SIGPwny

SP2025 Week 14 • 2025-05-01

# Kernel Security

Nikhil Date and Akhil Bharanidhar

# Announcements

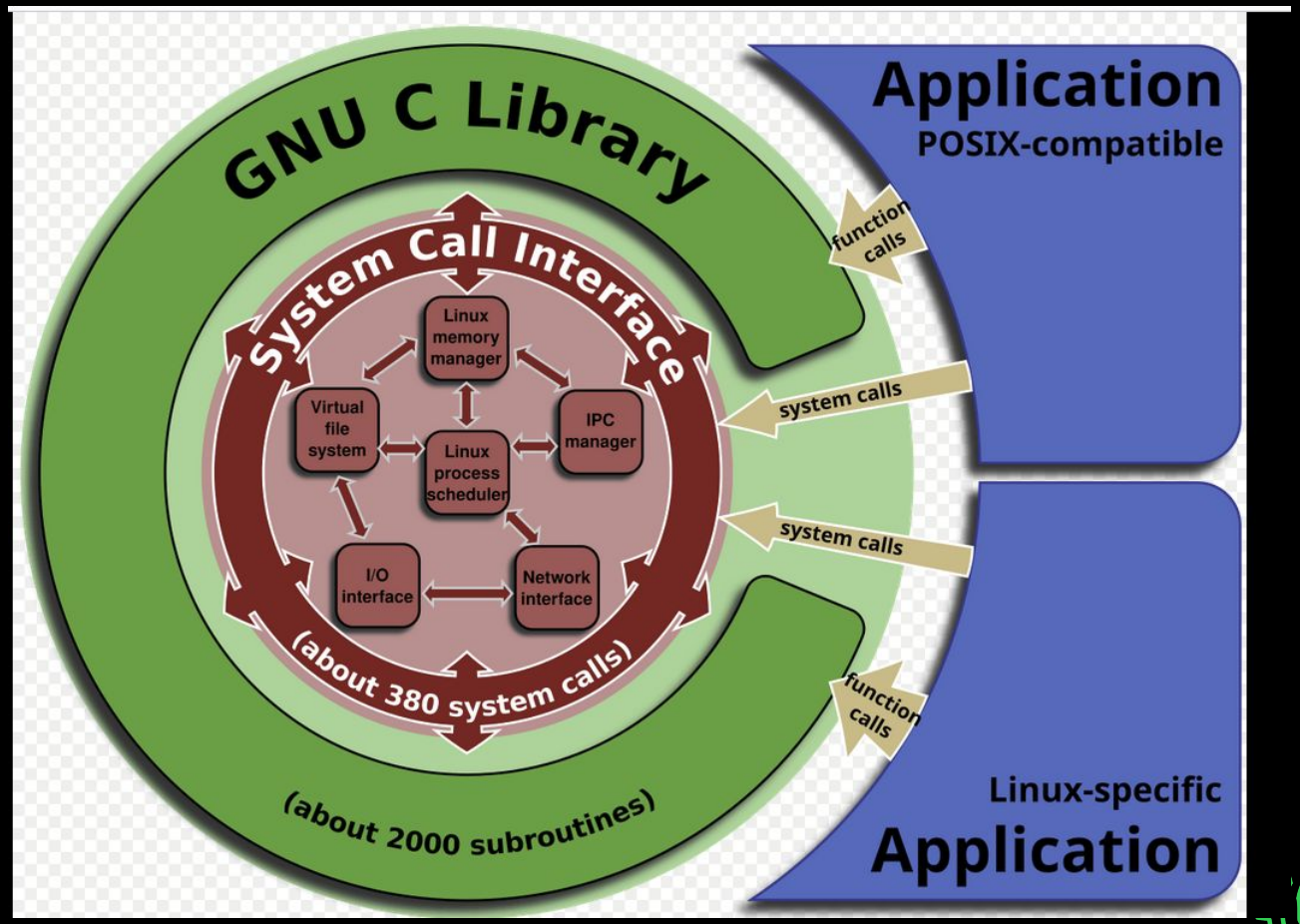
- End of year Social on Wednesday 5/7!
  - Come to our end of year social to celebrate our graduating members and a year's worth of hard work!

