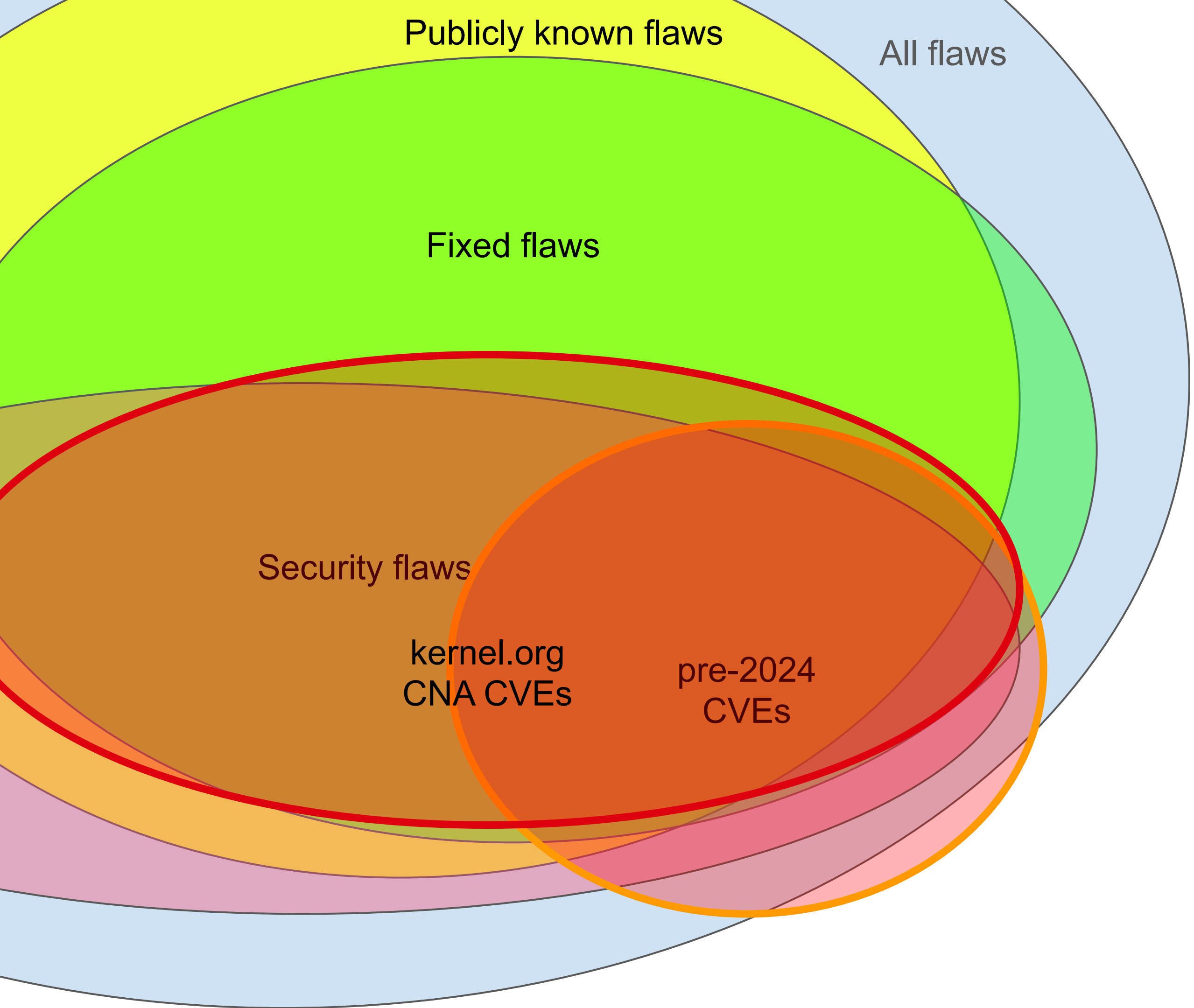


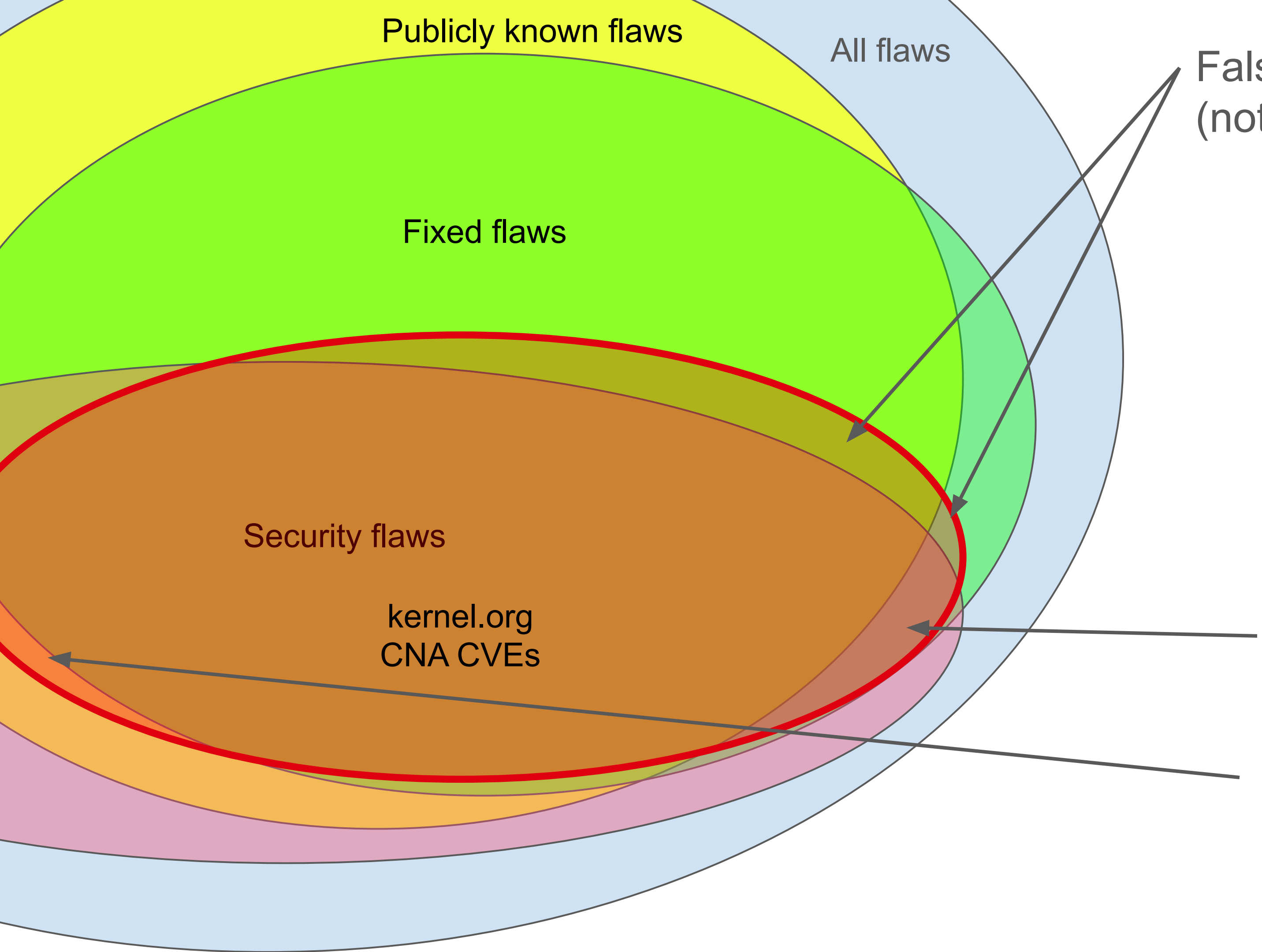
False positives
(not security flaws)

