10. Linux漏洞挖掘

游伟, 谢冬晨

中国人民大学信息学院智能软件安全研究团队

课程目录

1. 漏洞挖掘概述
2. 漏洞成因与内存模型
3. 漏洞挖掘方法论
   - 静态分析（Static Analysis）
   - 动态分析（Dynamic Analysis）
   - Fuzzing（模糊测试）
   - 符号执行（Symbolic Execution）
   - 构建完整漏洞挖掘流程
4. CodeQL 与 Syzkaller 实战

1. 漏洞挖掘概述
2. 漏洞成因与内存模型
3. 漏洞挖掘方法论
4. CodeQL 与 Syzkaller 实战

什么是漏洞挖掘（Vulnerability Discovery）

- **漏洞挖掘**指的是通过多种分析技术，**主动发现软件中的安全缺陷**。
- 对于 Linux 内核而言，漏洞挖掘具有重要意义：
  - 内核运行在 **ring 0**，漏洞影响整个系统。
  - 大量驱动、子系统代码复杂，极易出现内存错误。
  - 内核承担文件系统、网络、进程管理等功能，任何漏洞都可能造成严重后果。
- 漏洞挖掘的目标：
  - 找到缺陷 -> 精确定位触发路径 -> 构建可复现的 POC -> 协助修复。

为什么 Linux 内核漏洞很重要？

- Linux 内核是服务器、云平台、嵌入式设备、Android 的基础组件。
- 内核漏洞可能导致：
  - **本地提权（Privilege Escalation）**
  - **任意代码执行（Arbitrary Code Execution）**
  - **逃逸容器 / 沙箱（Container Escape）**
  - **永久持久化 Rootkit**
  - **影响整个系统可用性（DoS）**
- 内核漏洞的攻击面随内核规模扩大而不断增加：
  - 支持的文件系统数量不断增加
  - eBPF、io_uring、KVM 等新特性增加复杂性
  - 各种 Vendor 定制补丁引入不可控风险

Linux 内核中常见的漏洞类型（概览）

- **内存破坏类漏洞**
  - 越界访问（Out-of-Bounds）
  - Use-After-Free（UAF）
  - Double Free
  - 未初始化变量访问（Uninitialized Use）
- **并发类漏洞**
  - 锁使用错误
  - 数据竞争（Data Race）
  - TOCTOU
- **逻辑错误类漏洞**
  - 权限检查缺失（Lack of Permission Check）
  - 参数验证不完整
  - 引用计数错误（Refcount Bug）
- **系统调用接口漏洞**
  - copy_from_user / copy_to_user 错误使用
  - mmap / eBPF / io_uring 接口缺陷

漏洞示例 1：越界访问（Out-of-Bounds）

代码示例

```c
int kernel_array[10];

long my_ioctl(int index, int value) {
    // 缺失对 index 上限的检查
    if (index < 0) return -EINVAL;

    // 越界写：攻击者可控制 index 修改任意内核内存
    kernel_array[index] = value;
    return 0;
}
```

- **成因：** 未验证用户输入的索引是否在合法范围内。
- **后果：** 可导致内核崩溃（DoS）或任意地址写（提权）。

漏洞示例 2：Use-After-Free（UAF）

代码示例

```c
struct object *obj;
void thread_A() {
    kfree(obj); // 释放对象
    obj = NULL; // 理想情况应置空，但若有竞态...
}
void thread_B() {
    if (obj) {
        // 悬挂指针访问：obj 可能已被 thread_A 释放
        obj->func_ptr();
    }
}
```

- **成因：** 对象释放后，指针未被清除，仍被其他路径引用。
- **后果：** 若堆块被攻击者伪造的数据填充，可劫持控制流。

漏洞示例 3：Double Free

代码示例

```
1 void process_data(void *data) {
2     char *buf = kmalloc(64, GFP_KERNEL);
3     if (error_condition) {
4         kfree(buf); // 第一次释放
5         // 忘记 return，继续执行
6     }
7     // ...
8     kfree(buf); // 第二次释放
9 }
```

- **成因**：逻辑错误导致同一内存地址被释放两次。
- **后果**：破坏内存分配器的元数据（metadata），导致后续分配错误或崩溃。

漏洞示例 4：未初始化变量访问

代码示例

```
1 long get_info(void __user *arg) {
2     struct info_struct info; // 栈变量未初始化
3
4     info.id = 1;
5     // info.padding 或其他字段包含栈上残留的敏感数据
6
7     // 信息泄露：将未初始化的内核栈数据拷贝给用户
8     return copy_to_user(arg, &info, sizeof(info));
9 }
```

- **成因**：结构体存在 padding 或部分字段未赋值即被使用/拷贝。
- **后果**：泄露内核地址（KASLR bypass）或敏感密钥。

漏洞示例 5：并发类 - 锁使用错误

代码示例

```
1 spinlock_t my_lock;
2
3 void buggy_func() {
4     spin_lock(&my_lock);
5     if (some_error) {
6         return; // 错误：直接返回，忘记解锁！
7     }
8     spin_unlock(&my_lock);
9 }
```

