Breaking the Glass Sandbox: Find Kernel Bugs and Escape

Valentina Palmiotti

RECon Montreal 2022

ABOUT ME

Lead Security Researcher at Grapl, a next generation SIEM

Strange Beginnings

Background in economic research prior to switching to security

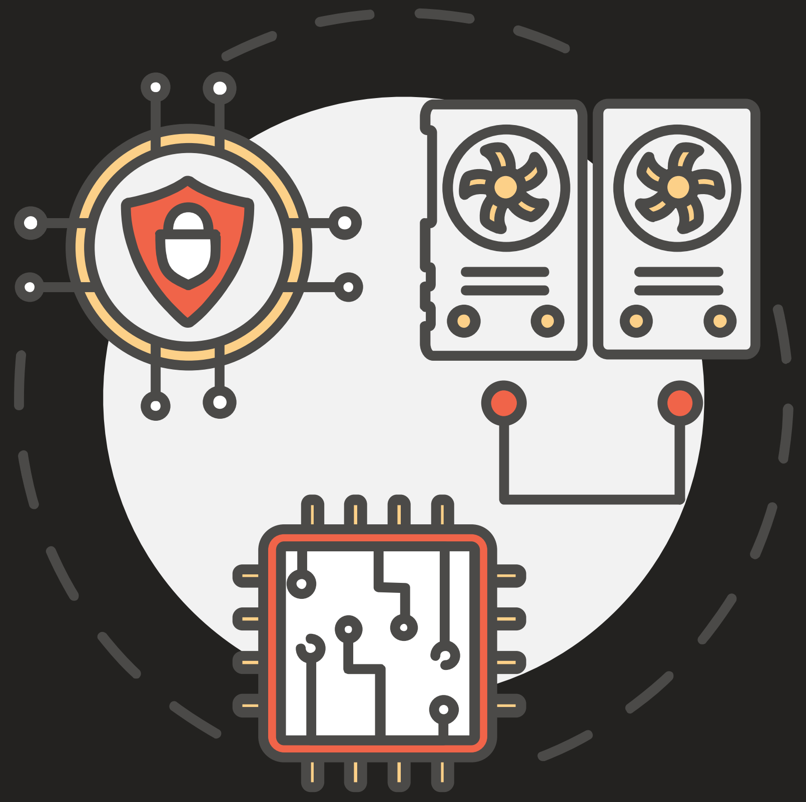
Offensive Minded

Focus on exploit development, techniques, and vulnerabilities at the OS level.
Interested in anything and everything offensive security

OS Internals

Linux (kernel), Windows, Android

Inspiration for this Talk



A

Android Rooting Community

Why can't we use a generic kernel bug to get root?

OR

I have a root shell, but it's useless!

B

WTF Does a Sandbox Do?

What are the actual restrictions that sandboxing methods impose with respect to the kernel? How much access/attack surface am I giving to random apps I install?

C

Better Bug Hunting

What kinds of bugs **CAN** we use?
How do I find them?

Exploit Development

Vulnerability Research

Require different but ***complementary*** skills

VS

Weaponization

Responsible for turning the theoretical impact of a bug into a real attack

Exploitability

Recognizing whether a bug is exploitable and how complicated it will be

Location, Location, Location

Identifying where in a code base a bug will be most useful

Code Auditing

Identifying vulnerabilities by meticulously reading code or reverse engineering disassembly

Tool Building

Creating tools that find bugs (ex: fuzzers i.e. corpus creation coverage, static analyzers etc)

What is a Sandbox?

Executing software in a restricted operating system environment, thus controlling the resources (e.g. file descriptors, memory, file system space, etc.) that a process may use

What about containers?

Is it a security boundary? ***Yes***, because containers provide restrictions on access to resources. Containers are built on sandboxing primitives.

Why Should I Care?

