TU Graz

SCIENCE
PASSION
TECHNOLOGY

Introduction to Linux Kernel Exploitation

Featuring CVE-2026-31431

Ernesto Martínez García

Pentesting Lab SS26

$ whoami

> I am Ernesto Martínez García, PhD Student from **SESYS** (Team Mangard)

My research focuses on the Linux Kernel: side channels, allocator issues, etc.

Contact & Website:

- ernesto.martinezgarcia@tugraz.at
- ecomaikgolf.com

LosFuzzys

Student Capture the Flag (CTF) team from ISEC TUGraz

We are looking for people like you :) Very chill I promise

Join: https://discord.gg/RrUKAvGB2N & Wednesdays @ 18:15 FuzzyLab

Table of Contents


Workbench


Kernel Interaction

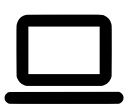

Kernel Bugs


Kernel Exploitation


Defenses


CVE-2026-31431

Workbench



Source Code

> The Linux kernel can be downloaded from `kernel.org`

You can either clone the repo or download a specific tarball version:

```
1 git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
2 wget 'https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.19.9.tar.xz'
```

You may find `git worktree` and `git blame` convenient

To obtain the Android common source code:

```
1 git clone https://android.googlesource.com/kernel/common
```

To get the full view, you should combine it with the mailing list: https://lkml.org/

And probably a good plaintext email reading setup: https://useplaintext.email/

Finally, for keeping up with news, the newsletter: https://lwn.net/

Buildsystem & KConfig I

> The Linux Kernel is built with makefiles and configured with KConfig

Before building the kernel, you need a configuration in `.config`:

```
CONFIG_CC_VERSION_TEXT="gcc (GCC) 16.0.1 20260321 (Red Hat 16.0.1-0)"
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=160001
...
```

It's just a list of variables that configure what your kernel needs:

- Support for USB Drawing Tablet XYZ
- Security Feature ABC

You can inspect your current config with:

```
cat /boot/config-$(uname -r)
```

Buildsystem & KConfig II

> Once the kernel has a `.config` available, it can proceed to build

You can generate the default `.config` with:

```
1 make ARCH=arm64 defconfig
```

And then build the kernel with:

```
1 make -j8
```

Generating the relevant files in:

```
1 vmlinux
2 arch/x86_64/boot/bzImage
```

Language Server Protocol (LSP)

> As the code is complicated, you'll probably want to use an LSP

There are multiple ways of doing it, I build with `llvm`:

```
1 make LLVM=1 -j8
```

And then use the provided script:

```
1 ./scripts/clang-tools/gen_compile_commands.py
```

Generating a `.compile_commands.json` that `clangd` can interpret

Warning: until `clangd` indexes the kernel you'll get a nice 100% CPU usage

But all `find-references` and `goto-*` should work like a charm now :)

Running

> You can run your built kernel virtualized or bare metal

To run it virtualized, you'll need a `rootfs` image (a filesystem):

You can get it from Buildroot, Docker exports, etc.

Then:

```
qemu-system-x86_64                                                    \
  -enable-kvm -cpu host -smp cores=2 -m 4G                            \
  -nographic                                                          \
  -no-reboot                                                          \
  -kernel ${CUR}/kernel/${KERNEL}/arch/x86_64/boot/bzImage            \
  -drive file=${CUR}/rootfs/${ROOTFS}/rootfs.qcow2,format=qcow2       \
  -append "root=/dev/sda console=ttyS0"
```

Furthermore you can configure SSH and folder sharing

Debugging I

> To debug the kernel, you can leverage QEMU's GDB server + kernel scripts

First, I would recommend building the kernel with the following settings enabled:

```
./scripts/config -e DEBUG_INFO_DWARF5 -e GDB_SCRIPTS -e FRAME_POINTER
```

Then, run the following command:

```
make scripts_gdb
$ file ./scripts/gdb/vmlinux-gdb.py
```

And add the following kernel parameter to the kernel boot command line:

```
nokaslr
```

Now we continue with QEMU

Debugging II

> With the debug kernel built, we use QEMU's GDB server

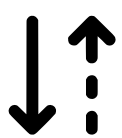
```
qemu-system-x86_64                                                    \
-enable-kvm -cpu host -smp cores=2 -m 4G                              \
-nographic                                                            \
-no-reboot                                                            \
-kernel ${CUR}/kernel/${KERNEL}/arch/x86_64/boot/bzImage              \
-drive file=${CUR}/rootfs/${ROOTFS}/rootfs.qcow2,format=qcow2         \
-S -s                                                                 \
-append "root=/dev/sda console=ttyS0 mitigations=off kpti=0 hash_pointers=never nokaslr"
```

On a separate terminal, connect to the debugger:

```
cd ${KERNEL}; gdb -ex 'target remote localhost:1234' vmlinux
```

A good setup is a big part of it: https://xairy.io/articles/pixel-kgdb

Kernel Interaction



Interaction

> The kernel exposes itself to userspace via different mechanisms

We will cover the main ones:

- `syscalls`
- `ioctls`
- `sysfs` (`/sys`)
- `procfs` (`/proc`)

Knowing the mechanisms is important for exploitation.

Different than userspace exploitation:

Maybe you have a vulnerability but you can’t reach it, or its unreliable

syscalls

> Generic interface for requesting kernel

The kernel holds a generated **switch** statement for syscall numbers:

```
1 long x64_sys_call(const struct pt_regs *regs, unsigned int nr) {
2   switch (nr) {
3     #include <asm/syscalls_64.h>
4     default: return __x64_sys_ni_syscall(regs);
5   }
6 }
```

Where syscalls are defined in `arch/x86/include/generated/asm/syscalls_64.h`:

```
1 __SYSCALL(0, sys_read)
2 __SYSCALL(1, sys_write)
3 __SYSCALL(2, sys_open)
```

syscalls

> You can list all `syscalls` via manual pages:

```
$ man 2 syscalls
...
System call                 Kernel        Notes
-----------------------------------------------
accept(2)                   2.0           See notes on socketcall(2)
accept4(2)                  2.6.28
access(2)                   1.0
acct(2)                     1.0
...
```