- **成因**：异常处理路径遗漏解锁操作。
- **后果**：死锁（Deadlock），导致系统挂起或拒绝服务。

漏洞示例 6：并发类 - 数据竞争（Data Race）

代码示例

```
1 int global_counter = 0; // 共享资源
2
3 void thread_func() {
4     // 错误：无锁保护的并发自增
5     // 两个线程同时读到旧值，导致计数错误
6     global_counter++;
7 }
```

- **成因：** 多线程读写共享变量未加锁或未使用原子操作。
- **后果：** 状态不一致，可能导致逻辑错误或更严重的 UAF（若竞争的是指针）。

漏洞示例 7：并发类 - TOCTOU

代码示例

```
// Time-of-Check: 检查用户是否有权访问文件
if (!access(filename, R_OK)) {
    // 攻击者在此间隙将 filename 符号链接指向 /etc/passwd

    // Time-of-Use: 打开文件
    f = open(filename, O_RDONLY);
}
```

- **成因**：检查（Check）与使用（Use）非原子操作，状态在中间被改变。
- **后果**：权限绕过，访问本不该访问的资源。

漏洞示例 8：逻辑错误 - 权限检查缺失

代码示例

```
1  long dev_ioctl(struct file *f, unsigned int cmd, unsigned
       long arg) {
2      switch (cmd) {
3          case CMD_RESET_HARDWARE:
4              // 错误：未检查 CAP_SYS_ADMIN
5              // 任何普通用户都可以重置硬件
6              hardware_reset();
7              return 0;
8      }
9  }
```

- **成因：** 敏感操作未验证调用者是否具有相应 capabilities。
- **后果：** 普通用户执行特权操作（如加载驱动、修改配置）。

漏洞示例 9：逻辑错误 - 参数验证不完整

代码示例

```
1 void *alloc_buffer(int count, int size) {
2     // 错误：count * size 可能发生整数溢出
3     // 若 count=2^30, size=4 -> 结果为 0 (或小整数)
4     int bytes = count * size;
5
6     char *buf = kmalloc(bytes, GFP_KERNEL);
7     // 后续拷贝大量数据导致堆溢出
8     return buf;
9 }
```

- **成因**：未考虑整数溢出或边界情况。
- **后果**：分配过小内存块，导致后续堆溢出。

漏洞示例 10：逻辑错误 - 引用计数错误

代码示例

```
1  struct obj *get_obj() {
2      struct obj *o = lookup_obj(); // refcount++
3      if (!o) return NULL;
4
5      if (check_permission(o)) {
6          // 错误：出错返回时忘记 put (refcount--)
7          return NULL;
8      }
9      return o;
10 }
```

- **成因**：错误路径未平衡引用计数。
- **后果**：内存泄露（Memory Leak），长期运行耗尽内存；或过早释放（UAF）。

漏洞示例 11：接口漏洞 - copy_from_user

代码示例

```
1 long set_data(void __user *arg) {
2     char buf[128];
3     // 错误：未检查返回值
4     // 若用户指针无效，buf 内容未被完全初始化
5     copy_from_user(buf, arg, sizeof(buf));
6
7     process_data(buf); // 处理了未初始化的垃圾数据
8     return 0;
9 }
```

- **成因**：假设用户内存拷贝一定成功，忽略了缺页或无效地址的情况。
- **后果**：逻辑错误或使用未初始化数据。

漏洞示例 12：接口漏洞 - 复杂接口缺陷 (eBPF)

代码示例（概念）

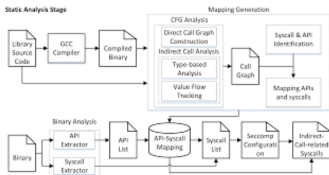
```
// eBPF 验证器逻辑
if (reg_val > 10) {
    // 验证器认为此路径不可达，标记为 safe
    // 但 JIT 编译时优化错误，导致实际可执行
}
// 实际执行中访问了越界内存
memory[reg_val] = 1;
```

- **成因：**验证器（Verifier）的静态检查模型与实际执行（JIT/Interpreter）不一致。
- **后果：**绕过安全检查，执行非法内存操作。

Linux 内核漏洞的现实案例（用于理解动机）

- **CVE-2022-0185**：堆越界写 + 命名空间逃逸 -> 本地提权到 root。
- **CVE-2017-6074**：dgram_recvmsg() 中 UAF -> 本地 root。
- **CVE-2021-3493**：OverlayFS 权限检查缺失 -> 容器逃逸。
- 这些案例告诉我们：
  - 内核漏洞不是概念性问题，而是攻击者最真实的武器。
  - 漏洞挖掘直接关系到系统攻防的前沿能力。

