计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(10):35−44 [doi: 10.15888/j.cnki.csa.007099]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于配对函数调用场景的设备驱动漏洞检测①

王 佳, 翟高寿, 刘 峰, 李红辉

(北京交通大学 计算机与信息技术学院, 北京 100044)
通讯作者: 王 佳, E-mail: 16120421@bjtu.edu.cn

摘 要: 由于 Linux 系统的设备驱动工作在内核模式中, 在这种特定的工作场景下, 由设备驱动引发的漏洞问题极易影响操作系统的稳定性和安全性. 当前在各类设备驱动漏洞中所占比例较高的当属资源操作类漏洞, 针对这种情况, 我们提出了一种基于配对函数调用场景的设备驱动漏洞检测方法. 首先引入配对函数的概念, 据此对特定的驱动程序做配对函数的自动提取与优化; 随后结合手工分析结果构建配对函数在资源申请与释放过程中的执行路径; 最后基于相应的函数调用场景进行配对检查, 检测并验证设备驱动程序中内存资源的申请和释放是否为完全层次性匹配. 为验证该方法的有效性, 实验分别选取不同的设备驱动应用该漏洞检测方法, 记录相应的漏报率、误报率及覆盖度. 实验结果表明, 该设备驱动漏洞检测方法精确率较高, 检测速度快. 并且该方法不依赖于实时编译以及硬件设备等条件.

关键词: Linux; 设备驱动; 配对函数; 调用场景; 漏洞检测

引用格式: 王佳,翟高寿,刘峰,李红辉.基于配对函数调用场景的设备驱动漏洞检测.计算机系统应用,2019,28(10):35–44. http://www.c-s-a.org.cn/1003-3254/7099.html

Vulnerability Detection of Device Drivers Based on Pair Functions' Calling Context

WANG Jia, ZHAI Gao-Shou, LIU Feng, LI Hong-Hui

(School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China)

Abstract: Since the device drivers of Linux work in the kernel mode, in this specific work scenario, the vulnerability caused by the device drivers can easily affect the stability and security of the operating system. At present, the most proportion of various types of device drivers' vulnerabilities is resource operation vulnerability. In this case, a vulnerability device detection method of device drivers based on pair functions' calling context is proposed. Firstly, we introduced the concept of pair function, according to which the automatic extraction and optimization of the pair function were performed for the specific drivers. Then the execution path of the pair function in the resource request and release process was recorded based on manual analysis results. Finally, the pair function was combined with the corresponding calling context scenario to verify whether the application and release of memory resources in the device driver matched in the hierarchy exactly. In order to verify the effectiveness of this method, vulnerability detection method was applied to different drivers in the experiment, and the corresponding false negative, false positive, and coverage were recorded. The experimental results show that the device drivers' vulnerability detection method has higher accuracy and faster detection speed, and the method does not depend on conditions such as real-time compilation and hardware devices.

Key words: Linux; device driver; pair function; calling context; vulnerability detection

① 基金项目: 国家重点研发计划 (2016YFF0204002); 教育部产学合作协同育人项目 (201702025004)
Foundation item: National Key Research and Development Program of China (2016YFF0204002); MOE Industry-University Cooperation Project for Collaborative Education (201702025004)
收稿时间: 2019-03-16; 修改时间: 2019-04-17; 采用时间: 2019-04-19; csa 在线出版时间: 2019-10-15

© 中国科学院软件研究所 http://www.c-s-a.org.cn

设备驱动程序作为操作系统中的重要组成部分, 占据了 Linux 内核源码约 70% 的部分. 2001 年, Chou 等人[1]最早通过将静态分析器应用于 Linux 1.0 版本至 2.4.1 版本用以进行故障分析, 研究表明驱动程序目录包含的错误比内核中的其他目录多达 7 倍. 驱动程序相当于一个处于硬件和应用程序之间的软件接口, 它负责对硬件设备底层 I/O 操作进行管理, 可以将其视为一种内核模块. 由于驱动代码本身可能存在缺陷和漏洞, 设备驱动代码开发人员通过编译检查很难排查到一些特定条件下才会触发的代码错误[2]. 2011 年, Chen 等人[3]对 Linux 内核中的漏洞做了具体的分析归纳, 主要问题表现为内核接口误用、缓冲区溢出、空指针及指针错误、竞争与死锁、内存资源操作不当等. 其中内存资源操作类漏洞作为其中的一种主要安全威胁, 严重时甚至可直接造成系统崩溃.

设备驱动程序一般会调用特定的内核接口函数来申请和释放资源, 我们可称之为配对函数[4], 这种将资源操作类函数以成对形式进行提取的概念最早来自于 Engler 等人提出的一种名为 ECC[5]的方法, ECC 可以提取源代码中的配对函数信息, 并相应地提取潜在地正常执行路径上的路径规则. 目前对于这类函数的文档描述极少, 但由于其涉及到内存资源的相关操作, 一旦出现资源操作的错误, 将会危及整个操作系统的安全. 因此对于这类函数的相关检测和处理至关重要.

