上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

网络空间安全学院
SJTU-CSE

Detecting Kernel Memory Leaks in Specialized Modules With Ownership Reasoning

NIS-8018 论文阅读

谌翔　杨光

2022 年 4 月

目录

1. Introduction
2. K-MELD
3. Ownership Reasoning
4. Implementation
5. Evaluation
6. Conclusion

第 1 节

Introduction

Background

- Kernel Memory
  - Shared by hardware and all processes
  - Hard to reclaim
  - Denial of Service
- Challenges
  - Specialized functions.
  - Complicated and lengthy data flow.

Contributions

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

- An approach for identifying specialized allocation functions.
- A rule-mining approach for corresponding specialized deallocations.
- An ownership reasoning mechanism for kernel objects.
- A scalable implementation K-MELD(Kernel MEmory Leak Detector)
  - 218 new bugs, 41 CVEs assigned.

CVE-2019-19062

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

```c
/* File: crypto/crypto_user_base.c */
static int crypto_report(struct sk_buff *in_skb,
        struct nlmsghdr *in_nlh, struct nlattr **attrs)
{
    struct crypto_dump_info info;
    ...
    alg = crypto_alg_match(p, 0);
    if (!alg)
        return -ENOENT;

    err = -ENOMEM;
    /* Memory is allocated here */
    skb = nlmsg_new(NLMSG_DEFAULT_SIZE, GFP_KERNEL);
    if (!skb)
            goto drop_alg;
    ...
    info.out_skb = skb;
    ...
    err = crypto_report_alg(alg, &info);

drop_alg:
    crypto_mod_put(alg);

    if (err)
            return err; /* Memory leaks here */
     ...
}
```

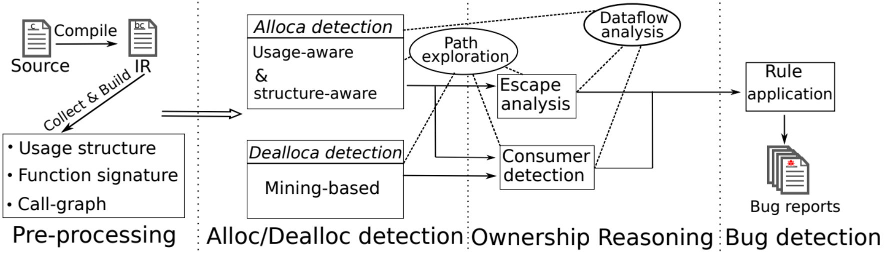
第 2 节

K-MELD

Kernel Memory Allocation and Leaks

- kmalloc & kfree
- vmalloc & vfree
- Specialized allocation
  - netlink: nlmsg_new & nlmsg_free

K-MELD


Source
Compile
IR
Collect & Build
• Usage structure
• Function signature
• Call-graph
Pre-processing
Alloca detection
Usage-aware
&
structure-aware
Dealloca detection
Mining-based
Alloc/Dealloc detection
Path exploration
Dataflow analysis
Escape analysis
Consumer detection
Ownership Reasoning
Rule application
Bug reports
Bug detection

Allocation and Deallocation Identification

- Properties of allocation functions
  - Return a pointer
  - Followed by a NULL check
  - Not derived from another base pointer
  - Initialized before being used
- Methods
  - Use-finding
  - Source-finding

Allocation and Deallocation Identification

- Detect deallocation functions
  - Error-Handling Paths
    - Explore the control flow graph(CFG)
  - Sequential Pattern Mining
    - $\langle$call FOI, check, release, return$\rangle$

第 3 节

Ownership Reasoning

Ownership

何谓 Ownership[1] ⟶ 对于动态分配内存的管理权

- 申请该动态内存的函数也负责了这块内存的回收
- 如果将动态分配的内存传递[2]给了 caller 或者 callee， ownerships 随之转移

[1] https://www.oreilly.com/library/view/programming-rust/9781491927274/ch04.html
[2] 通过引用或者全局变量

Ownership

Ownership Reasoning 的作用 ⟶ 辅助过程间分析，追踪 ownership 的转移情况和内存的释放情况

推断方式	推断方向	作用
Enhanced escape analysis	CFG Upward	减少 caller 释放造成的误报
Consumer fuction detection	CFG Downward	减少 callee 造成的误报和漏报

Enhanced escape analysis

Escape：ownership 被转移到了上层函数，当前函数不再需要负责内存的回收。

- **path-sensitive analysis**：不需要再检测 escape 所在路径的内存泄漏情况
- **inter-procedure analysis**：追踪 escape 以外的路径或遍历 escape 指针可能到达的上层函数，检测内存泄漏情况