Linux 漏洞挖掘的一般流程



- **1. 寻找易错点：** 接口、驱动、复杂子系统
- **2. 进行静态分析或使用自动化工具：** 定位疑似点
- **3. 动态执行 / Fuzzing：** 自动生成触发输入
- **4. 捕获错误输出：** Oops/KASAN 报错
- **5. 手动调试：** QEMU + gdb 分析根因
- **6. 构造最小 POC**
- **7. 提交补丁或进行安全研究**

本课程的教学目标

- 理解 Linux 内核漏洞形成的根因
- 掌握主流漏洞挖掘技术（静态分析、动态分析、Fuzzing）
- 学会使用 syzkaller 挖掘内核漏洞
- 掌握内核漏洞调试方法（KASAN, gdb, crash）
- 能够阅读真实 CVE 的 Patch 并理解漏洞成因
- 能够完成最小 reproducer（POC）的构建

Chapter 1 小结

- Linux 内核漏洞危害极大，与系统安全直接相关。
- 内核漏洞类型包括：内存破坏、竞争、逻辑错误、接口错误等。
- 漏洞挖掘是一套系统性的流程：从分析 -> 执行 -> 调试 -> 修复。
- 后续章节将深入讲解：
  - 漏洞成因与内存模型（Chapter 2）
  - 漏洞挖掘方法论（Chapter 3）
  - 工具链：KASAN / syzkaller / 静态分析工具（Chapter 4）

1 漏洞挖掘概述

2 漏洞成因与内存模型

3 漏洞挖掘方法论

4 CodeQL 与 Syzkaller 实战

本章概览

- 本章目标：
  - 理解 C 与内核环境下的未定义行为对安全的影响；
  - 掌握 Linux 内核的主要内存分配机制（slab/slub、page、vmalloc）；
  - 理解内核对象生命周期、引用计数与常见错误模式；
  - 通过代码示例认识常见漏洞根因：越界、UAF、double free、未初始化、并发条件、接口误用等。

C 语言的未定义行为与内核安全

- 内核大量使用 **C 语言**，而 C 的“未定义行为（UB）”在内核中会放大漏洞风险：
  - 编译器可基于假设（如无越界访问）进行激进优化 -> 导致生成无法预期的代码路径。
  - 内核中低级指针运算、手动管理内存、位运算等都容易触发 UB。
- 常见导致 UB 的操作：
  - 访问已释放内存（UAF）
  - 越界读取/写入（OOB）
  - 未初始化变量的使用
  - 串行化/并发假设（data race）
- 提示：认为“代码在某台机器上没崩溃”并不代表没有 UB——UB 可能只在特定优化或特定内存布局下可观察到。

内核的内存分配模型：总体视图

User-space memory management

Kernel dynamic memory allocation

vmalloc()

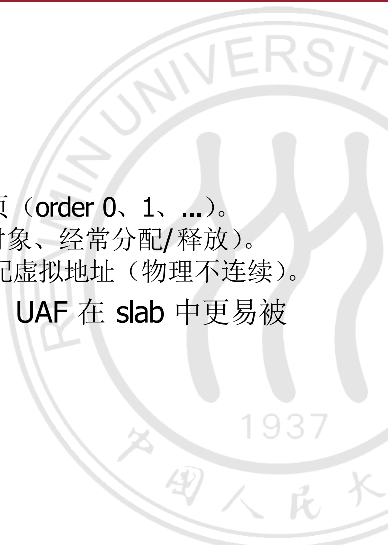
kmalloc()

SLAB layer

Low-level memory allocator (*Buddy system*)

Physical RAM

内核的内存分配模型：总体视图



- 高层次分类：
  - **页分配器（page alloc）**：分配整页（order 0、1、...）。
  - **slab/slub**：面向对象的缓存（小对象、经常分配/释放）。
  - **vmalloc**：为大块虚拟连续内存分配虚拟地址（物理不连续）。
- 不同分配器对漏洞影响不同（例如：UAF 在 slab 中更易被重用，导致可利用性更高）。

slab / slub：面向对象的内存缓存

- slab/slub 设计目的：减少频繁小对象分配的开销，复用缓存对象。
- 特点：
  - 对象按类型（cache）组织：例如 kmalloc-64，kmalloc-256 等。
  - 释放后对象可能很快被其他对象重用 -> UAF 风险高。
  - 有些配置启用 **对象填充（poison / redzone）**、**对象初始化**有助于检测错误（如 KASAN）。
- 代码示例：kmalloc/kfree 的典型使用

Listing 1: UAF 示例

```
1 struct foo *p = kmalloc(sizeof(*p), GFP_KERNEL);
2 kfree(p);
3 p->v = 1; /* UAF */
```