ctf.sigpwny.com

# sigpwny{the_docs_are_the_source_code}

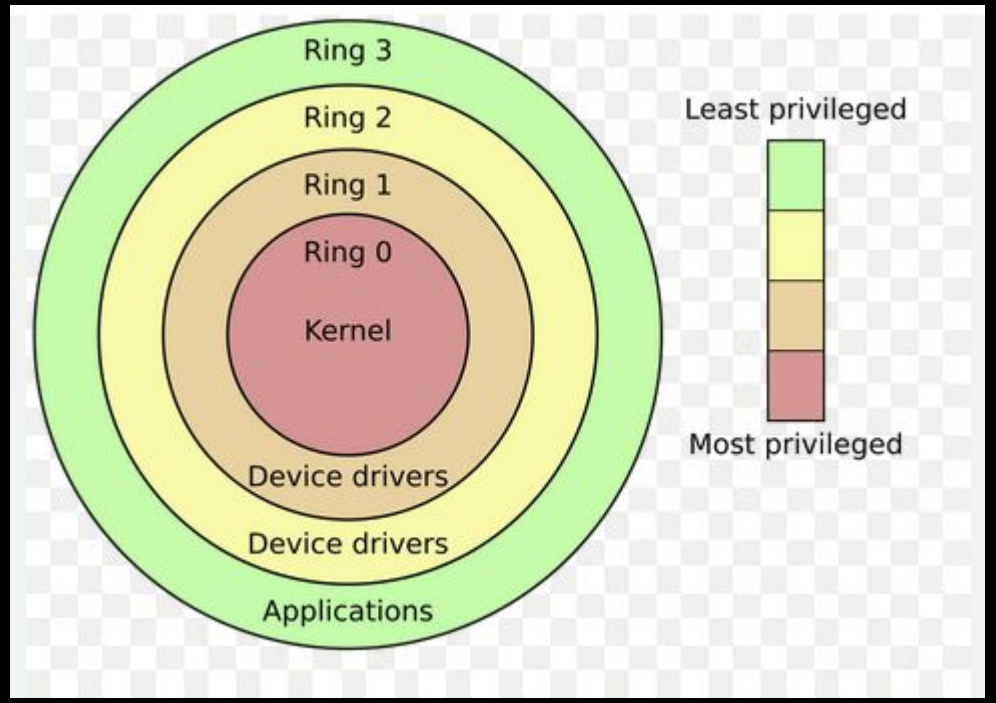
# The Kernel

- Provides (along with drivers) an interface to hardware resources
- Enforces isolation and security mechanisms
- Manages all “low-level” tasks
- Provides process scheduling and concurrency
- And more



# Why care about kernel security?

- Kernel runs in most privileged mode, has unrestricted access to hardware
- Most security mechanisms (authentication/access control, process isolation, etc.) rely on the integrity of the kernel
- Once you compromise the kernel, you can do pretty much anything you want on the system




# Standard Kernel Pwn Setup

- “Secure” kernel image with vulnerable kernel module
- Could also have vulnerable kernel image (typically with a diff provided against a real Linux version)
- Also provided root filesystem (doesn’t really have anything to do with root user)
- Access to non-privileged shell on machine
- Can run arbitrary code as user
- Goal is to become root

# Standard Kernel Pwn Setup

- Run kernel image using QEMU
- Attach GDB to debug exploit

# ret2usr

- When you ask the kernel to do something (e.g. through a system call), the CPU switches into a more privileged mode
- Kernel code then runs with privileged access
- Once kernel is done, CPU switches back and hands control to user program