True positives
(accidentally fixed or
identified)

True positives
(unfixed)

False positives
(not flaws)


Publicly known flaws
All flaws
Fixed flaws
Security flaws
kernel.org CNA CVEs
pre-2024 CVEs

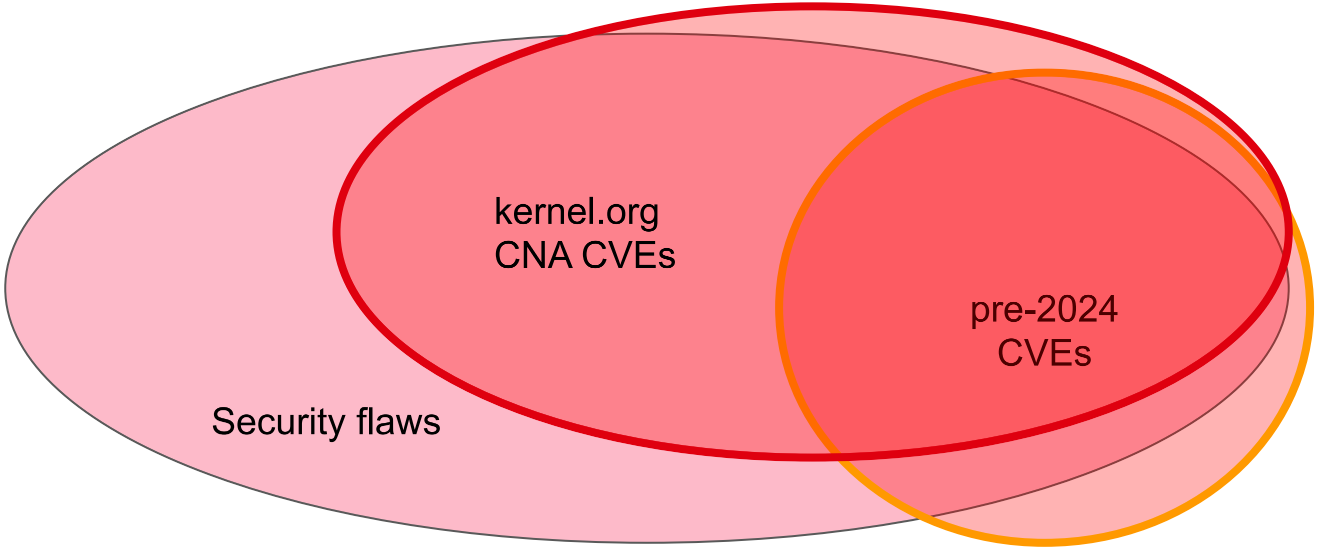


False positives
(not security flaws)

True positives
(accidentally fixed)

True positives
(unfixed)

Reminder: the goal is to fix *security flaws*, not CVEs… (kernel.org CNA CVEs match reality much better)