页分配器（page allocator）与 vmalloc 的差别

- **页分配器：**
  - 分配连续的物理页（order 可调整）。
  - 常用于大内存缓冲（如网络包缓冲、页表内存等）。
  - 越界写入跨页会破坏邻居页，后果严重。
- **vmalloc：**
  - 提供虚拟地址连续但物理不连续的区域。
  - 适合大块内核空间数据结构。
  - 缺点：访问速度慢，非 NUMA 感知。
- 教学要点：不同分配器的重用模式和布局决定了漏洞利用难度与可控性。

内核对象、生命周期与引用计数（refcount）

- 内核对象通常由结构体表示（例如 struct file, struct inode, struct sock 等）。
- **引用计数**用于管理共享对象的生命周期（atomic_t / refcount_t）。
- 常见错误模式：
  - 漏增（missing acquire）或漏减（missing release）-> 内存泄露或过早释放。
  - 对已释放对象再次 ‘put‘ 导致 double free 或引用计数下溢。
- 课堂示例：错误的 refcount 使用（伪代码）

```
1 obj_get(obj); /* supposed to increment */
2 ...
3 obj_put(obj); /* decrement */
4 obj_put(obj); /* double put -> 可能导致重复释放 */
```

常见漏洞根因—逐项详解（1/2）

1. 越界访问（Out-of-Bounds）

- 发生场景：数组索引错误、长度检查不充分、parse 数据时漏校验长度。
- 风险：可读泄露、可写破坏邻近内存（控制 metadata）。
- 示例（伪代码）：

Listing 2: 示例：基于长度缺失的越界写

```
1 void copy_user_data(char *buf, size_t len) {
2     char local[64];
3     if (len > 0) {
4         memcpy(local, buf, len); /* 若 len > 64 则越界 */
5     }
6 }
```

常见漏洞根因—逐项详解（2/2）

2. Use-After-Free (UAF)

- 场景：释放对象后仍保留指针并再次访问，或未同步的并发路径重用对象。
- 可利用性高：在 slab 中释放后对象立即被相同大小对象重分配，可把受控数据放入被重用对象。
- 示例（简化）：

Listing 3: 示例：UAF 导致控制字段被覆盖

```
1 struct foo *p = kmalloc(...);
2 do_something(p);
3 kfree(p);
4 /* 这里 p 仍被其他代码使用 */
5 use(p->ops); /* 可能跳到攻击者控制的函数指针 */
```

3. Double Free / 未初始化 / 逻辑错误（简述）

并发类漏洞：数据竞争与 TOCTOU

- 并发问题常见于多核、异步回调、工作队列、软中断、中断处理程序等场景。
- **Data race**：两个并发执行路径未同步访问同一内存（至少一个是写）。
- **TOCTOU**（time-of-check-to-time-of-use）：
  - 先检查条件（check），后执行（use），中间状态被改变 -> 条件不再成立。
  - 在权限检查与实际使用之间被利用，导致权限绕过或非法访问。
- 工具：KCSAN 用于检测数据竞争；也可以使用静态分析和手工代码审计定位。

子系统风险：驱动、文件系统、eBPF 等

- **驱动（Device Drivers）**：
  - 驱动多、来源杂、测试少；直接操作硬件与 DMA，复杂性高。
  - 用户可通过 ioctl、mmap、read/write 触发驱动未校验代码路径。
- **文件系统**：
  - 解析复杂（路径处理、权限、链接等），容易出现边界与竞态错误。
- **eBPF / io_uring / netcode**：
  - 新特性扩展了内核暴露面，攻击者通过这些复杂接口构造攻击链。
  - eBPF 能加载用户态代码到内核环境（受 verifier 限制，但仍有绕过历史）。

Chapter 2 小结

- C 的未定义行为是内核漏洞的土壤；编译器优化会放大这些风险。
- 理解内核的内存分配器（slab、page、vmalloc）对分析漏洞非常重要。
- 常见漏洞类型：越界、UAF、double free、未初始化、并发问题、接口误用。
- 驱动、eBPF、文件系统等子系统由于复杂性与暴露度，是高价值挖掘目标。
- 下一章将讲 **漏洞挖掘方法论**：静态分析、动态分析、fuzzing、符号执行等（Chapter 3）。

1. 漏洞挖掘概述
2. 漏洞成因与内存模型
3. 漏洞挖掘方法论
   - 静态分析（Static Analysis）
   - 动态分析（Dynamic Analysis）
   - Fuzzing（模糊测试）
   - 符号执行（Symbolic Execution）
   - 构建完整漏洞挖掘流程
4. CodeQL 与 Syzkaller 实战

本章目标

- 介绍系统化的内核漏洞挖掘方法；
- 讲解静态分析、动态分析、fuzzing、符号执行的工作原理；
- 分析常见工具链（clang sanitizer, syzkaller, KASAN 等）；
- 结合真实漏洞的挖掘流程构造完整“分析链”。