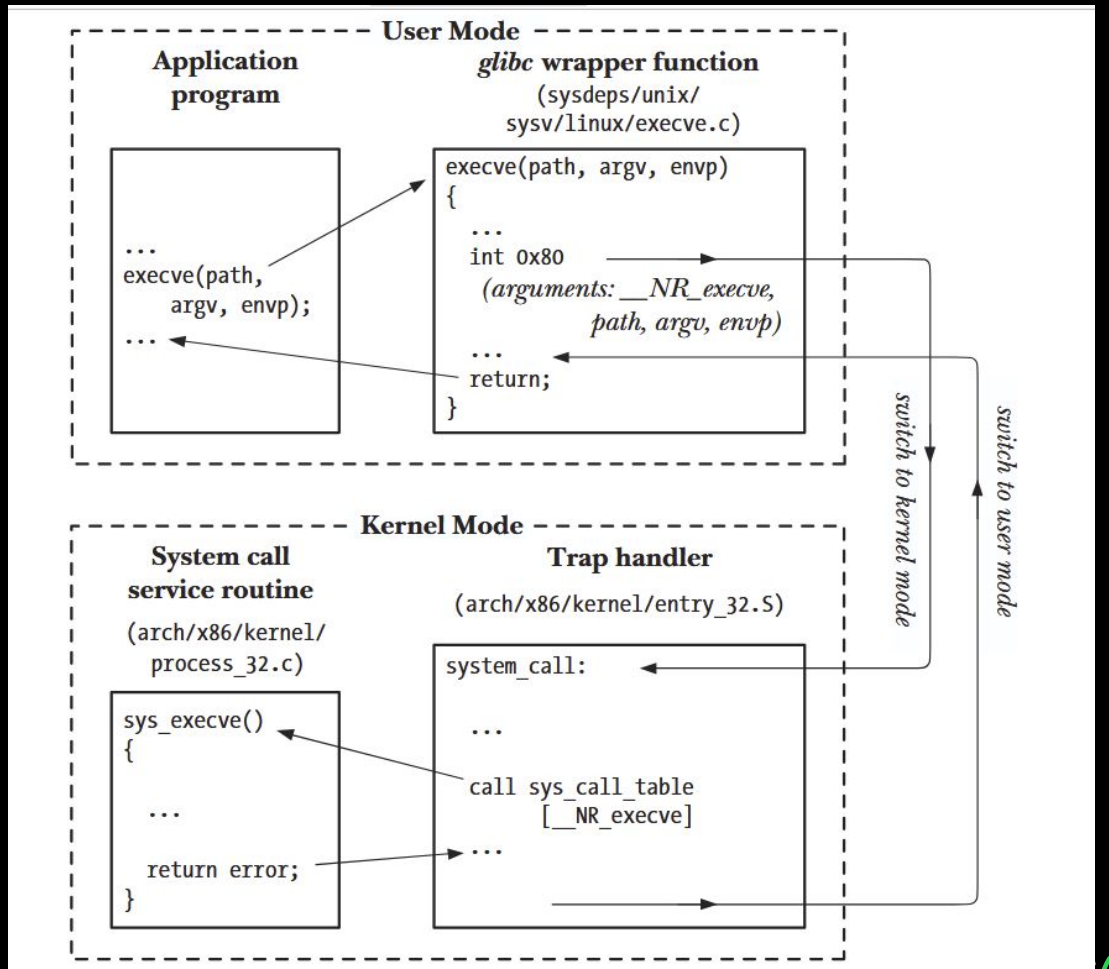


Figure 3-1: Steps in the execution of a system call

# ret2usr

- Suppose we could trigger the following code in the kernel
- `copy_from_user` copies data from user-space into kernel-space

```
void vulnerable_kernel_func(unsigned long addr) {
    char buffer[256];
    volatile int size = 512;
    return copy_from_user(&buffer, addr, size);
}
```