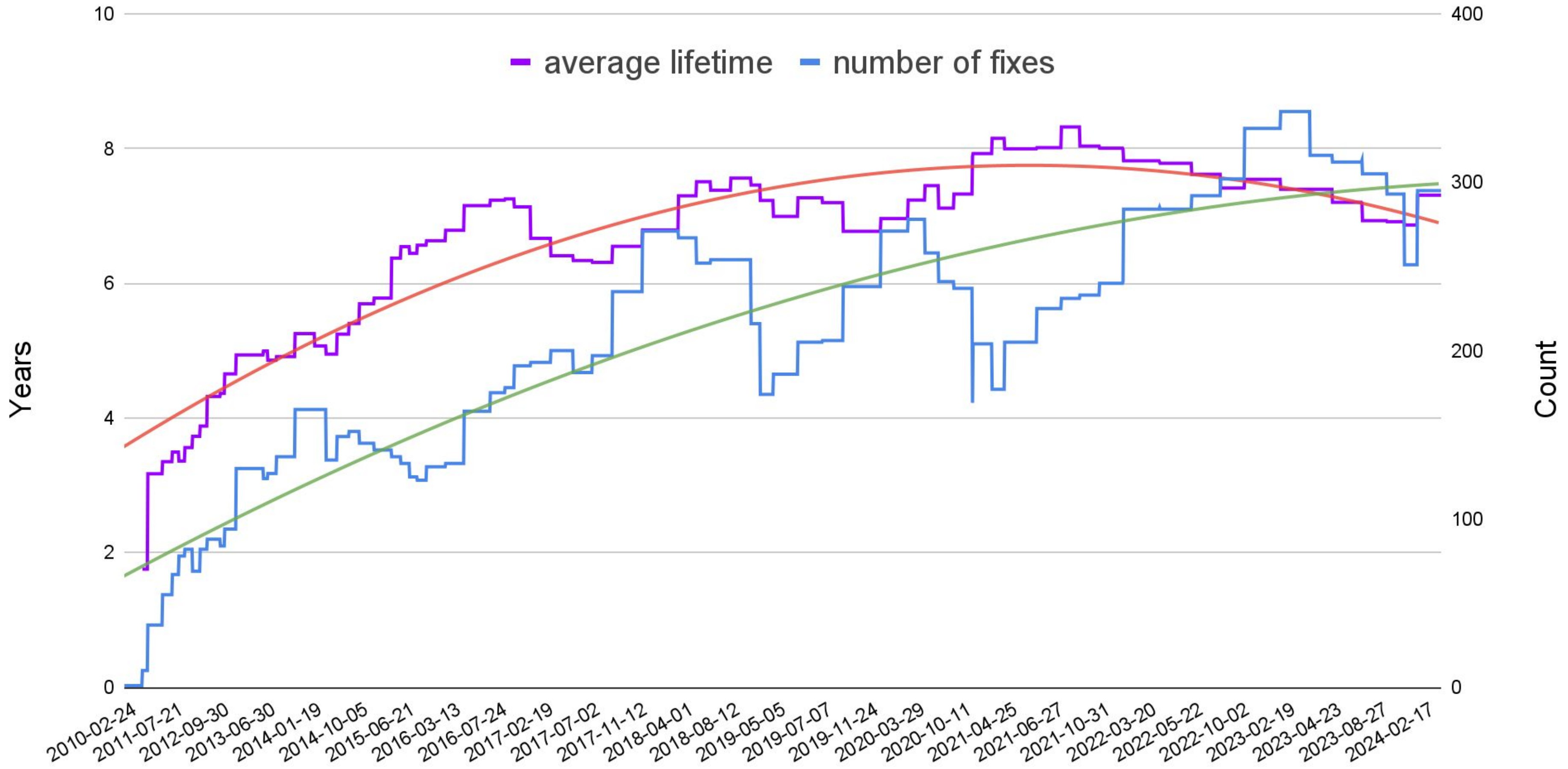
Lies, Damn Lies, and Statistics

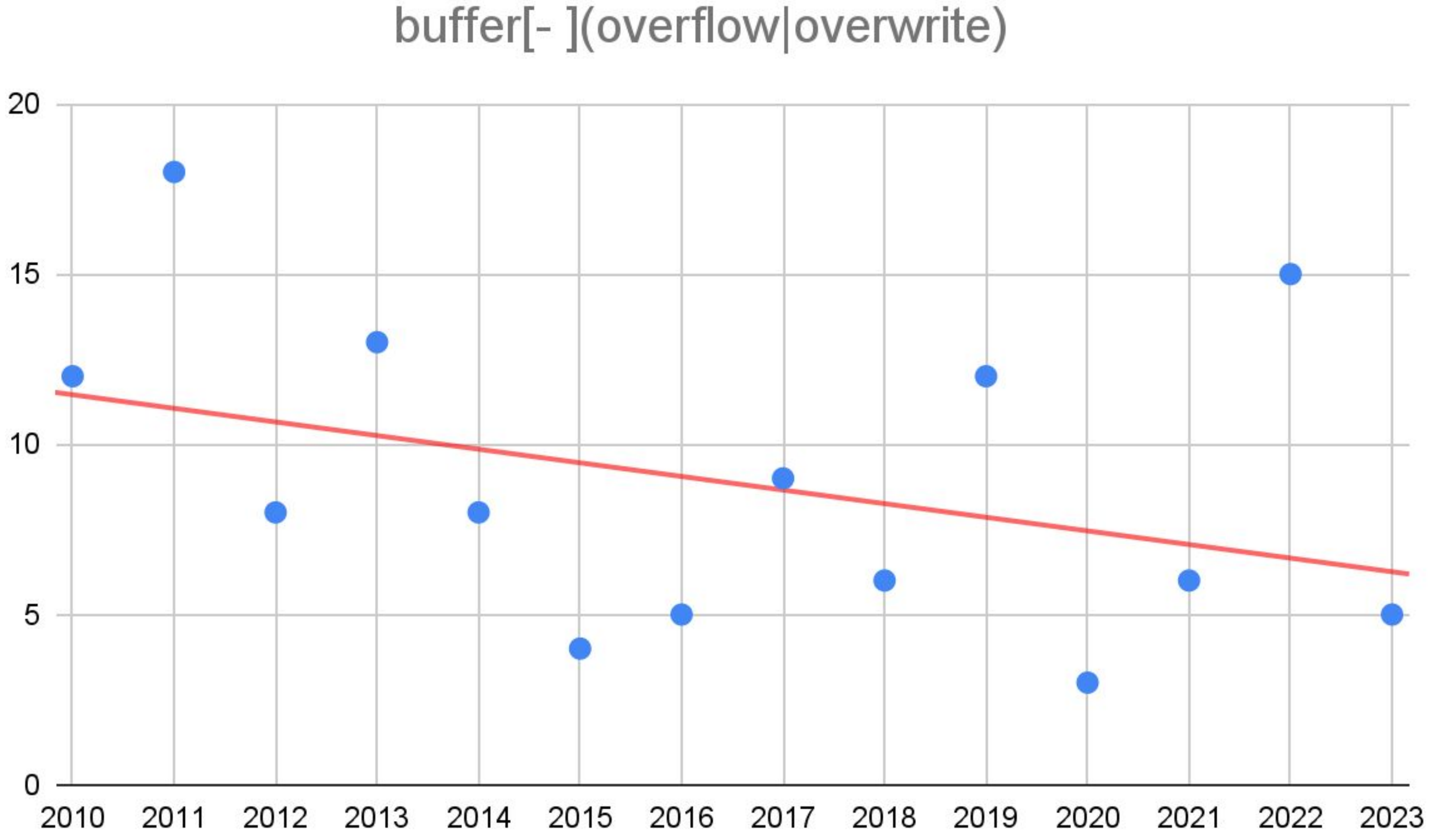
- I use the Ubuntu CVE Tracker for my vulnerability statistics – they track the commits that introduced flaws as well as commits that fixed flaws, and they assign severity. This is everything I need to examine trends and lifetimes.

- Doing a retrospective examination of CVEs across the switch between CVE assignment methods isn't going to be easy. So I won't! To get a historical sense of vulnerability class trends, I only looked at pre-CNA CVEs, going back to 2010.