漏洞挖掘的基本思路：两类路径

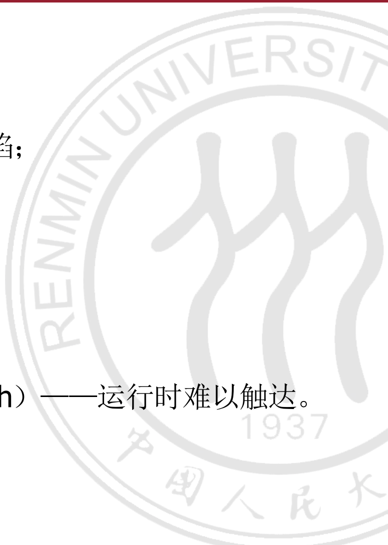
- 漏洞挖掘通常遵循两类主线：
  1. **通用路径（generic approach）**基于内核通用组件（驱动、文件系统、网络栈、eBPF...）的已知脆弱性；
  2. **定向路径（targeted approach）**针对单个模块、单个系统调用、单个 driver 做深入分析。
- 这两类路径通常结合使用：
  - 先用自动化工具发现异常点；
  - 再进行手工分析确定漏洞根因与可利用性。

内核漏洞挖掘的基础技能树

- 代码理解与架构图阅读；
- 内存分配器行为、对象生命周期；
- 用户态接口语义分析；
- 并发路径、锁逻辑；
- 工具链（fuzzer、sanitizer、调试器）使用；
- 漏洞复现与触发条件控制。

1. 漏洞挖掘概述
2. 漏洞成因与内存模型
3. 漏洞挖掘方法论
   - 静态分析（Static Analysis）
   - 动态分析（Dynamic Analysis）
   - Fuzzing（模糊测试）
   - 符号执行（Symbolic Execution）
   - 构建完整漏洞挖掘流程
4. CodeQL 与 Syzkaller 实战

静态分析的目的与优势



- 在不执行代码的前提下发现潜在缺陷；
- 对“结构性错误”有高效检测能力：
  - 空指针检查不足；
  - 越界索引；
  - 未初始化变量；
  - 串行化错误（缺锁/多锁）；
  - API 误用（例如 put/get 不匹配）。
- 可以覆盖代码中极端路径（rare path）——运行时难以触达。

LLVM/Clang 静态分析与漏洞挖掘

- Clang/LLVM 提供丰富的静态分析能力：
  - `clang-tidy`：可配置的风格/缺陷检查器，支持自定义规则；
  - Clang Static Analyzer：基于路径敏感分析，能发现内存泄露、UAF、越界等；
  - LLVM IR Pass：在中间表示层面做数据流、污点分析，适合研究型挖掘。
- 优势：
  - 与编译器器紧密集成，能理解复杂宏与模板展开；
  - 可在 CI 中自动跑，持续发现回归问题；
  - 方便做“定制检查器”，针对内核 API 误用写专门规则。

课堂练习：Clang 静态分析实战

1. 构造一个简单的漏洞示例 (demo.c)

```
1 void test_func(int *p) {
2     int v;
3     if (p) { v = *p; }
4     /* 路径敏感分析：若 p 为空，v 未初始化 */
5     if (v == 10) { /* Clang 能够发现 v 可能未初始化 */
6         free(p);
7     }
8 }
```

2. 运行 Clang 静态分析

```
1 # -Xanalyzer -analyzer-output=text 可输出详细路径
2 clang --analyze -Xanalyzer -analyzer-output=text demo.c
```

课堂练习：结合 Sanitizer 动态检测

3. 编译并启用 AddressSanitizer (ASAN)

```
1 # 编译时加入 -fsanitize=address -g
2 clang -fsanitize=address -g demo.c -o demo_asan
```

- **实验步骤：**
  1. 编写 main 函数构造触发路径；
  2. 运行编译后的程序 ./demo_asan；
  3. 观察 ASAN 输出的报错堆栈（如 Use-after-free, heap-buffer-overflow）。

CodeQL：基于查询的漏洞挖掘

- 核心思想：将代码视为数据库，用逻辑查询语言（QL）编写“漏洞模式查询”。
- 工作流程：
  - 用 CodeQL CLI/VS Code 插件从源码构建数据库（CodeQL DB）；
  - 在 DB 上运行内置或自定义 QL 查询；
  - 对命中结果进行人工审计，筛选出真实漏洞。
- 适合场景：
  - 在大规模代码库中批量搜“同类错误”(某一 API 误用模式);
  - 跟随漏洞补丁写规则，回溯历史版本寻找“同模式”缺陷。

CodeQL 查询示例：未检查的 copy_from_user

- 目标：查找没有检查返回值就继续使用缓冲区的 `copy_from_user` 调用。

```
1 ssize_t my_ioctl(void __user *u_hdr)
2 {
3     struct hdr h;
4     copy_from_user(&h, u_hdr, sizeof(h));
5     /* 忘记检查返回值，继续使用 h -> 风险 */
6     return do_something(&h);
7 }
```