Then:

```
$ man 2 accept
...
```

ioctl

> We don't want to clutter the kernel with a syscall for every action

Problem: Vendor A needs an interface for doing B on a niche device C

New syscall? How many syscalls? How to assign syscall numbers?

ioctl is "generic" syscall, heavily used to interact with devices:

```
1 static long foo_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
2   switch (cmd) {
3   case FOO: {
4     if (copy_from_user(&args, (void __user *)arg, sizeof(args)))
5       return -EFAULT;
6     ...
7   }
8   case BAR: {
```

`ioctl`

> We can define commands for a `fd`, usually a kernel-represented device

Use case: Vendor XYZ built an NPU accelerator for Android devices

Vendor develops kernel module **`npu.ko`** and exposes **`/dev/vertex`**

Userspace can use hundreds of **`ioctl`** commands to interact with the device:

- `CMD_NPU_INIT`
- `CMD_NPU_DO_WHATEVER`
- `CMD_NPU_DO_WHATEVER2`

Simple, extensible and convenient way of exposing kernel functionalities

Thosands of developers of all kinds, thousands of **vulnerabilities**:

- `CMD_FREE` and `CMD_USE` not locking
- `CMD_XYZ` unchecked argument, etc

The procfs interface: `/proc`

> **procfs** is a RAM-backed virtual filesystem exporting runtime information

Contains the famous `/proc/<pid>/*` runtime information for processes

But also contains aggregated information about the kernel:

```
1 [root@laptop ~/]$ cat /proc/slabinfo
2 kmalloc-8k            48     48   8192    4    8 : [...]    12     12      0
3 kmalloc-4k           181    184   4096    8    8 : [...]    23     23      0
```

More examples:

```
1 [root@laptop ~/]$ cat /proc/kallsyms
2 ffffffffc13fc010 T kvm_x86_init [kvm]
3 ffffffffc13fc010 T kvm_x86_init [kvm]
```

Lots of unstructured information

The procfs interface: `/proc`


```
/proc/modules - loaded kernel modules.
/proc/kallsyms - kernel symbol table
/proc/kcore - ELF view of kernel virtual memory
/proc/kmsg - kernel log stream
/proc/sys/kernel/core_pattern - kernel coredump command
/proc/sys/kernel/tainted - kernel tainted check
/proc/crypto - registered crypto algorithms
/proc/cmdline - kernel boot parameters
/proc/version - kernel version & more
```

...

The sysfs interface: `/sys`

> **sysfs** is a RAM-backed virtual filesystem exporting kernel information

Similar to **procfs** but (tries to be) more structured:

```
1 [ecomaikgolf@laptop ~/]$ sudo cat /sys/class/net/wlo1/statistics/tx_bytes
2 419716335
```

Usually, files contain a single value or number, with very specific paths.

Userspace can just `ls+open+read` precise kernel-backed information.

Certain files can be written to, tweaking the kernel:

```
1 [root@laptop ~/]$ echo 1 > /sys/kernel/mm/ksm/run
```

or

```
1 [ecomaikgolf@laptop ~/]$ sudo sysctl kernel.kptr_restrict=0
```

The sysfs interface: `/sys`


```
/sys/bus/pci/devices/0000:00:1f.6/vendor - vendor for a PCI device
/sys/class/power_supply/BAT0/status - status for BAT0 specifically
```

See how requests are very specific and provide a single value:

```
[ecomaikgolf@laptop ~/]$ cat /sys/class/block/nvme0n1/size
2000409264
```

Should not be slept on! Convenient for **side channels and leakages**

Extra tip: you can even read your Windows OEM License key:

```
[ecomaikgolf@laptop ~/]$ sudo strings /sys/firmware/acpi/tables/MSDM
MSDMU
_ASUS_Notebook
ASUS
TJX6J-N4PGT-QXFD4-QV6TP-7CFJW # Not a real one!
```

Kernel Bugs

Classic Bugs

> As with every piece of software, we have the classic bugs:

- Out of Bounds Reads
- Out of Bounds Writes
- Use After Frees
- Double Frees
- Invalid Frees
- Uninitialized Memory
- Integer Overflows
- Race Conditions

Nothing new I hope :)

New not-so-new Bugs I

> Imagine the kernel providing an interface such as:

```
1 long foo_ioctl(struct file *f, unsigned int cmd, unsigned long arg){
2   ...
3   case CMD_FOO:
4     ...
5     hw_accelerated_foo(arg->dst, arg->src); // process src, write to dst
6 }
```

An attacker could trick the kernel with `arg->src` being a kernel address.

If `arg->src = 0xffffffffc03d1e6c` then we have a kernel read!

If `arg->dst = 0xfff...` then we have a kernel write!

Kernel must be careful when accessing user data: `access_ok()`, `copy_from_user()`.

New not-so-new Bugs II

> Similarly, you can't trust legitimate references to userspace:

```
1 long foo_ioctl(struct file *f, unsigned int cmd, unsigned long arg){
2   char buf[N];
3   ...
4   case CMD_FOO:
5     struct foo *foo = (struct foo*)arg; // assume valid pointer
6     if(foo->size > N) return -1;
7     ...
8 }
```

An attacker could replace `arg->size` before/after the check

Similar to userspace exploitation, always make copies

Here, consequences might be worse!

New not-so-new Bugs III

> Even in-bound writes can be wrong

Userspace has a simplified view of the memory, a heap buffer is a heap buffer.

The kernel has a more complicated view of the memory:

- Some pages are transparently shared between users
- Some pages should never be written to
- Some pages belong to userspace
- Some pages belong to the page cache or the slab allocator

Mistakenly using certain memory for read/write buffers can be fatal.

Even if the write is in-bounds and sanitizers won't report it.

Later we'll see an example :)

Kernel Exploitation

Arbitrary Code Execution

> Assume we have a bug granting code execution (e.g. ROP)

How can we exploit this bug from within the kernel?