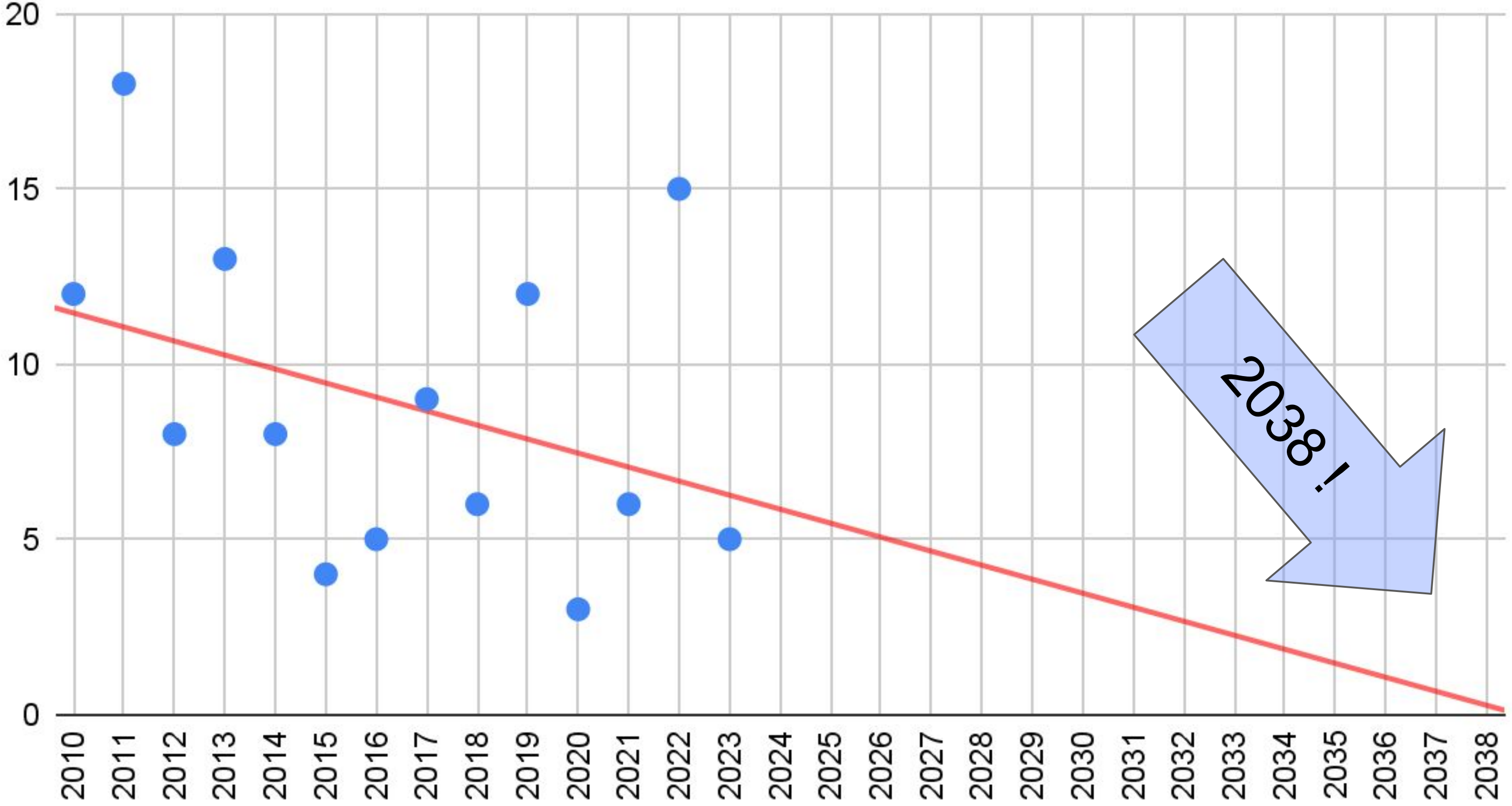
- Time for graphs …

Lifetime of fixed flaws, 1 year rolling average




buffer[-](overflow|overwrite)
20
15
10
5
0
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023

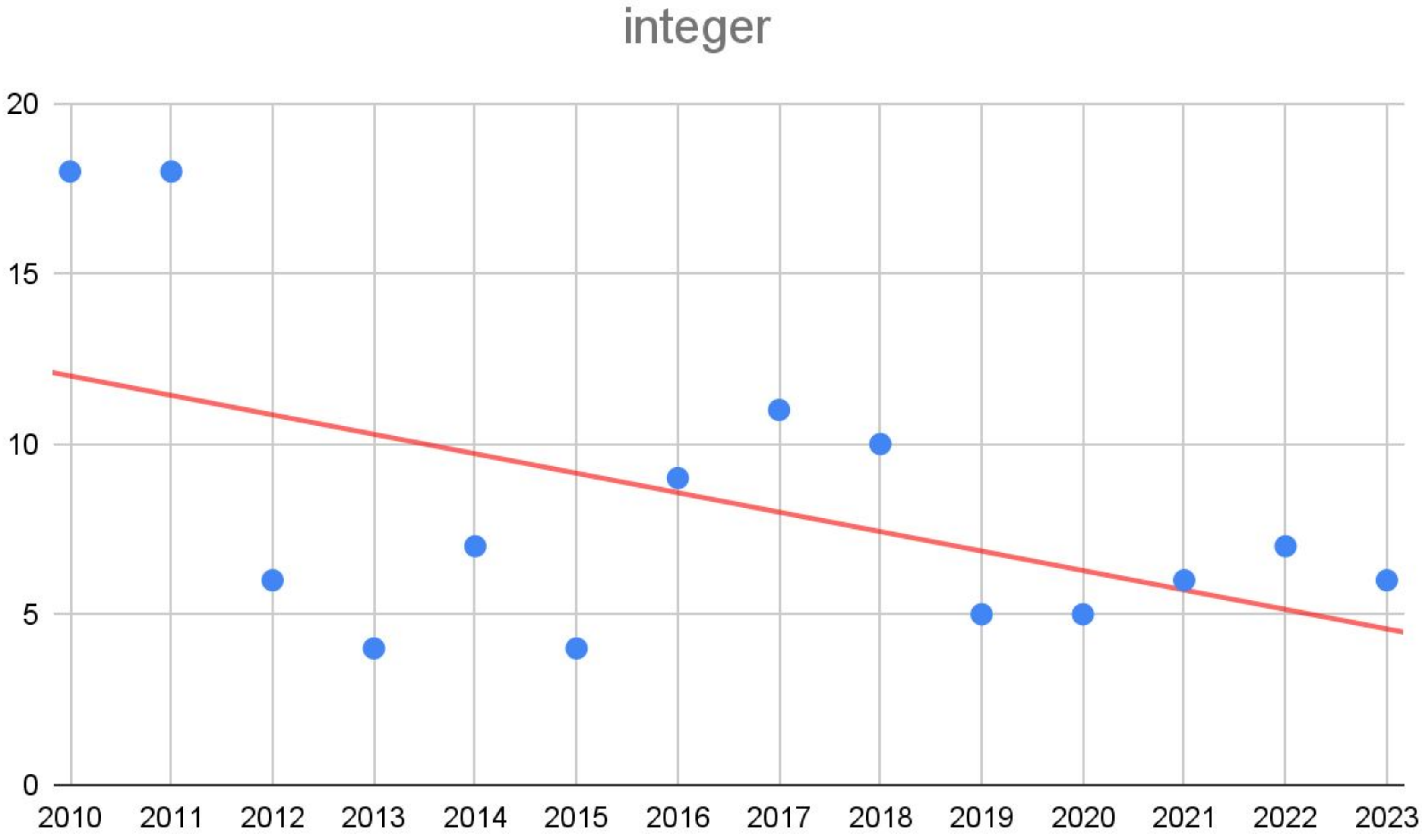
buffer[-](overflow|overwrite)