CodeQL 查询示例：QL 代码片段

- 简化版 QL 查询（概念示意）：

```
/** 查找未检查返回值的 copy_from_user 调用 */
import cpp

from FunctionCall call, Function f
where
  call.getTarget().hasName("copy_from_user") and
  // 返回值未被使用
  not exists(Expr use | use = call.getParent*)
select call, "copy_from_user␣result␣is␣ignored␣here"
```

- 实际规则会更复杂：需要考虑封装函数、宏、数据流等。

静态分析工具链

- **编译期分析：**
  - clang static analyzer；
  - gcc warnings（-Warray-bounds、-Woverflow 等）；
  - sparse（内核常用，对类型安全极为敏感）。
- **自动化扫描器：**
  - smatch（长期用于内核补丁审查）；
  - coccinelle/spatch：通过语义匹配查找 API 误用；
  - commercial 工具：CodeQL、Coverity。
- **patch review + 静态模式分析** Linux kernel 在提交 patch 时会自动跑 sparse + smatch。

静态分析工具对比小结

- **sparse / smatch**：
  - 优点：与内核社区深度集成，规则贴合内核编码规范；
  - 缺点：对非内核项目支持有限，表达复杂跨函数数据流较弱。
- **LLVM/Clang 分析**：
  - 优点：与编译器绑定，能理解复杂宏/模板，易于在 CI 部署；
  - 缺点：更偏通用 C/C++ 项目，对内核特定约束需自定义检查器。
- **CodeQL**：
  - 优点：查询语言强大，适合“全局搜同类漏洞”和追踪补丁模式；
  - 缺点：学习曲线相对陡峭，构建大项目数据库成本较高。

使用 sparse 检测类型错误（示例）

- sparse 能追踪内核自定义的地址空间标记（__user、__kernel）；
- 正确使用可以找到 copy_to_user/copy_from_user 类误用。

```
1  /* sparse 可检测: 将 __user 指针当作内核指针使用 */
2  int foo(int __user *p)
3  {
4      int v = *p; /* ERROR: dereference of user pointer */
5      return v;
6  }
```

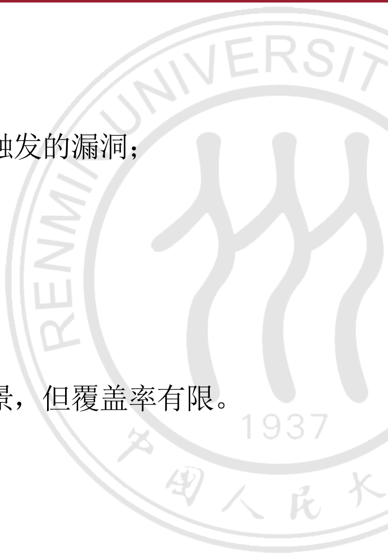
1 漏洞挖掘概述

2 漏洞成因与内存模型

3 漏洞挖掘方法论
- 静态分析（Static Analysis）
- 动态分析（Dynamic Analysis）
- Fuzzing（模糊测试）
- 符号执行（Symbolic Execution）
- 构建完整漏洞挖掘流程

4 CodeQL 与 Syzkaller 实战

动态分析的目的



- 在运行时监控内核行为，捕获实际触发的漏洞；
- 常常可以检测：
  - UAF；
  - OOB；
  - 未初始化内存使用；
  - 并发数据竞争；
  - 死锁/锁反转。
- 相比静态分析：更接近真实利用场景，但覆盖率有限。

主要动态分析工具

- **KASAN**：检测堆越界 / UAF；
- **UBSAN**：检测未定义行为；
- **KCSAN**：检测并发数据竞争；
- **KFENCE**：高性能边界检查；
- **lockdep**：检测死锁路径；
- **kmemleak**：检测内核内存泄露；
- **debugfs + tracepoint**：用于复杂路径调试；
- **ftrace / perf**：分析运行时函数调用。

示例：KASAN 检测 UAF

```
1 static int test(void)
2 {
3     char *p = kmalloc(32, GFP_KERNEL);
4     kfree(p);
5     return p[0]; /* KASAN: use-after-free */
6 }
```

- KASAN 输出中会包含：
  - 崩溃发生位置；
  - 分配 / 释放的调用栈；
  - 对象的 cache 名称（如 kmalloc-32）。

1. 漏洞挖掘概述
2. 漏洞成因与内存模型
3. 漏洞挖掘方法论
   - 静态分析（Static Analysis）
   - 动态分析（Dynamic Analysis）
   - Fuzzing（模糊测试）
   - 符号执行（Symbolic Execution）
   - 构建完整漏洞挖掘流程
4. CodeQL 与 Syzkaller 实战