We can try to run the kernel side of `setuid(0)` minus the permission checks :)

The kernel holds the permissions of a process (task) on a `struct cred`:

```
1  [ecomaikgolf@laptop ~/]$ sudo pahole cred
2  struct cred {
3    atomic_long_t             usage;           /*     0     8 */
4    kuid_t                    uid;             /*     8     4 */
5    kgid_t                    gid;             /*    12     4 */
6    ...
7    kgid_t                    fsgid;           /*    36     4 */
8    unsigned int              securebits;      /*    40     4 */
```

Arbitrary Code Execution

> Now we construct the shellcode for the ROP chain (or similar)

The kernel changes the privilege level by installing a new `struct cred`:

```
1 // commit_creds - Install new credentials upon the current task
2 int commit_creds(struct cred *new) {
3 ...
```

We would still need to allocate or reference a `struct cred`:

```
1 //prepare_kernel_cred - Prepare a set of credentials for a kernel service
2 struct cred *prepare_kernel_cred(struct task_struct *daemon) {
3 ...
```

Originally, `commit_creds(prepare_kernel_cred(NULL))` in shellcode would work

Nowadays, `commit_creds(prepare_kernel_cred(&init_task))` is the replacement

Arbitrary Address Write I: modprobe

> Assume we have an Arbitrary Address Write (AAW)

How can we exploit this bug from within the kernel?

One well-used trick is the `modprobe_path` trick.

If kernel needs to load a module on-demand, it will ask userspace to do so!

usermodehelper + modprobe

`usermodehelper` is a kernel technique to run userspace code on its behalf

`modprobe` uses it when the kernel needs to load a kernel module

The module is inserted via normal userspace `/usr/bin/modprobe`

Can't the kernel do it? Userspace admin. policies, code complexity, etc.

Arbitrary Address Write I: modprobe

> The binary of the helper is stored in a writeable global

Modprobe internals show how it calls `modprobe_path`:

```
1  char modprobe_path[KMOD_PATH_LEN] = CONFIG_MODPROBE_PATH;
2  ...
3  static int call_modprobe(char *orig_module_name, int wait) {
4    ...
5    static char *envp[] = { "HOME=/", "TERM=linux", "PATH=/sbin:/usr/sbin:/bin:/usr/bin" ...
6    ...
7    argv[0] = modprobe_path;
8    argv[1] = "-q"; argv[2] = "--"; argv[3] = module_name argv[4] = NULL;
9    ...
10   info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL, ...
```

The kernel calls it, so its run as **root**

Arbitrary Address Write I: modprobe

> So what can we do if we can overwrite it?

1. Create a shell script in `/tmp/a`
2. Write any code you need to run as root (e.g. change password)
3. Overwrite `modprobe_path` with `/tmp/a\x00`
4. Force the kernel to call `modprobe`

How can we force the `call_modprobe` codepath from userspace?

Run an unknown filetype! The kernel will try to load a module for it:

```
1 user$ echo -ne '\xff\xff\xff\xff' > /tmp/b
2 user$ chmod +x /tmp/b
3 user$ /tmp/b # kernel -> unknown magic bytes -> modprobe
4 kernel$ call_modprobe -> /tmp/a -> give $USER root
5 user$ su
```

Arbitrary Address Write I: modprobe

> The Linux Kernel already has a opt-in defense against it:

`CONFIG_STATIC_USERMODEHELPER=y`

Makes the `usermodehelper` path variables static `/sbin/usermode-helper`:

```
1   #ifdef CONFIG_STATIC_USERMODEHELPER
2       sub_info->path = CONFIG_STATIC_USERMODEHELPER_PATH;
3   #else
4       sub_info->path = path;
5   #endif
```

`/sbin/usermode-helper` is run with `modprobe`, or others, as `argv[0]`

The `usermode-helper` can inspect and filter malicious requests

Or, you can also disable `usermodehelper` entirely

Arbitrary Address Write II (physmap)

> Assume we have an Arbitrary Address Write (AAW), again

This time, we cannot use the `usermodehelper` trick

physmap / linear map / linear mapping

The kernel keeps a linear virtual mapping of all installed RAM

If you know the address, you can basically write directly to system memory

An attacker could direct writes to the `physmap`, specially towards kernel `.data`

On Android its even worse, as nowadays `physmap` location is not randomized:

https://projectzero.google/2025/11/defeating-kaslr-by-doing-nothing-at-all.html

Arbitrary Address Write II (physmap)


```
...
ffff8000000 ... ffff87fffffffff |    8 TB | ... guard hole, also reserved for hypervisor
ffff8800000 ... ffff887fffffffff |  0.5 TB | LDT remap for PTI
ffff8880000 ... ffffc87fffffffff |   64 TB | direct mapping of all physical memory ...
ffffc880000 ... ffffc8fffffffff |  0.5 TB | ... unused hole
ffffc900000 ... ffffe8fffffffff |   32 TB | vmalloc/ioremap space (vmalloc_base)
ffffe900000 ... ffffe9fffffffff |    1 TB | ... unused hole
ffffea00000 ... ffffeafffffffff |    1 TB | virtual memory map (vmemmap_base)
ffffeb00000 ... ffffebfffffffff |    1 TB | ... unused hole
ffffec00000 ... fffffbfffffffff |   16 TB | KASAN shadow memory
...
```

CTF Bonus: https://ecomaikgolf.com/posts/0017-glacierctf2025---flipfliphooray/

One bit flip to root without KASLR leak via `physmap`

Out of Bounds Write

> Assume you have an out of bounds write past a `kmalloc-XYZ` object

Problem: you don't know what's past your object

Slab Page (simplified)

A page tracked by the slab allocator that contains objects

Those slots get served/freed via a freelist within the page

Without `CONFIG_SLAB_FREELIST_RANDOM`, objects on a fresh slab are predictable

The freelist just points to consecutive memory locations, filling it in order

Thus, on a fresh slab page, we can just allocate `kmalloc-XYZ` and a victim object