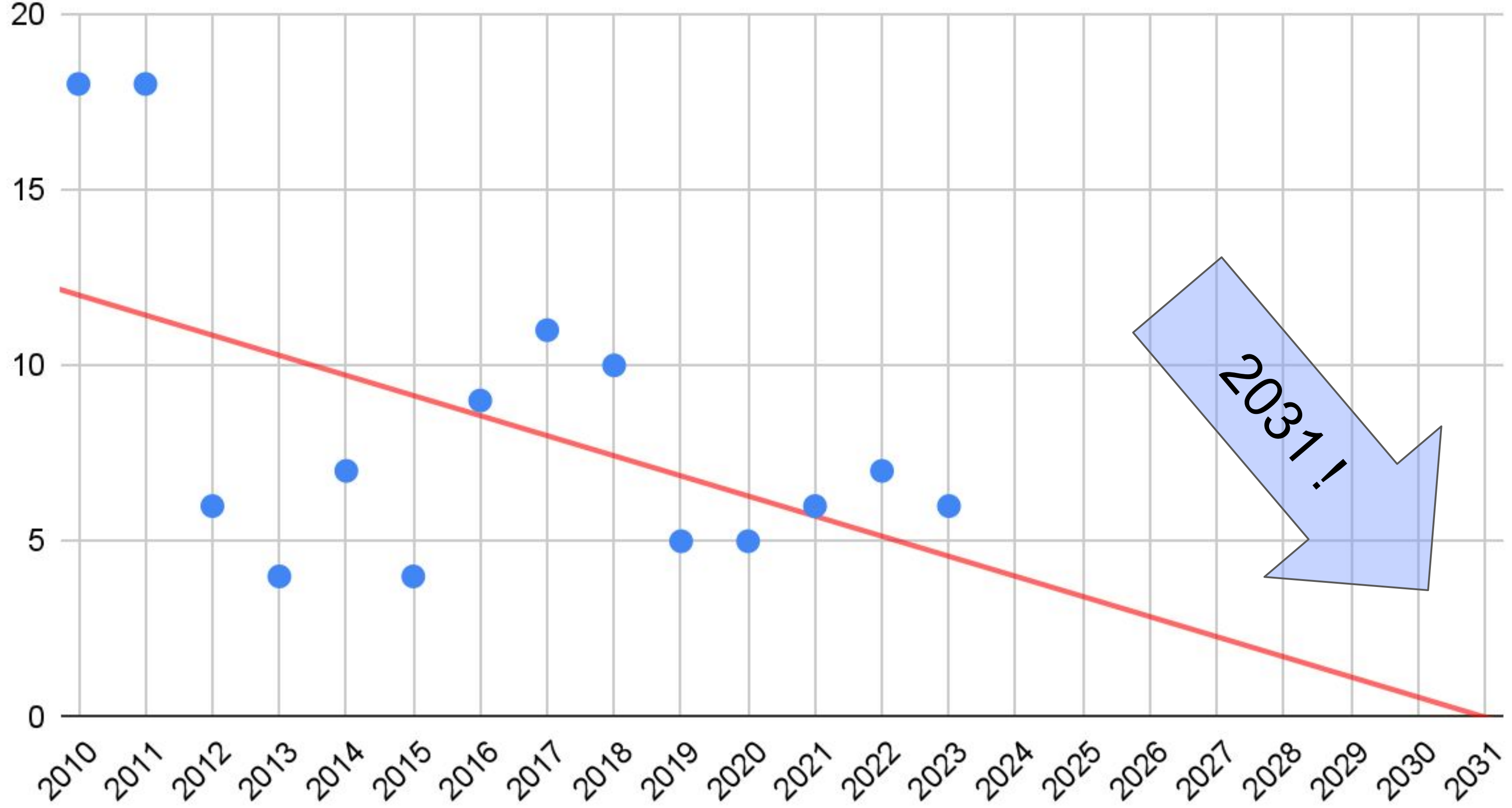
32-bit time_t Unix Epoch wrap!

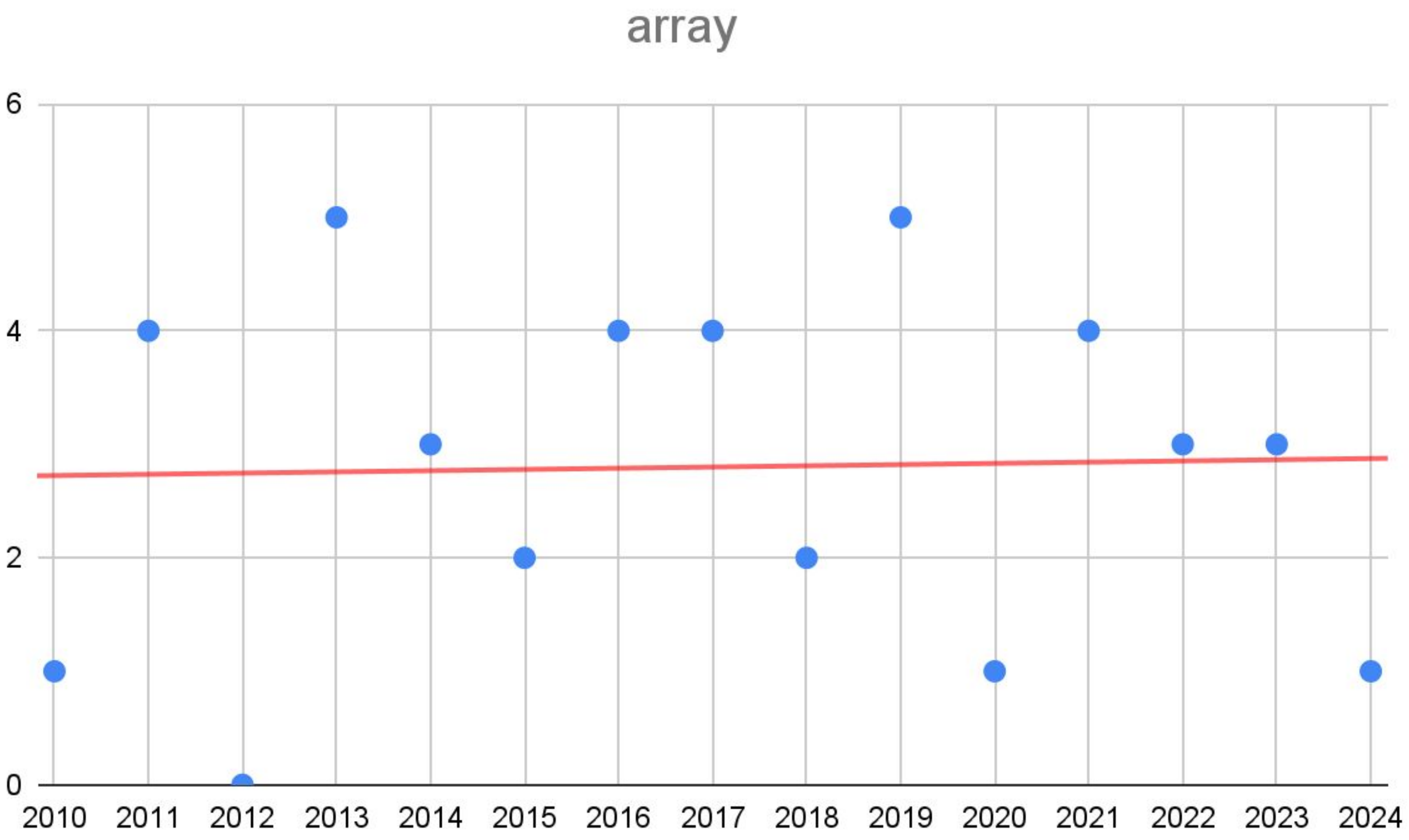
```
11111111111111111111111111111111 03:14:07 19 Jan 2038 UTC
                              +1         *tick*
--------------------------------
00000000000000000000000000000000 00:00:00  1 Jan 1970 UTC
```

So … integer overflows …


integer
20
15
10
5
0
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023

integer


20
15
10
5
0
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2031 !