What to keep in mind before you start bug hunting

Security Impact

Kernel bugs that bypass one or more sandboxing boundary are most valuable because they work on the **most systems**

Shorter Chain

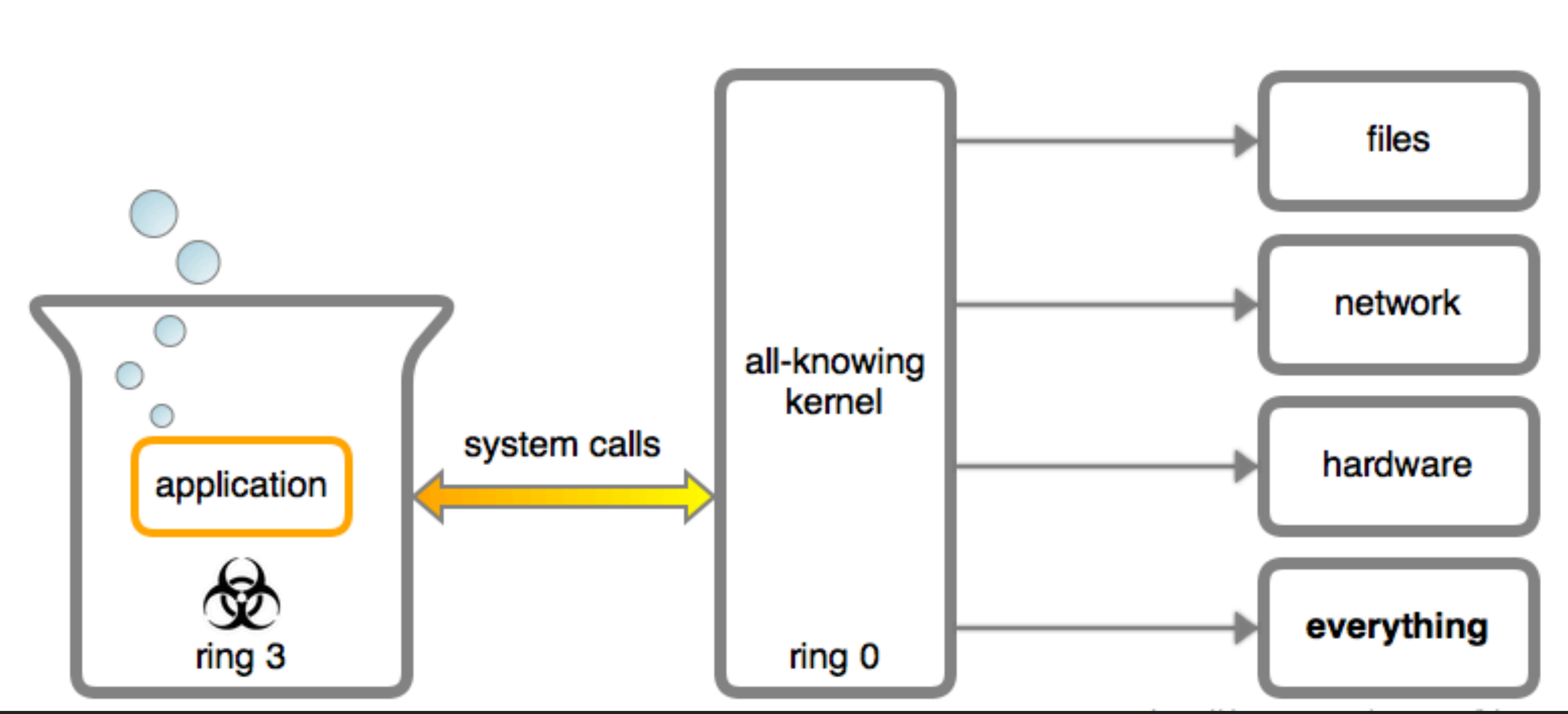
The more sandboxing primitives bypassed, the shorter the chain of bugs needed to finish privilege escalation.

Standardize Exploits

Often the type of bugs that are easier to exploit generically

How does a process interact with the kernel?