Case Study1 - Enhanced escape analysis

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

```c
/* File: drivers/virt/vboxguest/vboxguest_utils.c */
static int hgcm_call_preprocess_linaddr(
    const struct vmmdev_hgcm_function_parameter *src_parm,
    void **bounce_buf_ret, size_t *extra)
{
    void *buf, *bounce_buf;
    bool copy_in;
    u32 len;
    int ret;

    buf = (void *)src_parm->u.pointer.u.linear_addr;
    len = src_parm->u.pointer.size;
    copy_in = src_parm->type != VMMDEV_HGCM_PARM_TYPE_LINADDR_OUT;

    if (len > VBG_MAX_HGCM_USER_PARM)
        return -E2BIG;

    /* Memory is allocated here */
    bounce_buf = kvmalloc(len, GFP_KERNEL);
    /* Check for allocation success */
    if (!bounce_buf)
        return -ENOMEM;

    if (copy_in) {
        ret = copy_from_user(bounce_buf, (void __user *)buf, len);
        if (ret)
                return -EFAULT;
    } else {
        memset(bounce_buf, 0, len);
    }

    /* Allocation pointer is assigned to a reference argument */
    *bounce_buf_ret = bounce_buf;
    hgcm_call_add_pagelist_size(bounce_buf, len, extra);
    return 0;
}
```

Consumer fuction detection

Consumer：ownership 被转移到了下层函数，需要追踪 callee 的内存回收情况。如果 callee 的所有路径都对指针进行了逃逸/释放的操作，则 callee 不产生内存泄漏。

- **path-sensitive analysis**：探索每条路径的内存泄漏情况
  - **Unconditional**：caller 的所有执行路径都会调用 consumer
  - **Conditional**：caller 的部分执行路径不会调用 consumer
- **inter-procedure analysis**：需要继续追踪 consumer 中内存对象 escape 的情况

Case Study2 - Unconditional consumer

SHANGHAI JIAO TONG UNIVERSITY

```c
/* File:  drivers/net/ethernet/chelsio/cxgb4/srq.c */
int cxgb4_get_srq_entry(struct net_device *dev,
            int srq_idx, struct srq_entry *entryp)
{
    ...
    struct adapter *adap;
    struct sk_buff *skb;
    ...
    adap = netdev2adap(dev);
    ...
    /* ALLOCATION */
    skb = alloc_skb(sizeof(*req), GFP_KERNEL);
    if (!skb)
            return -ENOMEM;
    ...
    t4_mgmt_tx(adap, skb);  /* CONSUMER */
    ...
    return rc;
}

/* File: drivers/net/ethernet/chelsio/cxgb4/sge.c */
```

```c
/* File: drivers/net/ethernet/chelsio/cxgb4/sge.c */
int t4_mgmt_tx(struct adapter *adap, struct sk_buff *skb)
{
    int ret;
    ...
    ret = ctrl_xmit(&adap->sge.ctrlq[0], skb); /* CONSUMER */
    ...
    return ret;
}

static int ctrl_xmit(struct sge_ctrl_txq *q, struct sk_buff *skb)
{
    ...
    if (unlikely(!is_imm(skb))) {
            WARN_ON(1);
            dev_kfree_skb(skb);     /* RELEASE */
            return NET_XMIT_DROP;
    }
    ...
    if (unlikely(q->full)) {
    ...
        __skb_queue_tail(&q->sendq, skb);    /* ESCAPE */
        spin_unlock(&q->sendq.lock);
        return NET_XMIT_CN;
    }
    ...
    kfree_skb(skb);     /* RELEASE */
    return NET_XMIT_SUCCESS;
}
```

Case Study3 - Conditional consumer

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

```c
/* File: net/dsa/tag_ksz.c */
static struct sk_buff *ksz_common_xmit(struct sk_buff *skb,
    struct net_device *dev, int len)
{
    nskb = alloc_skb(NET_IP_ALIGN + skb->len +
            padlen + len, GFP_ATOMIC);
    if (!nskb)
        return NULL;
    ...
    /* CONDITIONAL-CONSUMER */
    if (skb_put_padto(nskb, nskb->len + padlen))
        return NULL;
    ...
    return nskb;
}
/* File: net/core/skbuff.c */
```

```c
/* File: net/core/skbuff.c */
int __skb_pad(struct sk_buff *skb, int pad, bool free_on_error)
{
    int err;
    int ntail;
    if (!skb_cloned(skb) && skb_tailroom(skb) >= pad) {
        memset(skb->data+skb->len, 0, pad);
        return 0;
    }
    ...
    if (likely(skb_cloned(skb) || ntail > 0)) {
        err = pskb_expand_head(skb, 0, ntail, GFP_ATOMIC);
        if (unlikely(err))
            goto free_skb;
    }
    err = skb_linearize(skb);
    if (unlikely(err))
        goto free_skb;
    memset(skb->data + skb->len, 0, pad);
    return 0;
free_skb:
    ...
    kfree_skb(skb);
    return err;
}
```