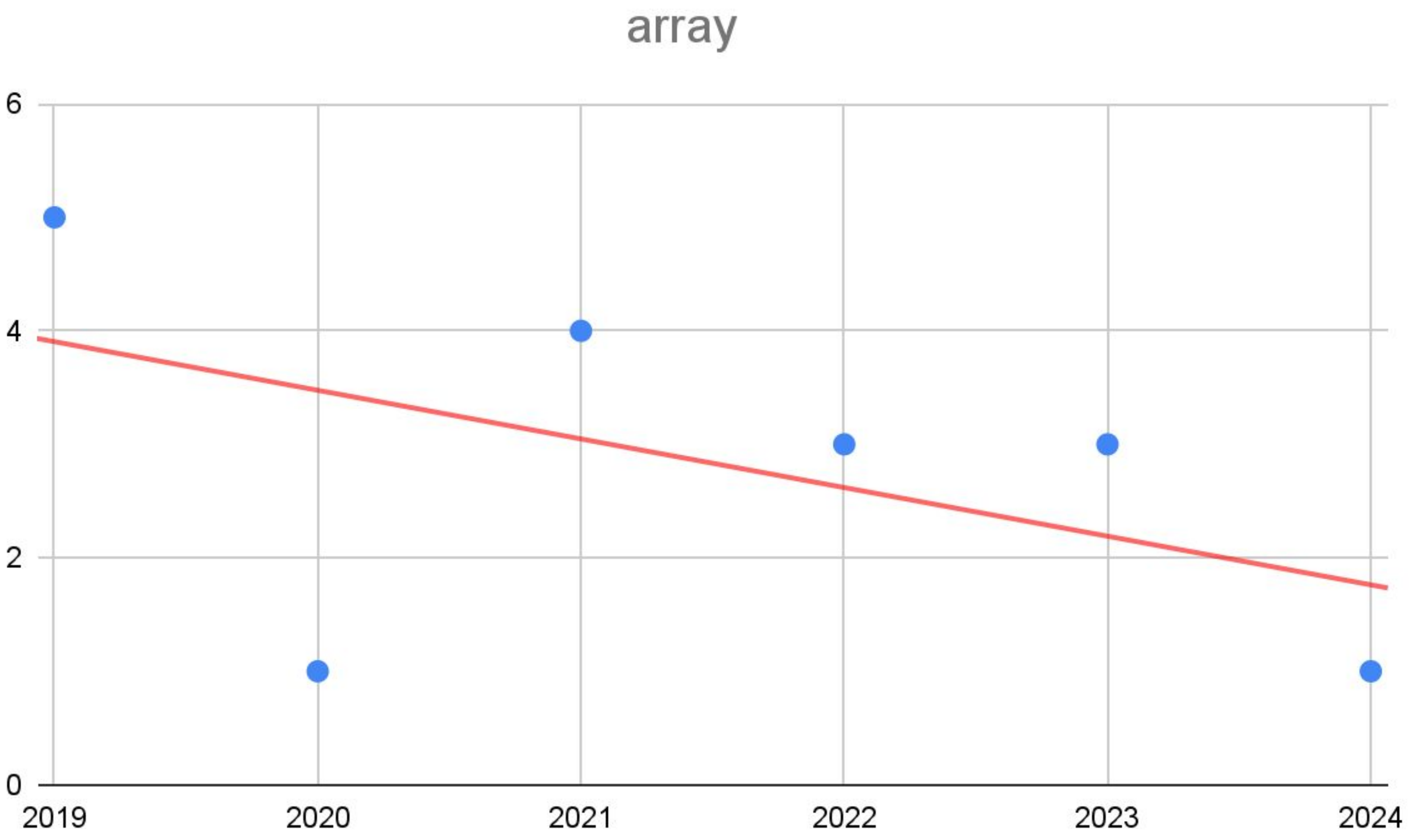

array
6
4
2
0
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024

2020: BleedingTooth

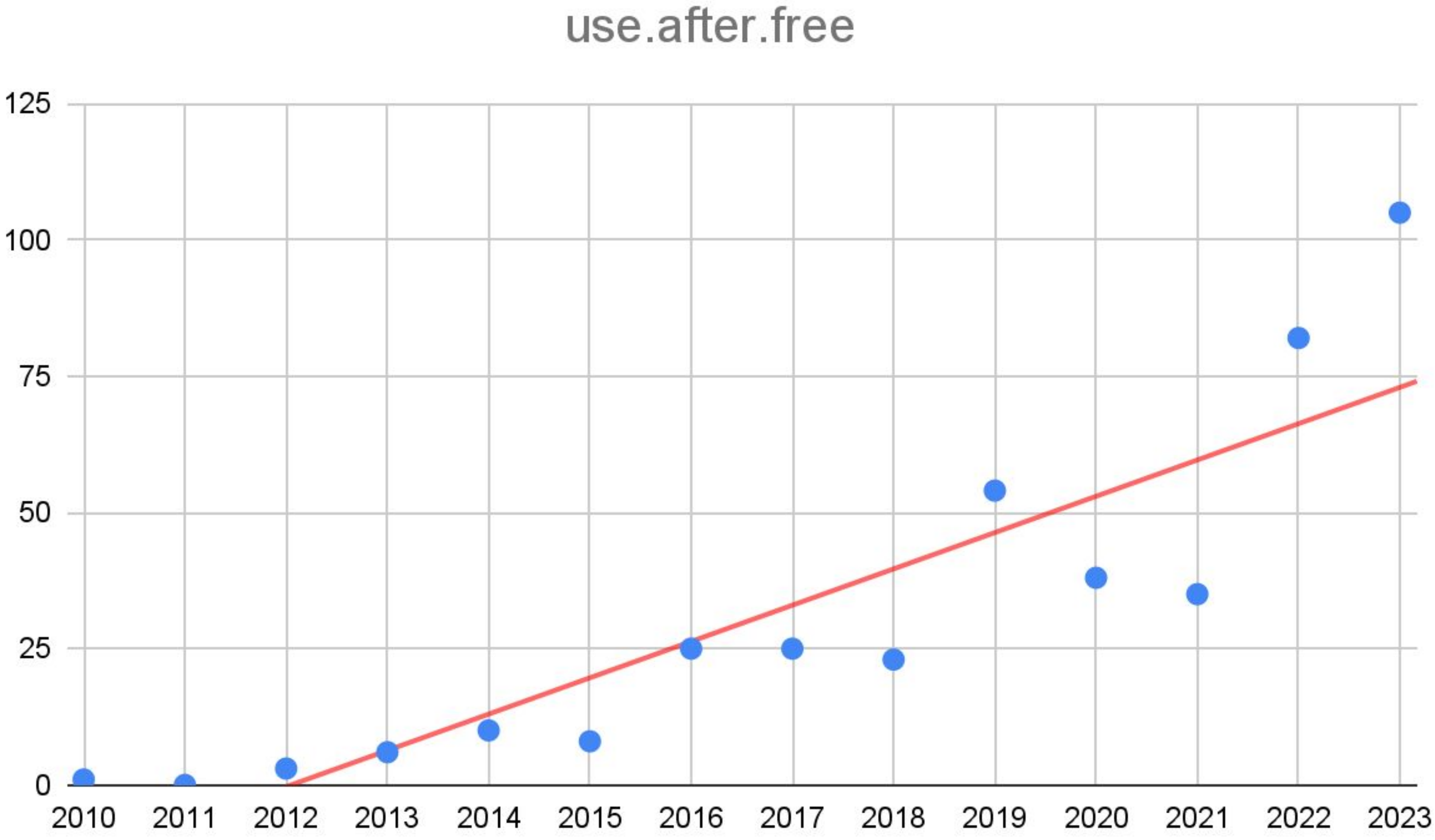
https://google.github.io/security-research/pocs/linux/bleedingtooth/writeup.html

```
struct hci_dev {
        ...
        struct discovery_state {
                ...
                u8 last_adv_data[HCI_MAX_AD_LENGTH];
                ...
        };
        ...
        struct list_head {
                struct list_head *next;
                struct list_head *prev;
        } mgmt_pending;
        ...
};
```

```
memcpy(d->last_adv_data, data, len);  /* len > HCI_MAX_AD_LENGTH ?! */
```


array
6
4
2
0
2019
2020
2021
2022
2023
2024

So where is the low hanging fruit now?



use.after.free

Where are all the Use-After-Free flaws coming from?

```
30 net/netfilter
28 net/l2tp
17 drivers/android/binder.c
16 sound/core
15 fs/ext4
14 net/sched
14 fs/io_uring.c
11 net/bluetooth
10 net/ipv4
 9 kernel/futex.c
 8 net/ax25
 7 fs/btrfs
 6 net/nfc
 6 kernel/trace
 5 net/sctp
 5 net/packet
 5 net/ipv6
```

```
5 fs/io-wq.h
5 drivers/tty/vt
5 drivers/net/hamradio
5 drivers/gpu/drm
4 net/unix
4 net/socket.c
4 fs/ntfs3
4 fs/namei.c
4 fs/eventpoll.c
4 fs/cifs
4 drivers/usb/misc
4 drivers/media/dvb-core
4 drivers/media/cec/core
4 drivers/gpu/drm/vmwgfx
4 drivers/block
3 net/xfrm
  ...
```

Use-After-Free (UAF) Research and Mitigation

- Google kernelCTF Vulnerability (and Patch) Reward Program https://google.github.io/security-research/kernelctf/rules
  - netfilter https://docs.google.com/spreadsheets/d/e/2PACX-1vS…wfvYC2oF/pubhtml
  - io_uring https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html
- Android Binder being rewritten in Rust: https://rust-for-linux.com/android-binder-driver

UAF: Type Confusion attacks

- The Linux slab allocator uses pre-chosen allocation size buckets (e.g. 96 bytes, 128 bytes, 256 bytes), so allocations in the same bucket size may come from different allocated types, including arbitrarily sized allocations (e.g. allocations via `msgsend` IPC syscall).