- Processes interact by making system calls into the operating system (i.e. the kernel).
- System calls are an interface to the services provided by the OS
- Kernel is responsible for enforcing security - is this application allowed to access the resource it's asking for?



How are these Boundaries Enforced?

And can we **break** them?

A

Still Need Kernel

Untrusted processes still need **some** access to the kernel

B

Lots of Bugs

The Linux kernel has lots of bugs to be found.

C

New systems + redesigns

New kernel subsystems and redesigns of kernel components introduce new attack surfaces reachable from sandbox

Picking a Target in the Kernel

Reducing Code Subset

Honing in on a particular subset of code you are considering

&

Finding More Reachable Code

Learning the internals of how the Linux kernel works and models things to push the limits of the sandbox and find more reachable code

Kernel Sandboxing Mechanisms

What are they?

Users/Groups - Traditional Unix system of users, groups, and RWX permissions (DAC)

Capabilities - Controls access to system-level privileges that are not covered by traditional file privileges.

Namespaces - Partitions global kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources

seccomp - System call filtering

Linux Security Modules (LSM) - Hooks in user level system calls where loaded security modules are called into and return an access decision. **(Ex: SELinux, App Armour)**

Sandboxing != Post/Exploitation Mitigation

These methods are intended to reduce reachable attack surface. They ARE NOT intended to provide any sort of protection if a reachable kernel vulnerability exists and/or has been exploited. Even if some vendors may try to use them that way.

Reachable kernel bug == WIN (game over)

seccomp

System call filtering

- **Seccomp provides a means filter accessible system calls from a process.**
- **Specifically Designed to Reduce Reachable Kernel Code - Limiting code as an attack surface**

- **Original: Strict Mode:**
  - O**nly allow the syscalls** exit(), sigreturn(), read() and write() to already-open file descriptors.
  - If any other syscall is made, the process is killed using SIGKILL
- **Seccomp-bpf**

  Filtering of system calls using a configurable policy using a classic BPF (not eBPF) program.

seccomp

System call filtering - problems

- Developers need to think about what system calls their applications make, not what resources it accesses
    - Can cause compatibility issues - ex: a libraries getting recompiled using new system calls, vDSO
    - Easier to make policy a deny list vs allow list — weakening attack surface reduction

- Restricts types of system calls can be called, but unable to do deep argument inspection
    - Filters can only look at top level system call arguments, pointers can't be dereferenced

Remember, everything on Linux is a file! File operations for different file types are handled by File operation functions defined in this structure.

File Operation structure for /proc/<pid>/mem

```
static const struct file_operations proc_mem_operations = {
    .llseek     = mem_lseek,
    .read       = mem_read,
    .write      = mem_write,
    .open       = mem_open,
    .release    = mem_release,
};
```

/proc/<pid>/mem: Read/Write Implementation in the Kernel

```c
static ssize_t mem_rw(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos, int write)
{
        struct mm_struct *mm = file->private_data;
        unsigned long addr = *ppos;
        ssize_t copied;
        char *page;
        unsigned int flags;

        if (!mm)
                return 0;

        page = (char *)__get_free_page(GFP_KERNEL);
        if (!page)
                return -ENOMEM;

        copied = 0;
        if (!mmget_not_zero(mm))
                goto free;

        flags = FOLL_FORCE | (write ? FOLL_WRITE : 0);

        while (count > 0) {
                int this_len = min_t(int, count, PAGE_SIZE);

                if (write && copy_from_user(page, buf, this_len)) {
                        copied = -EFAULT;
                        break;
                }

                this_len = access_remote_vm(mm, addr, page, this_len, flags);
                if (!this_len) {
                        if (!copied)
                                copied = -EIO;
                        break;
                }

                if (!write && copy_to_user(buf, page, this_len)) {
                        copied = -EFAULT;
                        break;
                }

                buf += this_len;
                addr += this_len;
                copied += this_len;
                count -= this_len;
        }
        *ppos = addr;

        mmput(mm);
free:
        free_page((unsigned long) page);
        return copied;
}
```

SELinux

Mandatory Access Control (MAC)

SELinux is a Linux Security Module which allows administrators mandatory access control. SELinux adds finer granularity to access controls.