现有的主流 Linux 设备驱动程序分析方法主要有动态分析、静态分析以及符号执行 3 种. 在如今的程序分析技术路线中, 现有的大多数漏洞检测方法主要针对用户模式下的内核 API 调用规则和资源检测[6], 很多方法并不能很好地应用于工作在内核模式下的设备驱动程序上. 在程序的安全缺陷检查方面, 动态分析通常需要在源码中结合程序插桩技术[7], 在程序执行过程中依据制定的验证规则, 对执行的中间结果进行分析. Zhou 等人开发的 SafeDrive[8]原型工具在编译驱动程序时, 根据内核开发人员的注释插入相应的检查规则, 然后在运行时验证程序的安全性和完整性. 动态分析不需要对源码进行系统地分析, 但也因此达不到较高的覆盖程度, 且一般情况动态分析下都依赖于真实的硬件, 这为实时检测带来很多难度. 静态分析则与动态分析的分析方向不同, 其满足在不运行程序的情况下, 通过各种词法、语法分析等分析技术来检测分析源程序的数据流或控制流. 由于设备驱动程序源码中存在较多的条件分支和循环语句, 静态分析可以满足全覆盖源码的条件, 不依赖于真实的硬件设备, 无需考虑很多执行过程中的限制因素, 但是 Linux 设备驱动的代码结构十分庞大, 去分析其中的源码结构和库函数也是一件费时费力的事. 符号执行[9]的分析方法则是用符号值替代具体的程序变量, 并对所有可能的执行路径使用约束求解技术[10]生成特定的路径约束条件, 得到符合条件的程序执行路径. Cadar 等人开发的轻量级符号执行工具 KLEE[11]可以基于 LLVM[12]下的中间语言并对其进行符号执行, 通过程序执行状态的变化模拟真实程序的执行情况. 路径爆炸是符号执行分析技术中产生的一个最主要的问题, 伴随着符号执行分析技术的发展, 如何更有效地减少路径分支的指数增长情况, 避免产生路径爆炸的问题正成为研究热门.

以上 3 类设备驱动程序的分析方法都不可能 100% 覆盖所有潜在的安全漏洞. 目前的设备驱动资源漏洞检测工作大多基于编译后的中间结果, 耦合度和复杂程度高, 获取的信息不全面, 给漏洞分析检测任务带来许多挑战, 并且基于编译过程中的中间结果有时不能直观反映出设备驱动程序各函数的执行情况. 在本文的工作中, 我们提出了基于配对函数调用场景的设备驱动漏洞检测方法并设计实现 PFED (Pair Function Extraction and Detection) 原型, 首先是提取配对函数时优化配对的规则, 自动化提取结合手工分析验证; 随后, 在记录配对函数各项参数和调用信息的基础上, 通过获取驱动函数调用场景信息以更新配对函数在驱动函数执行路径中调用的记录; 最终, 结合调用场景验证并检测可能存在的内存资源的申请和释放不匹配问题.

本文的主要贡献有: 预处理源码并优化提取资源操作类配对函数的结果, 在驱动程序配对函数的互斥语义上做了更进一步的扩展与验证; 提出构建配对函数调用场景的概念, 用于完整记录配对函数在执行路径上的对应关系, 对源码的覆盖程度更高; 设计并实现了基于配对函数调用场景的设备驱动漏洞检测原型, 实验结果表明, 该方法更适用于对设备驱动程序的分析, 同时进一步降低了漏报率、误报率.

1 相关工作

1.1 静态分析

目前大部分的研究工作主要集中在检测设备驱动

© 中国科学院软件研究所 http://www.c-s-a.org.cn

漏洞的缓冲区溢出、整数溢出等这些溢出问题上, kint[13]是一个用于检测 C 语言源程序中出现的整数溢出错误的静态分析工具, 图 1 是其原型架构设计的流程图, 通过对基于 LLVM 形成的中间语言和注释信息的分析、约束求解过程来生成最终的错误报告, 可用于分析 Linux 内核, 协助内核开发工作人员检测程序中出现的整数溢出错误. 由于经历了中间语言的重写和各项元数据集中处理的复杂过程, 最后获取的生成约束表达式并不十分准确, kint 分析得到的整数溢出漏洞结果中存在着较高的误报率. 除此以外, 在设备驱动的资源操作类漏洞方面, 目前开展的研究工作主要是针对内存资源泄露的检测.