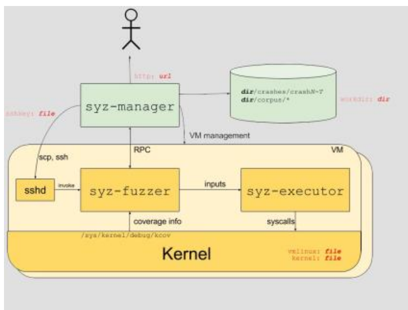
Fuzzing 在内核漏洞挖掘中的关键地位

- 当前最成功的内核漏洞发现方式；
- syzkaller 主导了过去多年 Linux 0-day 的发现；
- 能自动探索复杂接口（syscall、ioctl、eBPF 等）；
- 结合 KASAN/KCOV，能高覆盖率触发深层路径；
- 自动生成最小化 POC（syzkaller-repro）。

内核 fuzzing 的核心组件

- 输入生成器（mutator）；
- 覆盖率反馈（coverage feedback）；
- 语义引导（例如 syzbot 的接口模型）；
- 环境控制（QEMU/KVM、snapshot、crash restore）。

syzkaller 体系结构与工作机制



- **syz-manager**：协调任务、分发工作；
- **syz-fuzzer**：生成变异输入；
- **syz-executor**：执行系统调用序列；
- 使用 KCOV 收集 coverage 信息；
- 不断迭代生成触发更多覆盖率的输入。

syzkaller 简化示例（系统调用序列）

```
1 r0 = socket$inet_tcp()
2 bind$inet(r0, addr, 16)
3 listen(r0, 1)
4 accept(r0)
```

- syzkaller 能自动组合 syscall，从而探索复杂路径；
- fuzzing 生成的序列可能触发：
  - race condition；
  - 内存越界；
  - interface misuse；
  - 未处理的 error path。

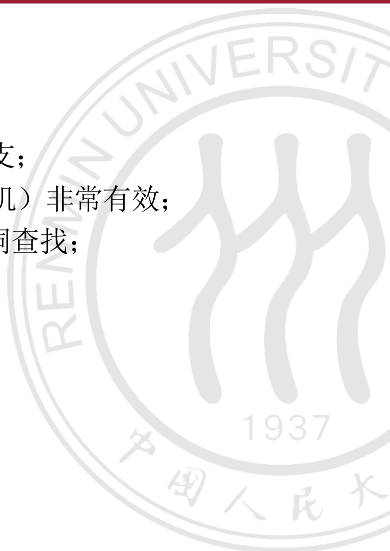
1 漏洞挖掘概述

2 漏洞成因与内存模型

3 漏洞挖掘方法论
- 静态分析（Static Analysis）
- 动态分析（Dynamic Analysis）
- Fuzzing（模糊测试）
- 符号执行（Symbolic Execution）
- 构建完整漏洞挖掘流程

4 CodeQL 与 Syzkaller 实战

符号执行的作用与局限性



- 能自动分析路径条件，覆盖极深分支；
- 对复杂条件分支（权限判断、状态机）非常有效；
- 适合驱动与协议 handler 的语义漏洞查找；
- 局限：
  - 路径爆炸问题（path explosion）；
  - 内核环境复杂，需隔离执行环境；
  - 输入规模大，建模困难。

符号执行工具链

- KLEE（最经典的符号执行器）；
- syzkaller + concolic execution（实验性支持）；
- angr（可处理 Linux kernel 部分模块）；
- S2E（支持系统级符号执行，QEMU 插桩）。

1. 漏洞挖掘概述
2. 漏洞成因与内存模型
3. 漏洞挖掘方法论
   - 静态分析（Static Analysis）
   - 动态分析（Dynamic Analysis）
   - Fuzzing（模糊测试）
   - 符号执行（Symbolic Execution）
   - 构建完整漏洞挖掘流程
4. CodeQL 与 Syzkaller 实战

完整漏洞挖掘流程（1/2）

1. **确定目标面：** syscall、ioctl、驱动、fs、网络、eBPF；
2. **收集代码结构信息：** 阅读入口函数、数据流、对象模型；
3. **构造输入模型**（struct, buffer, flags）；
4. **跑 fuzzing + sanitizer**；
5. **收集 crash 信息**（堆栈、对象类型、分配点）；

完整漏洞挖掘流程（2/2）

6. 手工分析 **crash** 是否为真实漏洞；
7. 定位触发条件，最小化 **POC**；
8. 构造利用条件（可选）；
9. 修复漏洞并验证；
10. 撰写报告或提交 **patch**。

Chapter 3 小结

- 静态分析：结构性错误定位高效；
- 动态分析：运行时错误检测全面；
- fuzzing：现代漏洞挖掘核心技术；
- 符号执行：覆盖分支复杂但成本高；
- 真实漏洞挖掘需要组合使用多种方法。