To obtain a freesh slab page, we just force lots of allocations

Out of Bounds Write

> Originally the freelist pointed one after eachother, nowadays its randomized

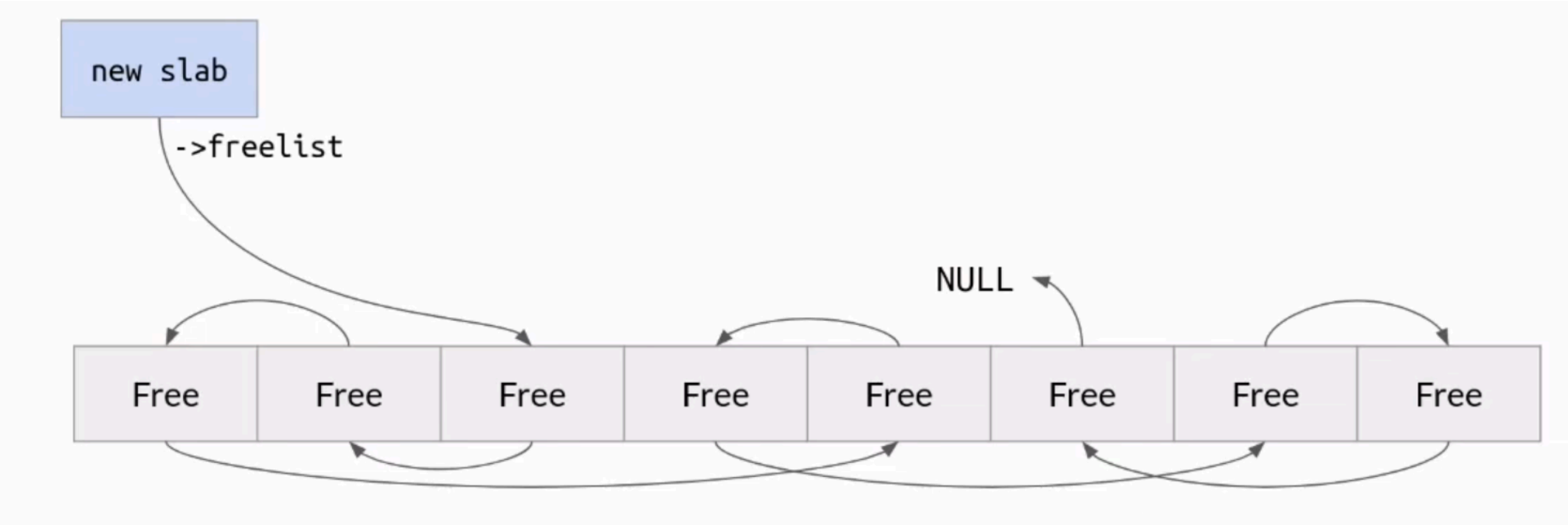
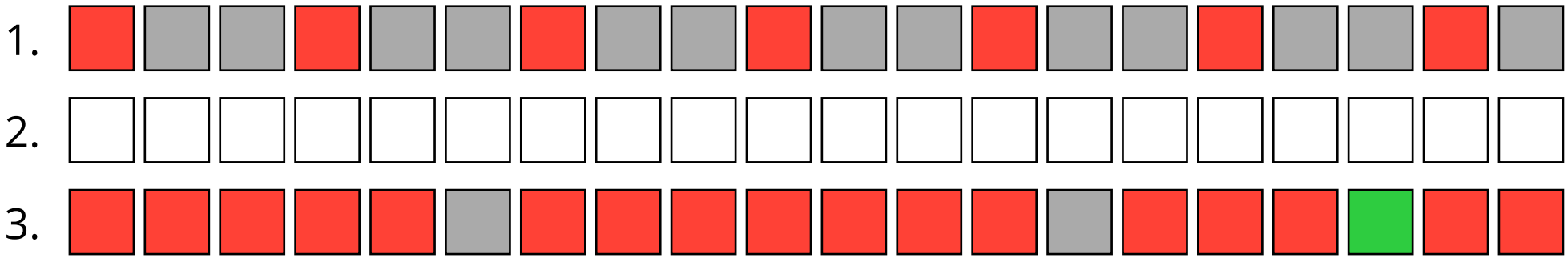


Image Source: SLUB Internals for Exploit Developers (xairy.io)

Out of Bounds Write

> As `CONFIG_SLAB_FREELIST_RANDOM` is heavily used, how could we bypass it?

1. Allocate many objects so we fill the current slab page
2. Once a new slab page is allocated, allocate our vulnerable OOB object
3. Allocate many victim objects which we want to overwrite
4. Cause the OOB



Worst case scenario, the vulnerable OOB object gets allocated at the end

Out of Bounds Write

> Does this look reliable to you?

First, how do we know a new slab page was allocated?

A slower allocation likely means it required allocating a new page

Second, what if we couldn't fill the entire page with victim objects? ...

Explotation Success Rate

On kernel exploitation there are a lot of unreliable exploitation steps

Two things are heavily valued in kernel exploitation:

- Success rate of the exploit overall
- Consequence of an unsuccessful run (e.g. crashes the machine?)

Use After Free Write

> Assume you have an UAF write on a `kmalloc-XYZ` object

An easy path forward would be overlapping a `struct cred` from an attacker task

The dangling pointer from `kmalloc-XYZ` would now point to a `struct cred`

We can allocate `struct cred` with unprivileged `setuid(getuid())`

The attacker can then write from the dangling reference to overwrite the `uid`

First Problem: Object Size

We have a vulnerable object in `kmalloc-XYZ` with XYZ being the size

`struct cred` must allocate from the same size cache

Otherwise, find alternative object

Use After Free Write

> Now let's assume modern allocator hardening with caches

Our attacker object is `kmalloc-XYZ`, victim is `struct cred`

Second Problem: Heap Segregation

`kmalloc-XYZ` and `struct cred` are completely separated

They do not share memory, they do not share freelists

Dangling pointers in `kmalloc-XYZ` only affect `kmalloc-XYZ` objects

We deallocate tons of `kmalloc-XYZ` so the pages backing the objects are freed

We allocate tons of `struct cred` so requests new pages

And hopefully the page containing the UAF reference comes back :)

Breaking KASLR

> Assume you can hijack the execution, but you miss the KASLR leak

You can use projects such as https://github.com/bcoles/kasld

kasld

A database of tricks and side channels to leak KASLR

All packed in a single big-red-button tool

Just run it and obtain:

```
1 Kernel text (virtual)     0xffffffffa7a00000  (1 source)
2 ...
```