SELinux

Mandatory Access Control (MAC)

SELinux is a Linux Security Module which allows administrators mandatory access control. SELinux adds finer granularity to access controls.

Reminder: Access check functions run as hooks in the kernel

```
SYSCALL_DEFINE5(perf_event_open,
                struct perf_event_attr __user *, attr_uptr,
                pid_t, pid, int, cpu, int, group_fd, unsigned long, flags)
{
        struct perf_event *group_leader = NULL, *output_event = NULL;
        struct perf_event *event, *sibling;
        struct perf_event_attr attr;
        struct perf_event_context *ctx, *gctx;
        struct file *event_file = NULL;
        struct fd group = {NULL, 0};
        struct task_struct *task = NULL;
        struct pmu *pmu;
        int event_fd;
        int move_group = 0;
        int err;
        int f_flags = O_RDWR;
        int cgroup_fd = -1;

        /* for future expandability... */
        if (flags & ~PERF_FLAG_ALL)
                return -EINVAL;

        /* Do we allow access to perf_event_open(2) ? */
        err = security_perf_event_open(&attr, PERF_SECURITY_OPEN);
        if (err)
                return err;
```

SELinux

Mandatory Access Control (MAC)

It has no concept of a "root" superuser.

```
avc: denied  { connectto } for  pid=2671 comm="binder_uaf" path="/dev/socket/dnsproxyd"
scontext=u:r:shell:s0 tcontext=u:r:netd:s0 tclass=unix_stream_socket
```

Source context: shell
Target context: netd
Class: unix stream socket
Permission: Connect

SELinux

Mandatory Access Control (MAC)

- Has both userspace and kernel components that enforce policy. SELinux Policy developers have to be aware of various implementation details

- It’s very complex. Hard to write scalable and maintainable policy

- Because of this, misconfigurations are common

- Implementation bugs in the kernel also occur

SELinux

Examples of Mistakes and Areas for Attack

- Not implementing granular control for new components: ex Qualcomm NPU driver - Your security is only as good as your policy.
- Doesn’t work if reachable code doesn’t have an LSM hook (i.e. io_uring)
- Incorrect implementations
    - SEPolicy disabling entire runtime mitigations: ex mmap_min_addr,
    - hook functions ex: CVE-2020-10751 netlink sendmsg message handling

Opportunity to find policy gaps such as these with the SELinux static analyzers i.e. SELint , which looks for SEPolicy convention violations, poor style and policies that could cause unexpected/insecure outcomes

Namespaces

- Way to isolate a containerized application into its own file system, process space, etc.
- Often times containers are configured s.t. the application runs with higher privileges in the container namespace. Ex: container application runs as root in its namespace
- Creating a user namespace from unprivileged is allowed by default in popular distributions
    - Bugs in "privileged" kernel code now have more severe security implications

CVE-2022-0185

- Reachable via fsconfig system call. File systems that don't set init_fs_context field in fs_context structure default to legacy (there are tons of them)
- Leads to buggy legacy code - heap overflow in legacy_parse_param due to integer underflow

```
    if (len > PAGE_SIZE - 2 - size)
            return
invalf(fc, "VFS: Legacy: Cumulative options too large");
```