第 4 节

Implementation

Implementation

- 工具链：LLVM Pass[3] 和 Python 脚本
- 思路：排除掉上述两种情况造成的误报后，对剩余的路径进行模式匹配，判断是否正确回收内存
- 过程：
  1. 分析 call 指令的参数，追踪已分配内存对象的指针在函数之间的传递
  2. 处理 ownership 的转移，排除掉上述两种情况造成的误报
  3. 从剩余的路径提取 <call FOI, check, call release, return> 四元组，进行模式匹配，判断是否正确回收内存
  4. 对于漏洞样本进行手工分析，向 Linux 上游提交 patch

[3] https://llvm.org/docs/WritingAnLLVMPass.html

第 5 节

Evaluation

Scalability

- 实验对象：5.2.13 版本的 Linux kernel 进行，将其编译为 allyesconfig[4]版本的 bitcode
- 实验环境：48 核 Intel Xeon CPU，256G 内存的服务器上进行

过程	耗时	备注
生成 bitcode	5 hours	可重用
收集内存分配函数	2 hours	可重用
内存回收函数模式匹配	1 hour	
对 FOI 的检查	几分钟到 4 个小时不等，平均 3 分钟	可以独立并行

[4]编译尽可能多的内核驱动模块

Set of Allocations and Associated Deallocations

- 从 4621 个备选函数中，提取出了 807 个内存分配/回收函数对[5]，15 个为误报
- 其中 21 个为 primitive allocators，即实际操作内存的函数，其余大部分为 specialized allocator

  To the best of our knowledge, none of the previous detection techniques used such a rich set of allocation-deallocation functions.

[5] https://github.com/Navidem/k-meld/blob/main/results/FOIs.txt

Bug finding

- 一共报出 458 个 warnings，手工分析后认为 218 处存在内存泄漏漏洞，即误报率为 52%
- 向 Linux 内核提交了 107 个 patch，申请到了 41 个 CVE
- Bug 分布在各类函数中，例如 kmalloc (9244 call-sites)，sync_file_alloc（2 call-sites)
- 有 115 个 bugs 来自于 specialized modules（call-sites $\leq$ 400）的函数

Exploitability Analysis

- 使用 fuzzing 或者符号执行的方法花销很大
- Linux 内核中存在三类用户可控输入，即 entry points
- 构建 CFG，找到 entry point 和 leak 所在函数的最短路径
- 83.9% 的 bugs 能够被本地用户输入触发

Entry functions	**Attacker**	**Count**
System calls	Userspace	137
Ioctl handlers	Userspace	173
IRQ handlers	Hardware	173
Reachable from any entry		182 (83.9%)

TABLE I: The number of reachable bugs for different types of entry functions. IRQ = Interrupt request, Ioctl = I/O control.

False Positive&Negative Analysis

CVE #	Reproducible	K-MELD Success/Failure
CVE-2019-8980	✓	Success
CVE-2019-9857	✓	Success
CVE-2019-16995	✓	Failure
CVE-2019-16994	✓	Success
CVE-2019-15916	✓	Success
CVE-2019-15807	✓	Success
CVE-2019-12379	✓	Success (correctly rejected)
CVE-2018-8087	✓	Success
CVE-2018-7757	✓	Success
CVE-2018-6554	X	—
CVE-2016-9685	✓	Success
CVE-2016-5400	✓	Success
CVE-2015-1339	X	—
CVE-2015-1333	X	—
CVE-2014-8369	✓	out-of-scope
CVE-2014-3601	✓	out-of-scope
CVE-2010-4250	X	—

- **False Positive**
  - 不可行的执行路径，可以通过符号执行来缓解
  - 对于 specified 内存回收函数的识别错误
- **False Negative**
  - 过于复杂的指针传递
  - Linux 内核版本不同

Effectiveness of Escape and Consumer Analysis

- 在分别禁用 escape analysis 和 consumer analysis 两种分析的情况下，工具报告的数量上升到 **1292** 和 **1386** 个
- 随机抽取 20 个新增的报告手工分析均为误报，证明两种方法确实有效

第 6 节

Conclusion

简单复现

- 论文代码：https://github.com/Navidem/k-meld
- 从大型项目构建 bitcode：https://github.com/travitch/whole-program-llvm
- 尝试分析一下 FreeBSD：https://github.com/travitch/whole-program-llvm/blob/master/doc/tutorial-freeBSD.md[6]

[6] K-MELD is also extendable to other OS kernels like FreeBSD.

阅读感想

- Linux 内核非常复杂，漏洞涉及到的调用链长，漏洞验证很麻烦
- 专注于某一类漏洞（memory leak，memory corruption，data race，lock 等）的研究
- 作者在 Linux 内核安全研究领域引发的一些学术伦理讨论[7]

[7] https://www.inforsec.org/wp/?p=4893

谢谢

谌翔 杨光 · Detecting Kernel Memory Leaks in Specialized Modules With Ownership Reasoning