Side-Channel Security explains one of the tricks on the practicals

Breaking KASLR

> What if you **need** to bypass KASLR but you are lazy?

```
[ecomaikgolf@laptop ~/]$ id
uid=1000(ecomaikgolf) gid=1000(ecomaikgolf) groups=1000(ecomaikgolf)
[ecomaikgolf@laptop ~/]$ xxd /sys/kernel/notes
00000000: 0600 0000 0400 0000 0101 0000 4c69 6e75  ............Linu
00000010: 7800 0000 0000 0000 0600 0000 1600 0000  x...............
00000020: 0001 0000 4c69 6e75 7800 0000 362e 382e  ....Linux...6.8.
00000030: 342d 3230 302e 6663 3339 2e78 3836 5f36  4-200.fc39.x86_6
00000040: 3400 0000 0400 0000 0800 0000 1200 0000  4...............
00000050: 5865 6e00 0000 001e 0000 0000 0400 0000  Xen.............
00000060: 0600 0000 0600 0000 5865 6e00 6c69 6e75  ........Xen.linu
00000070: 7800 0000 0400 0000 0400 0000 0700 0000  x...............
00000080: 5865 6e00 322e 3600 0400 0000 0800 0000  Xen.2.6.........
00000090: 0500 0000 5865 6e00 7865 6e2d 332e 3000  ....Xen.xen-3.0.
000000a0: 0400 0000 0800 0000 0300 0000 5865 6e00  ............Xen.
000000b0: 0000 0080 ffff ffff 0400 0000 0800 0000  ................
000000c0: 0f00 0000 5865 6e00 0000 0000 8000 0000  ....Xen.........
000000d0: 0400 0000 0800 0000 0100 0000 5865 6e00  ............Xen.
000000e0: 7038 d0a0 ffff ffff 0400 0000 1500 0000  p8..............
000000f0: 0a00 0000 5865 6e00 2177 7269 7461 626c  ....Xen.!writabl
00000100: 655f 7061 6765 5f74 6162 6c65 7300 0000  e_page_tables...
00000110: 0400 0000 0400 0000 0900 0000 5865 6e00  ............Xen.
00000120: 7965 7300 0400 0000 1000 0000 0d00 0000  yes.............
00000130: 5865 6e00 0100 0000 0000 0000 0100 0000  Xen.............
00000140: 0000 0000 0400 0000 0400 0000 1000 0000  ................
00000150: 5865 6e00 0100 0000 0400 0000 0800 0000  Xen.............
00000160: 0400 0000 5865 6e00 0000 0000 0000 0000  ....Xen.........
00000170: 0400 0000 0800 0000 0200 0000 5865 6e00  ............Xen.
00000180: 0000 029f ffff ffff 0400 0000 0400 0000  ................
00000190: 1100 0000 5865 6e00 0188 0000 0400 0000  ....Xen.........
000001a0: 0800 0000 0800 0000 5865 6e00 6765 6e65  ........Xen.gene
000001b0: 7269 6300 0400 0000 0400 0000 0e00 0000  ric.............
000001c0: 5865 6e00 0100 0000 0400 0000 1400 0000  Xen.............
000001d0: 0300 0000 474e 5500 922c 0f91 6603 327e  ....GNU..,..f.2~
000001e0: eb20 d2c8 f2bb b852 8e9b 6974            . .....R..it
[ecomaikgolf@laptop ~/]$ sudo cat /proc/kallsyms | grep startup_xen
ffffffffa0d03870 T startup_xen
[ecomaikgolf@laptop ~/]$ uname -r
6.8.4-200.fc39.x86_64
```

Don’t worry, the Xen straight leaked KASLR for 11 years on `/sys/kernel/notes`

Defenses

KASLR and FG-KASLR

> KASLR randomizes the location of the kernel base address, similar to ASLR

An attacker that wants to jump to `commit_creds()` must first leak the address

Leaking the address of a kernel function `foo()` also leaks `commit_creds()`

As the distance (offset) of `foo()` and `commit_creds()` is constant

Function Granular KASLR (FG-KASLR)

A fine-grained KASLR that rearranges kernel code at per-function granularity

Shuffles the place of the functions within the kernel

Now, leaking `foo()` does not give a predictable offset to `commit_creds()`

https://lwn.net/Articles/824307/

RANDSTRUCT

> Randomizes members of C structs within the kernel at build time

Sensitive structures can get a randomized layout with `__randomize_layout;`

Example: `cred->uid` is always at offset `0xABC`, now its at `0xXYZ`

Randomness comes from a build-time generated seed

Problem: peple do not build their own kernel

1. Attacker can download the same Ubuntu kernel image as the victim
2. Then, decompile the offsets of the relevant structures

Furthermode, all built modules would also need the seed

Benefit: if seed not known, the attacker may crash the kernel

SMAP and SMEP

> Security mechanism preventing userspace attacker data placement

SMAP: Supervisor Mode Access Prevention

SMEP: Supervisor Mode Execution Prevention

When running kernel code, SMAP and SMEP activate

This prevents the kernel from executing userspace addresses or accessing data

Usecase

Imagine an attacker with a single jump

An attacker could write shellcode on userspace buffers

And cause a jump from the kernel to the userspace shellcode memory

Slab Hardening

> The kernel slab allocator (SLUB) received all kinds of defenses, including:

Heap Segregation

Objects are divided into caches, which mantain their owm memory

Having an UAF on cache XYZ means you cannot reuse it with `struct cred`

This led to the new world of **cross-cache attacks**

`CONFIG_RANDOM_KMALLOC_CACHES`

Each generic `kmalloc` allocation picks a random `kmalloc-XYZ-N` cache instance

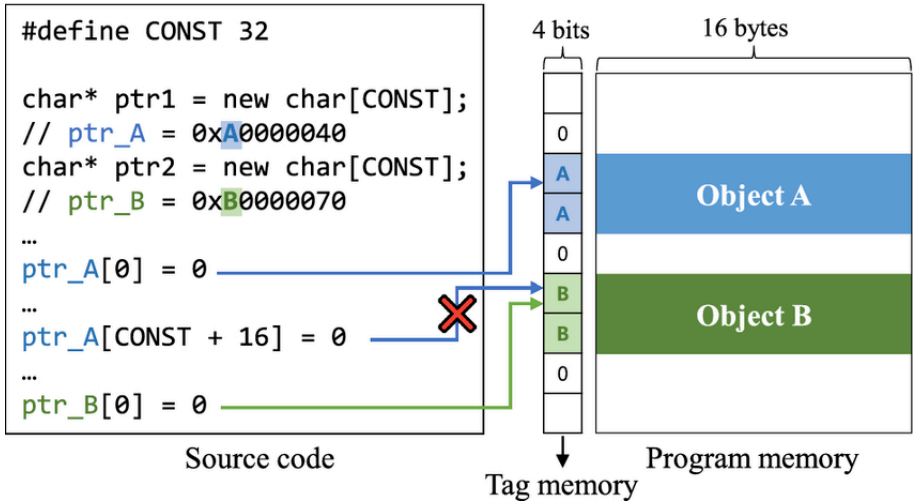
https://sam4k.com/exploring-linux-random-kmalloc-caches/

Hardware-Tags KASAN

> Memory Tagging is implemented as a production-ready KASAN mode

KASAN is the Linux kernel address sanitizer, usually for debugging

Novel `HW_TAGS_KASAN` mode uses ARM Memory Tagging Extension:



Intended for production usage: Android (Pixel 8 or newer) via `bootctl`

Security Enhanced Linux (SELinux)

> SELinux is a security module that provides access control security policies

When a process (**subject**) requests a file (**object**), SELinux is consulted

Much more complete and complicated than normal permissions:

```
1 $ ls -lZ
2 drwxr-xr-x. 1 ecomaikgolf unconfined_u:object_r:user_home_t:s0 archives
```

Policies are shipped on distribution packages, next to the binaries

Android & RedHat are heavy users of SELinux

You may have an unprivileged exploit for `/dev/XYZ`...

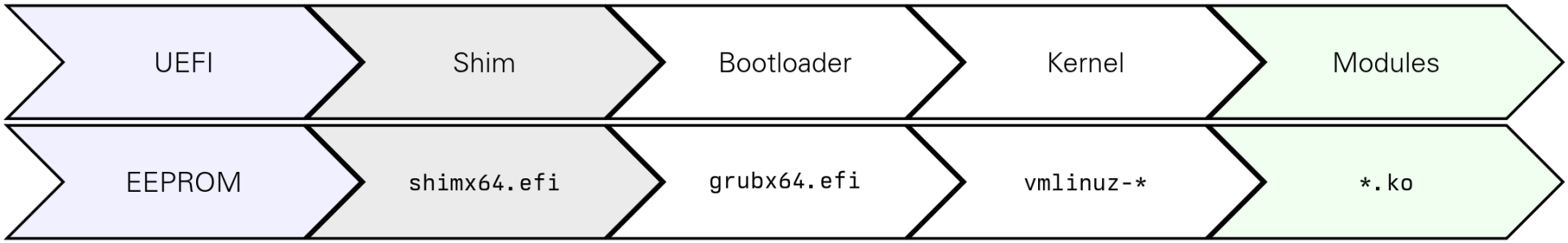
But Android SELinux policies forbid `u:r:untrusted_app_*` subject to open

Other apps may be different: `u:r:system_app:s0` , `u:r:priv_app:s0` ...

Secure Boot & Lockdown Mode

> Secure Boot (usually) instructs Linux to never run unsigned code

All code, from UEFI to bootloader to kernel to modules must be signed:



GRUB is probably not the first thing that runs on your system, its the `shim`

Secures against physical attacks and even root users (on certain scenarios).

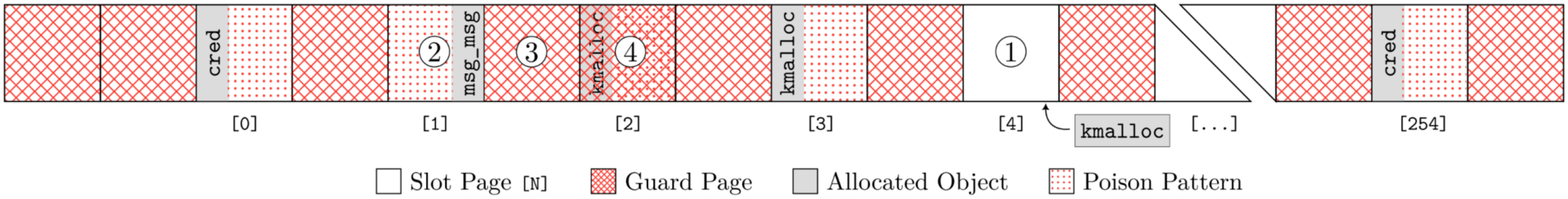
- https://ls.ecomaikgolf.com/slides/secureboot/slides-handout.pdf
- https://tube.tugraz.at/portal/watch/9fbfce8e-22cb-439b-b5ed-ecd6cd8a50f0

Kernel Electric Fence (KFENCE)

> A production-ready sample-based runtime memory sanitizer

Every N milliseconds, the next kernel allocation is served by KFENCE and not Slab.

The object is allocated within guard (unmapped) pages, faulting on access:



Detects memory corruption bugs on little served (sampled) objects

But, all Android phones have it enabled by default

Will billions of devices, some devices catch bugs, with backtraces sent to Google

Samsung Runtime Kernel Protection (RKP)

> Samsung, as part of Knox, runs the Linux Kernel under a Hypervisor (RKP)

Makes escalation of privileges harder with hypervisor-aided protections. E.g.:

Global Variable Protection

Global variables with `__rkp_ro` are placed on hypervisor-protected `ro` pages

Hardened Credential Change

Change of `struct cred` for a process goes through hypervisor checks

And much more:

https://blog.impalabs.com/2101_samsung-rkp-compendium.html

CVE-2026-3143*1

(aka Copy Fail)

CVE-2026-31431

> LPE vulnerability affecting every major Linux distribution since 2017

The vulnerability is a **logic issue** on the combination of:

- `authencesn`
- `AF_ALG`
- `splice`

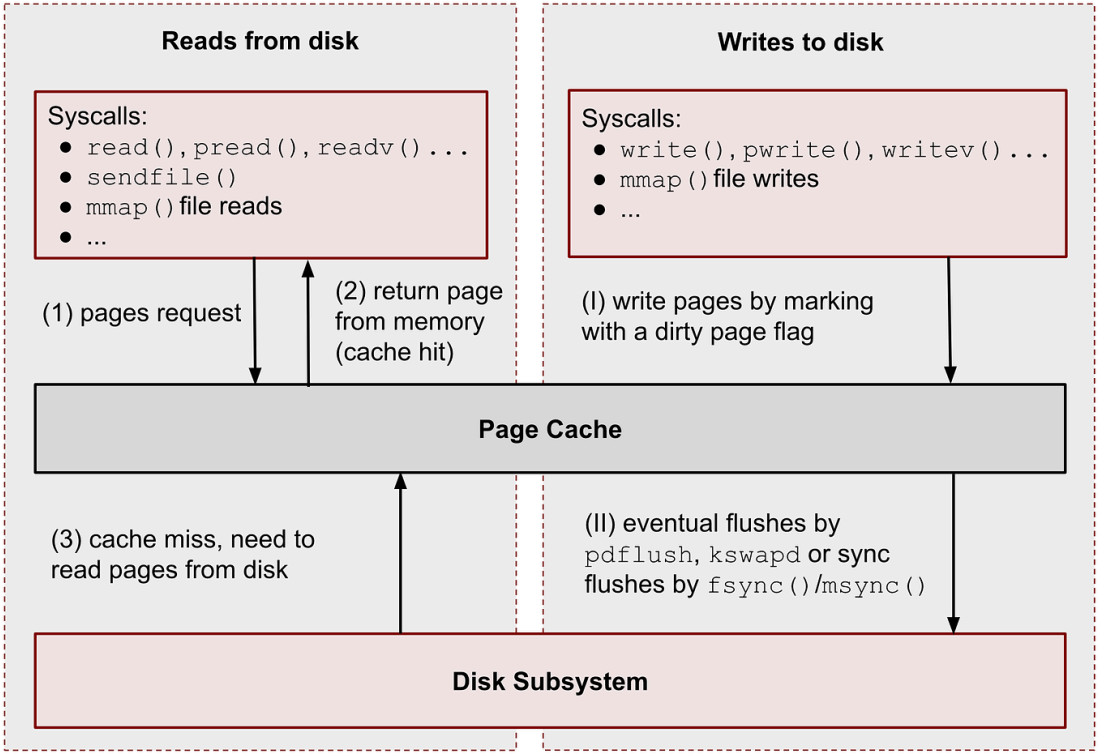
At the userspace-accessible IPSec+ESN (Extended Sequence Numbers) code

Primitive
Deterministic 4-byte write into the page cache of any readable file

Allows to write shellcode into a **setuid** binary, giving direct LPE to root.

Concept: Page Cache

RAM-backed cache for file read/write operations, saving disk accesses



Source: lamdafunc

Concept: Scatterlist

> An continuous buffer abstraction for discontinuous buffers

```
1 struct scatterlist {
2   unsigned long page_link; /* struct page */
3   unsigned int offset;     /* byte offset within page */
4   unsigned int length;     /* len */
5   ...
6 };
```

Effectively, you have different buffers within different pages:

```
1 page A (offset=512, len=XYZ) -> page B (offset=0, len=N) -> ... page C (offset= ...
```

Over which you can operate as continuous buffers:

```
1 scatterwalk_map_and_copy(dst, scatter, OFFSET, LEN, ...);
```

Concept: Splice

> A (much faster) technique to move data between file descriptors

I have an enormous compressed file that I need to send to decompress

You can `read()` a file into a buffer, then `write()` or send somewhere...

But this has to load into a buffer, go through userspace, etc.

Solution: splice

The kernel can just pass a reference of the page cache

Known as Zero-Copy IO

"Here's the page cache of file XYZ, read it, decompress it, and write it at `dst[]`"

No need for a copy/load into `src[]` :)

Background: AF_ALG

> Kernel's userpace crypto API, works similar to sockets

```c
int alg_fd = socket(AF_ALG, SOCK_SEQPACKET, 0);
struct sockaddr_alg sa = {
  .salg_family = AF_ALG,
  .salg_type   = "aead",
  .salg_name   = "authencesn(hmac(sha256),cbc(aes))"
};
bind(alg_fd, (struct sockaddr *)&sa, sizeof(sa));
```

Usage sample from: https://retr0.zip/blog/cve-2026-31431-copy-fail.html

An easy an extendable way of advertising hardware/kernel-accelerated crypto

You query + request an algorithm, and send + receive data from a socket

The kernel takes care of the rest!

Background: IPsec + Extended Sequence Numbers

> Protocol to authenticate and encrypt network packets started in 1992

Nothing special to explain, it encrypts packets as promised :)

Packets carry an authenticated sequence number to avoid **replay attacks**

Integer Overflow Problem

Sequence number is **32 bits** and increases on every packet transmission

With modern transfer speeds, the sequence number overflows in seconds

So, the IETF designed the **Extended Sequence Numbers (ESN)**

Both sides of the connection mantain a separate **authenticated** counter

Not transmitted, both implicitly synchronize on it lower-half sequence overflows

Packet Flow: Wire to IPSec/ESP

> AF_ALG has the `auth-enc-esn` special mode to deal with the **ESN quirk**

The following IPSec-related packet is received via socket for decryption:

SPI	seqno_lo	C	Tag

IPSec integrates the implicit upper-half counter into the packet:

SPI	seqno_hi	seqno_lo	C	Tag

Via https://elixir.bootlin.com/linux/v6.12.10/source/net/ipv4/esp4.c#L856

Which then is passed to **`authencesn`**

Packet Flow: IPSec/ESP to authencesn

> Authencesn now re-structures the packet (Standard/RFC)

The received data is the folowing:

SPI	seqno_hi	seqno_lo	C	Tag

But `authencesn` must re-organize again the packet, moving `seqno_hi` to the end

authencesn does a little trick, uses the previous Tag as scratchpad for `seqno_hi`:

```
1 tmp[0..1] = dst[0..7] // save SPI and seqno_hi
2 dst[4..7] = tmp[0] //  write SPI on seqno_hi
3 dst[assoclen+cryptlen] = tmp[1] // write seqno_ho
```

SPI	seqno_lo	C	seqno_hi

Bug: In-Place Decryption

> Commit `72548b093ee3` (2017) to use the same “buffer” for src+dst in decryption

`AF_ALG` worked with two scatterlists: `src` and `dst`:

- TX SGL (transmit scatterlist): can be from the page cache (readonly)
- RX SGL (receive scatterlist): allocated buffer on `recvmsg()`

```
1 TX SGL = AAD + C + Tag
2 RX SGL = AAD + M
```

The idea was to operate on a single scatterlist:

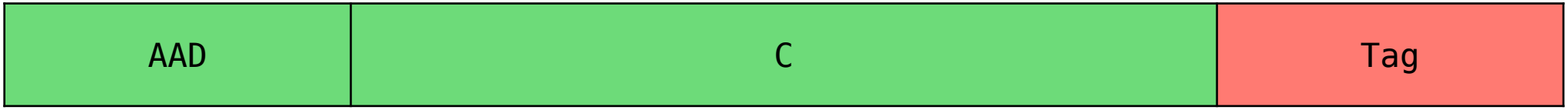
1. Copy AAD and C from TX to RX directly
2. Append Tag from TX to RX’s scatterlist

And now `src/dst` is a single scatterlist and you don’t need two

Bug: In-Place Decryption + Scatterlist

> The memory layout of the decryption “buffer” is… complicated

This is the `authencesn` linear view of the memory:



But, remember this is not a linear buffer, but a scatterlist.

AAD and C are fine, reside on RX’s allocated buffer

Tag comes directly from TX, appended on the scatterlist

Meanwhile AAD and C are on an allocated buffer, Tag’s situation is… unclear

Bug: In-Place Decryption + Scatterlist + Page Cache

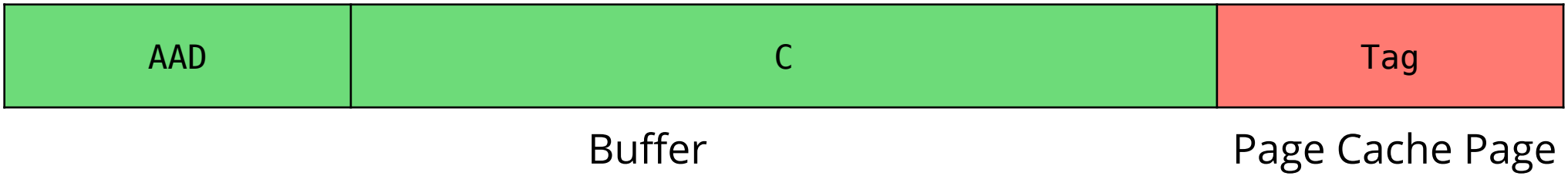
> Remember we said splice could pass a reference of the page cache?

An attacker can use `pipe` and `splice` to pass a “file” ciphertext to `authencesn`

This zero-copy move is done by sharing the page cache page from the file

`authencesn` receives a page from the page cache and puts it into TX scatterlist

As the in-place decryption merge happens, the final scatterlist is a mix:



Any write afer C is fatal, would corrupt in-memory file contents

Anyway, who writes to Tag during a decryption? For years, seems that no one

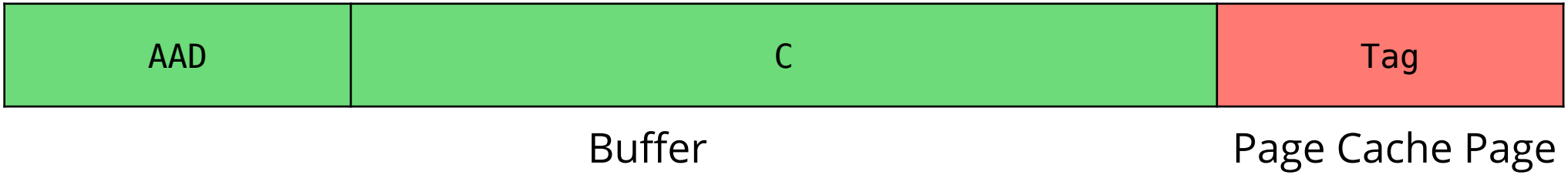
Exploitation: Page Cache Corruption Primitive

> `authencesn` is the only alg/mode that writes to Tag during a decryption

This is due to the Extended Sequence Numbers quirk.

`authencesn` takes `seqno_hi` and writes it into the Tag

As Tag comes from TX SGL, the scatterlist can be backed by the page cache:



`seqno_hi` is provided on the decryption request as part of the payload

Furthermore, the write to the Tag is done before checking[1] it

[1] As it gets previously, assumes it can be used as scratchpad

Exploitation: Page Cache of `/usr/bin/su`

> Corrupting the page cache is fatal and exploitation is straightforward

Whenever a user invokes a program, it gets read from the page cache (on hit)

If we change the page cache contents, executed program is different

Even if the program is a `setuid` binary or similar

Exploitation

1. Open `/usr/bin/su` (allowed!)
2. Send its page cache pages to `authencesn` via `splice`
3. Arbitrary byte write due to tag's overwrite, repeat for multiple bytes
4. Write `setuid(0); execve("/bin/sh")` shellcode
5. Run `/usr/bin/su`, reads contents from page cache, runs our code

Conclusion

Conclusion

> Linux Kernel Exploitation is a hard and dense topic

Usually its the last line of defense for a complete compromise

Lots of differences with userspace exploitation:

- Sources of uncertainty (e.g. shared allocator)
- Bad luck exploiting can crash the machine and signal the victim

Large codebase, lots of interaction, monolithic, external drivers:

- Perfect recipe for exploitation

Shares resources across multiple users:

- Perfect recipe for side channels

TU Graz

SCIENCE
PASSION
TECHNOLOGY

Introduction to Linux Kernel Exploitation

Featuring CVE-2026-31431

Ernesto Martínez García

Pentesting Lab SS26