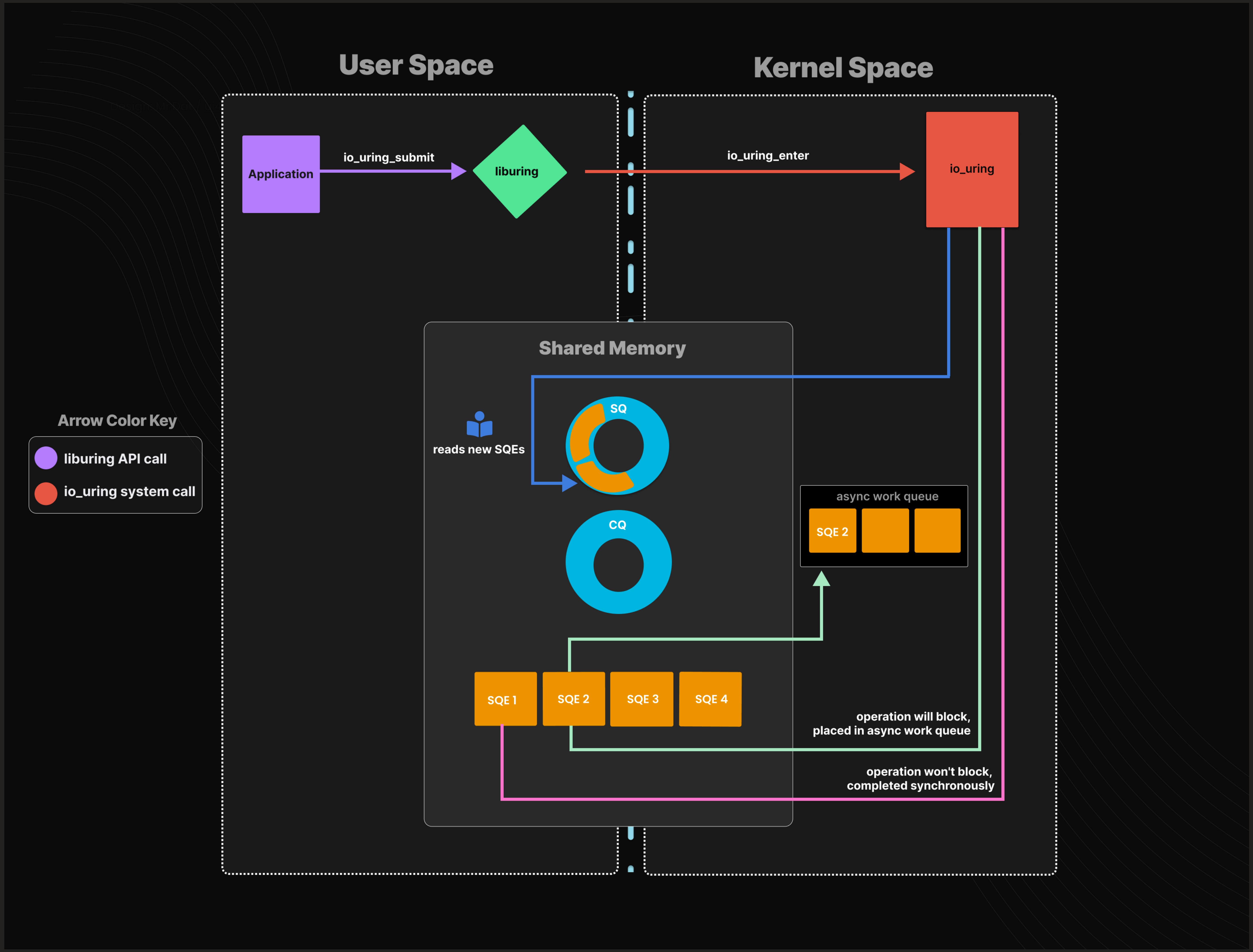
General Places to Look

General Places to Look

- Subsystems that don’t necessarily perform "privileged actions” but run complex code.