1 漏洞挖掘概述

2 漏洞成因与内存模型

3 漏洞挖掘方法论

4 CodeQL 与 Syzkaller 实战

本章目标

- 通过一个真实的 UAF 漏洞示例贯穿静态分析与动态 Fuzz；
- 展示如何用 CodeQL 编写查询并定位漏洞；
- 展示如何编写 Syzkaller 程序及描述文件触发漏洞；
- 最终构建一条完整的“漏洞挖掘链路”。

示例漏洞：错误路径 Double-Free 导致 UAF

- 场景：驱动/字符设备中常见错误处理逻辑；
- 调用链：my_ioctl() → create_session() → 错误路径；
- 触发点：copy_from_user() 失败 → 提前释放 buffer；
- 后果：destroy_session() 再次释放 → Double-Free → UAF。

核心漏洞代码（节选）

```c
if (copy_from_user(tmp, (void __user *)arg, 64)) {
    kfree(s->buffer);    /* 第一次 free */
    kfree(tmp);
    return -EFAULT;
}
...
destroy_session(s);      /* 第二次 free -> UAF */
```

漏洞成因分析（结构化）

- **资源管理不一致：**成功路径使用 destroy_session() 释放所有资源；
- **错误处理路径提前释放资源：**copy_from_user() 失败时手动 kfree()；
- **双重释放导致悬挂指针：**
  - s->buffer 被提前释放；
  - destroy_session() 再次 free；
- **可能被攻击者利用：**
  - 内核堆块被重新占用；
  - UAF 读写导致代码执行、越界写等利用链。

CodeQL 静态分析目标

- 自动识别“同一指针被释放两次”的路径；
- 追踪指针跨函数的流程（inter-procedural dataflow）；
- 检测在错误路径被释放后又在退出路径中再次释放；
- 最终在大型代码库中缩小搜索范围并定位漏洞。

核心思路

1. 捕获所有指向 `kfree()` 的调用；
2. 提取每次 free 的指针表达式；
3. 对比是否为“同一指针”；
4. 判断是否存在“后 free”的路径。

CodeQL 查询示例：检测 Double-Free

```
1  import cpp
2
3  class FreeCall extends FunctionCall {
4    FreeCall() { this.getTarget().getName() = "kfree" }
5    Expr freedPtr() { result = this.getArgument(0) }
6  }
7
8  from FreeCall f1, FreeCall f2
9  where
10   f1 != f2 and
11   f1.freedPtr().getFullyQualifiedName() =
12     f2.freedPtr().getFullyQualifiedName() and
13   f2.getLocation().getStartLine() >
14     f1.getLocation().getStartLine()
15 select f2, "Double-free␣detected;␣previously␣freed␣here:",
       f1
```

说明：

CodeQL 分析结果演示

- CodeQL 在分析 create/destroy + ioctl 的函数调用链后发现：
  1. 错误路径上提前释放 s->buffer；
  2. 对象随后被 destroy_session() 回收；
- 自动定位漏洞点：
  - 第一次 free：copy_from_user() 错误处理；
  - 第二次 free：destroy_session();

Syzkaller 动态触发目标

- 找到能进入错误路径的输入；
- 触发 copy_from_user() 失败；
- 使得提前 free 被激活；
- destroy_session() 被执行 → Double-Free → UAF。

Fuzzing 原则

1. 用错误的“用户态指针”触发拷贝失败；
2. 尽量覆盖更多 ioctl 参数组合；
3. 让内核路径进入“异常路径”。

Syzkaller 接口描述文件（syzlang）

```
1 ioctl$uafdev(fd fd, cmd const[int], arg intptr) {
2   cmd = CMD_PROCESS
3 }
```

- 建立“内核接口 → 模糊定义”关系；
- syzkaller 就能理解并自动调用 ioctl；
- 可以进一步设置 ioctl 参数为各种组合；
- 自动探索更多执行路径。

Syzkaller 触发程序

```
1 r0 = openat(AT_FDCWD, "/dev/uafdev", O_RDWR, 0)
2 ioctl$uafdev(r0, CMD_PROCESS, 0xffffffffffffffff)
```

- 传入无效用户态指针：0xffffffffffffffff；
- copy_from_user() 必然失败；
- 进入错误路径 → 第一次 free；
- 最终 destroy_session() 执行第二次 free；
- Syzkaller + KASAN 可立即报 UAF。

Syzkaller + KASAN 报错示例

- Syzkaller 会在触发 UAF 后自动记录崩溃；
- KASAN 给出精确的“第一次 free + 第二次 free”栈；
- 报错信息典型格式：

示例 KASAN 报错

```
KASAN: double-free in destroy_session+0x34
First free at:   copy_from_user_fail+0x20
Second free at:   destroy_session+0x34
```

完整漏洞挖掘链路总结

- **静态分析（CodeQL）：**
  - 自动检测双重释放；
  - 输出跨函数调用链；
- **动态分析（Syzkaller）：**
  - 自动探索执行路径；
  - 使用无效指针触发 copy_from_user() 错误路径；
  - KASAN 自动报告 UAF。