# ret2usr

- Buffer overflow gives us arbitrary code execution
- Ideally, we want to execute shellcode
- Place shellcode in userspace and jump to user code!
- No context-switch happens, so user code will run with kernel privilege

# ret2usr

- Once we have a kernel exploit, it's a good idea to use it to become root, since then we don't have to do the rest of the attack from "inside" the exploit
- How to become root in two function calls
  - `prepare_kernel_cred(0)` returns credentials for root user
  - `commit_creds` sets current user process creds
- If we call `commit_creds(prepare_kernel_cred(0))` in the user code we jump to, we are root
- We can then switch back to user mode and pop a shell (which is now a root shell)
- At this point we have full control of the system
- Actual details are a bit intricate, follow blog post linked at the end of slides

# Kernel Defenses

- Stack canaries
- DEP/W ^ X
- kASLR
  - randomizes kernel base address, but small amount of entropy due to alignment constraints (9 bits on Linux)
- SMEP (supervisor mode execution prevention)
  - There is no good reason why user code should be able to execute with kernel privileges
  - Use user/kernel bits in page entries to prevent code in user pages from being executed while CPU is in kernel mode
- SMAP (supervisor mode access prevention)
  - Stronger than SMEP: disallow all accesses to user memory in kernel-mode (with some exceptions when this is actually needed)
- Seccomp restricts user access to syscalls

# More Attacks

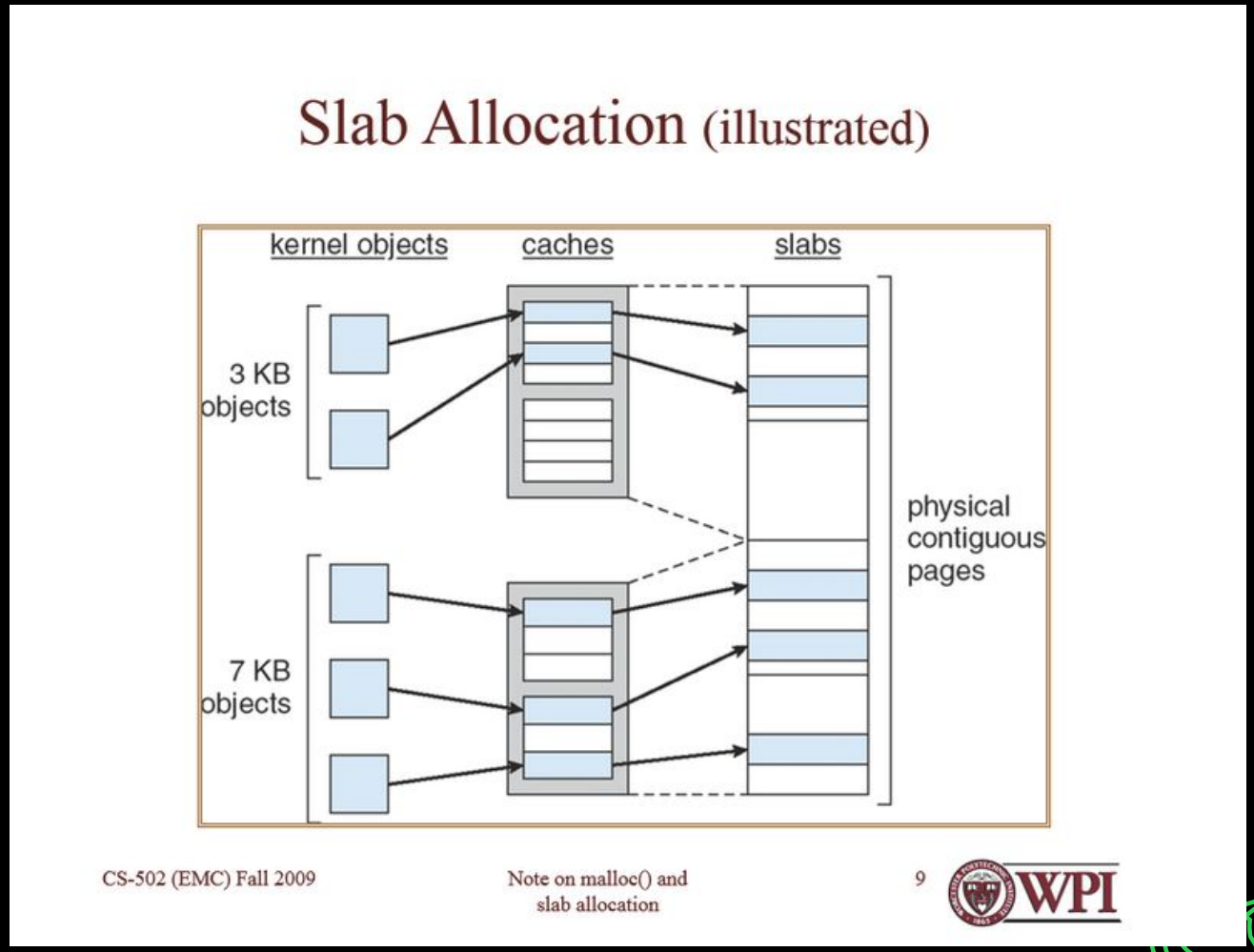
- Kernel ROP: useful if SMAP/SMEP is enabled, same idea as user ROP
- Data-only attacks: overwrite sensitive data
  - task structure contains uid/gid, overwrite to 0 to become root
- OOB reads to break kASLR
- Many attacks use similar primitives as user-space pwn