  Ex:
  - io_uring
  - NPU driver
- IPC mechanisms/protocol that allow process to access “locked down” resources
  - Binder Driver (Android)
  - Pipes, sockets, weird files
- “Weird” files and filesystem operations
- System calls or kernel entry points without LSM hooks
  - io_uring
  - (Formerly) perf

io_uring


User Space
Kernel Space
Application
io_uring_submit
liburing
io_uring_enter
io_uring
Shared Memory
reads new SQEs
SQ
CQ
async work queue
SQE 2
SQE 1
SQE 2
SQE 3
SQE 4
operation will block,
placed in async work queue
operation won't block,
completed synchronously
Arrow Color Key
liburing API call
io_uring system call

io_uring

- Redefines how system calls are done in Linux (makes async syscalls possible)
- Can be used to effectively bypass seccomp
- No LSM hooks on io_uring operations themselves -> no LSM sandboxing
    - Not a bypass LSM for system call operations, but complex io_uring code is all reachable
- Rapidly growing codebase, getting frequent major refactors — BUG$

CVE-2021-41073

- Just need to be able to read or write from a file that doesn’t implement read{write}_iter
- Lots of files don’t
  - ex: /proc/self/maps
  - Sandboxed processes usually have *some* access to procfs because LSM security context information is stored in /proc/self/attr/current.

```
 * For files that don't have ->read_iter() and ->write_iter(), handle them
 * by looping over ->read() or ->write() manually.
 */
static ssize_t loop_rw_iter(int rw, struct io_kiocb *req, struct iov_iter *iter)
{
	struct kiocb *kiocb = &req->rw.kiocb;
	struct file *file = req->file;
	ssize_t ret = 0;

	/*
	 * Don't support polled IO through this interface, and we can't
	 * support non-blocking either. For the latter, this just causes
	 * the kiocb to be handled from an async context.
	 */
	if (kiocb->ki_flags & IOCB_HIPRI)
		return -EOPNOTSUPP;
	if (kiocb->ki_flags & IOCB_NOWAIT)
		return -EAGAIN;

	while (iov_iter_count(iter)) {
		struct iovec iovec;
		ssize_t nr;

		if (!iov_iter_is_bvec(iter)) {
			iovec = iov_iter_iovec(iter);
		} else {
			iovec.iov_base = u64_to_user_ptr(req->rw.addr);
			iovec.iov_len = req->rw.len;
		}

		if (rw == READ) {
			nr = file->f_op->read(file, iovec.iov_base,
					      iovec.iov_len, io_kiocb_ppos(kiocb));
		} else {
			nr = file->f_op->write(file, iovec.iov_base,
					       iovec.iov_len, io_kiocb_ppos(kiocb));
		}

		if (nr < 0) {
			if (!ret)
				ret = nr;
			break;
		}
		ret += nr;
		if (nr != iovec.iov_len)
			break;
		req->rw.len -= nr;
		req->rw.addr += nr;
		iov_iter_advance(iter, nr);
	}

	return ret;
}
```

OK - now how do I find bugs?

- Now you have an idea of places to look - what are good strategies to find bugs?

syzkaller

- Coverage guided kernel fuzzer
- Don’t necessarily have to set up your fuzzer- can view live bugs being found on syzbot website
- Plenty of opportunities to improve coverage
    - Writing new system call descriptions for kernel interfaces with poor coverage
    - Vendor/hardware drivers that are open source but not being fuzzed
    - Tool calibration - is it finding Ndays?
- Has built in support for setuid and namespace sandboxing - can be tweaked to work with custom SELinux policy.

N-Days

- Looking at already reported bugs in a subsystem you want to target
- Understand security impact and exploitability
- Write your own exploits
    - Often leads to finding other bugs in the process
    - Bypassing sandboxing and exploit mitigations - good learning experience for learning OS internals.

Patch Gaps!

- Linux kernel culture is still very much hostile to security - “a bug is a bug”
    - Leads to obfuscating security related implications in commit messages
    - Exploitable bugs get fixed with no CVE by default!
- Often security related patches are not back ported to older kernel versions, which are used by many embedded devices.
- Individual vendors and distros are forced to cherry pick security commits. This difficult to do if there is no unified way to identify what is a security patch and what isn’t.
- Bad for security overall - recent ITW exploits targeting “0days” that have already been patched upstream for years - but good for offense :)
- Being vigilant in upstream commits yields really fruitful results with great bugs.

Questions?

Valentina Palmiotti
Twitter: @chompie1337
GitHub: chompie1337