- Attacker finds a UAF for one structure and finds a different structure of a similar size to target.
- For lots more details, see Andrey Konovalov's excellent SLUB Internals for Exploit Developers talk during the 2024 Linux Security Summit.

UAF Mitigation: Separate type allocation buckets

- Obvious solution is to separate types so they're not all in the same buckets.
- cgroup accounting already started this accidentally (2 general bucket sets)
- `CONFIG_RANDOM_KMALLOC_CACHES` explicitly uses 16 (randomly assigned)
- `CONFIG_SLAB_BUCKETS` splits userspace allocations from kernel allocations
- proposed `CONFIG_SLAB_PER_SITE` would use a separate allocation bucket for every call site – totally isolated every kmalloc in the kernel.

UAF: Cross-allocator attacks (extreme type confusion)

- An attacker can still force slab memory to get freed back to the page allocator where it can be re-used by a different slab bucket set or the page allocator itself ("cross-cache attacks").
- This requires more work to groom the heap layout (but it ends up being a relatively deterministic methodology: see Andrey's talk).
- This has been used for fun things like forcing a virtual memory address to be reallocated as a userspace Page Table Entry, and fiddling with the hanging pointer could change memory permissions ("let's make `/etc/shadow` writable via mmap!"). See Jann Horn's detailed write-up on the approach.

UAF Mitigation: Stop virtual address reuse

- To stop cross-allocator attacks, the virtual memory addresses associated with a give allocation type need to be pinned so they cannot be reused.
- The proposed CONFIG_SLAB_VIRTUAL does this, but (unsurprisingly) come with some performance overhead and some utilization limitations.

- Best solution so far is via hardware memory tagging (e.g. ARM's MTE), so that a given allocation has bits that associated it with a given allocator, state, and/or generation. But this is limited too: there are realistically only about 4 bits left in a 64bit address that can carry these details. Protection becomes somewhat probabilistic.