# Kernel Memory Allocation

- Two-levels of memory allocation: low-level page allocation and higher-level chunk allocation
- Page allocator (get_free_pages) allocates in units of pages using buddy allocator
- Higher-level allocator (kmalloc) is a slab allocator: SLAB, SLUB (default), SLOB, etc.
- Slab allocator maintains caches of chunks for specific kernel objects, along with generic chunks

# SLUB Overview

- Three levels: caches, slabs, and chunks
- Each cache is allocated with a kernel object (or is a generic cache) and contains slabs
- A slab consists of contiguous pages and holds chunks
- Chunks are the actual objects



# SLUB Overview

- SLUB free objects store pointer to next free object
- Allocated objects store no metadata (makes it easy to overwrite sensitive data without corrupting allocator state)
- We can also try to corrupt metadata, though there are some security checks (like checking that next free chunk actually resides in the current slab)

# Attacking SLUB

- As an attacker, we have the ability to allocate kernel objects by making system calls
- We can try to ensure that arrangement of objects is to our advantage (for e.g. sensitive object comes after vulnerable object)
- Heap sprays (allocating large number of objects) can help increase chance of success

# Resources

- https://lkmidas.github.io/posts/20210123-linux-kernel-pwn-part-1/ (basic attacks)
- https://duasynt.com/blog/linux-kernel-heap-feng-shui-2022
- pwncollege Kernel Exploitation Module
- https://github.com/xairy/linux-kernel-exploitation
- Great talk on SLUB: https://www.youtube.com/watch?v=2hYzxsWeNcE
- https://elixir.bootlin.com/linux/v6.14.4/source

# Challenge

- Try to reproduce the exploit here (there is a full solution given but try to do it without looking at the solution)
- https://lkmidas.github.io/posts/20210123-linux-kernel-pwn-part-1/

# Next Meetings

**2025-05-04 • This Sunday**

- Game Hacking
- Learn how to hack games!

**2025-05-07 • This Wednesday**

- End of Year Social
- Our last get together of the year, goodbye to our graduating members, prizes for pwnyctf, and more!

ctf.sigpwny.com

`sigpwny{the_docs_are_the_source_code}`

**Meeting content can be found at sigpwny.com/meetings.**

SIGPwny