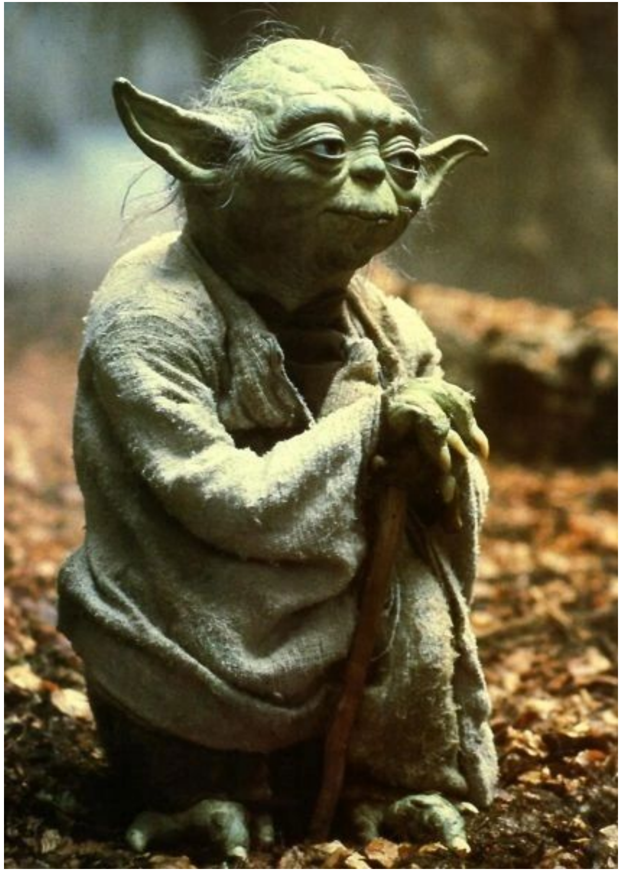
Okay, but how did we drive down the other bug classes?

- enforced memory protections (RO text, W^X, SMAP, `__ro_after_init`)
- replaced reference counters with trapping variant (`refcount_t`)
- hardened data structure integrity (`LIST, SLAB_FREELIST`)
- hardened String APIs (`USERCOPY`, `FORTIFY_SOURCE`)
- trap stack overflows (`STACKPROTECTOR_STRONG`, `VMAP_STACK`)
- obfuscated address locations (`KASLR, RANDOMIZE_KSTACK`)
- removed all Variable Length Arrays (VLAs) on the stack
- replaced open-coded allocation size arithmetic (`overflow.h`)
- replaced `set_fs()` API to avoid user/kernel address space confusions
- improved compiler to reject implicit switch case fall-throughs
- improved compiler to zero-initialize stack variables (`INIT_STACK_ALL_ZERO`)
- improved compiler to provide Control Flow Integrity (`CFI_CLANG`)
- improved compiler to actually check array sizes (`UBSAN_BOUNDS`)
- MOAR…

Okay, but how did we drive down the other bug classes?

- enforced memory protections (RO text, W^X, SMAP, `__ro_after_init`)
- replaced reference counters with trapping variant (`refcount_t`)
- hardened data structure integrity (`LIST, SLAB_FREELIST`)
- hardened String APIs (`USERCOPY`, `FORTIFY_SOURCE`)
- trap stack overflows (`STACKPROTECTOR_STRONG`, `VMAP_STACK`)
- obfuscated address locations (`KASLR, RANDOMIZE_KSTACK`)
- removed all Variable Length Arrays (VLAs) on the stack
- replaced open-coded allocation size arithmetic (`overflow.h`)
- replaced `set_fs()` API to avoid user/kernel address space confusions
- *improved compiler* to reject implicit switch case fall-throughs
- *improved compiler* to zero-initialize stack variables (`INIT_STACK_ALL_ZERO`)
- *improved compiler* to provide Control Flow Integrity (`CFI_CLANG`)
- *improved compiler* to actually check array sizes (`UBSAN_BOUNDS`)
- MOAR…

C supports ambiguity



"Ambiguity is the path to the Dark Side.

Ambiguity leads to confusion;

confusion leads to flaws;

flaws lead to suffering.

I sense much ambiguity in you."

– Yoda, about the C language

C supports ambiguity (but we can fix that)

- "*Unexpected* Behavior" is the source of **so many** flaws, a superset that includes "Undefined Behavior" which is just one special case of "language ambiguity"
- and of course the lack of memory safety, no variable lifetime enforcement, no safe concurrency

What to do about it?

- Remove ambiguity in C
- Write new stuff in Rust

With Undefined Behavior

Anything is Possible

https://raphlinus.github.io/programming/rust/2018/08/17/undefined-behavior.html

Remove Ambiguity in C
"uninitialized" stack variables

There is no such thing as "uninitialized" !

```
int function(int input)
{
    int on_the_stack;  /* contains whatever was on stack */

    return input * on_the_stack; /* returns what??? */
}
```

Remove Ambiguity in C
"uninitialized" stack variables

So now we build with `-ftrivial-auto-var-init=zero` ...

```
int function(int input)
{
    int on_the_stack;  /* contains 0 */

    return input * on_the_stack; /* returns 0 */
}
```

Some compiler folks worried "this will fork the language" … YES PLEASE

Remove Ambiguity in C
not all arrays can be bounds checked

```
struct foo {
    …
    …
    int fixed_size_array[16];
    int flexible_array[];
};
```

Can do bounds checking! (16 elements)

No dimension: no bounds checking :(

Remove Ambiguity in C
yes all arrays can be bounds checked

Now we can use the `counted_by` attribute …

```
struct foo {
    …
    int items;
    int fixed_size_array[16];
    int flexible_array[] __counted_by(items);
};
```

Can do bounds checking! (16 elements)

Can do bounds checking! ("`items`"-many elements)

Remove Ambiguity in C
next: *pointers* can be bounds checked

And coming is the `counted_by_ptr` attribute …

```
struct foo {
	…
	int num_items;
	struct item *items __counted_by_ptr(num_items);
	…
};
```

Can do bounds checking! ("`num_items`"-many struct item instances)

Remove Ambiguity in C
soon: *integers* can be bounds checked

On deck is the `__nowrap` attribute …

```
typedef __nowrap unsigned long size_t;
size_t position = 0;
position --; // boom


__nowrap u8 index = 255;
index ++; // boom
```

Will no longer silently wrap around!

Remove Ambiguity in C

The C Standard is strict, slow-moving, and prioritizes compatibility over robustness. The key to making any practical progress with GCC, Clang, and even MSVC is to use the magic phrase:

Remove Ambiguity in C*ompilers*

The C Standard is strict, slow-moving, and prioritizes compatibility over robustness. The key to making any practical progress with GCC, Clang, and even MSVC is to use the magic phrase:

I would like to add this **Language Extension** …

Then coordinate the extension between compilers, and the C Standard can catch up when they're ready.

Also: Write New Stuff in Rust :)

It's a long road, but the language bindings have been steadily landing. Entire graphics drivers have been written in Rust: Apple AGX, Nova. Also filesystems, block drivers, network PHY drivers…

You know it's time to ditch C/C++ when even governments have noticed the dumpster fire. National Security Agency (NSA), Cybersecurity and Infrastructure Security Agency (CISA), and Office of the National Cyber Director (ONCD):

The Case for Memory Safe Roadmap

Exploring Memory Safety in Critical Open Source Projects

https://rust-for-linux.com/

Thank you!

Questions / Comments?

Kees ("Case") Cook
https://fosstodon.org/@kees
kees@kernel.org

https://outflux.net/slides/2025/lss/kspp-decade.pdf