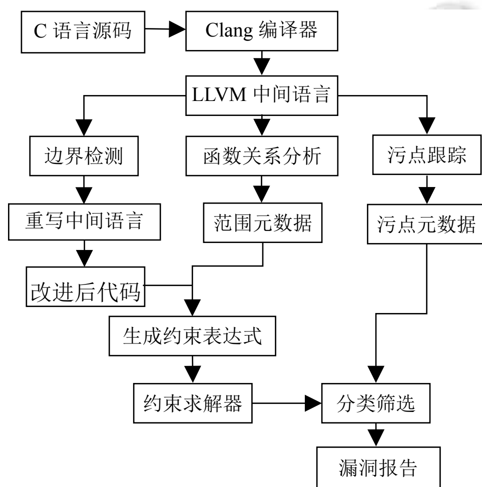


图 1 Kint 原型架构设计

1.2 运行时分析

PairDyn[14]是由 Bai 等人提出的一种运行时分析检测方法, 用来检测设备驱动程序中的资源申请和释放的匹配.

图 2 是 PairDyn 的架构设计图, 在驱动程序运行时, PairDyn 根据插入的探针记录下运行时的关键信息, 如参数返回值、函数调用信息, 再将这些信息与手动筛选得到的驱动程序配对函数记录列表相匹配, 获取相应的测试用例, 测试对于内存资源的申请是否有相应的释放函数调用与其对应. PairDyn 是通过人工方式选择配对函数的, 而人工方式可能会出现一些误判和遗漏; 只能在运行时检查正常情况下的执行路径, 因此其不能覆盖程序中的异常处理路径以及条件分支路径.

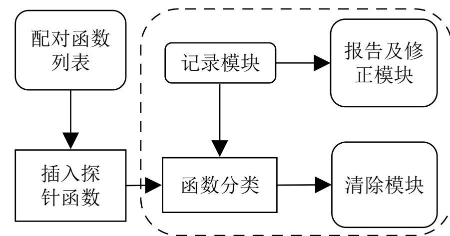


图 2 PairDyn 架构设计

2 设备驱动漏洞检测原型的构建

本文的主要目标是先从设备驱动源程序中提取出函数调用完整信息, 优化迭代并最终提取出真正的资源操作相关配对函数, 记录这些函数的调用上下文场景, 包括函数的主调函数、被调函数、参数返回值等有效信息. 在全覆盖驱动源码分析的基础上, 获取各对配对函数在驱动源程序的执行路径分支层级上的调用关系是否匹配和对称. 图 3 展示了 PFED 原型系统的设计, 接下来将对该方法涉及到的一些重要部分进行介绍说明.

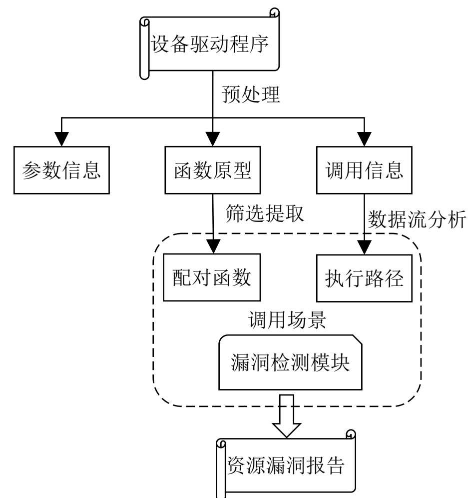


图 3 原型系统设计

3 设备驱动漏洞检测原型的设计与实现

为实现 PFED 原型, 首先要了解声卡、网卡、USB 等常用设备驱动的工作流程、功能模块以及主要程序结构. 根据几个不同类型的典型设备驱动分析其源程序结构, 由于设备驱动的源程序代码结构较为复杂, 代码量庞大, 为了更高效将驱动源程序中有用的函数信息提取出来, 首先我们要预处理源程序中的部分信息, 在源程序中略去大量注释、无用的条件预编译语句、

© 中国科学院软件研究所 http://www.c-s-a.org.cn

全局变量定义、结构体定义、宏定义及调用等冗余信息. 在分析设备驱动程序对内核函数的依赖接口时, 主要筛选出频繁与内核函数交互的设备驱动函数定义进行分析, 首先手工分析总结出涉及到内存资源操作的一些内核函数并记录, 其次开始着手自动提取驱动源程序中涉及到这些内核函数的所有函数原型列表、相关参数、调用关系、调用层级、执行路径等重要相关信息, 综合得出调用上下文场景.

整个原型的构建及实现过程简要归纳说明如下:

1) 分析内核模块的依赖接口, 来确定重点提取的内核函数列表;

2) 预处理驱动源程序, 略去大量无用信息, 提取重点函数原型列表;

3) 设计并自动化提取内存资源相关操作的配对函数, 手工验证总结、优化纠错;

4) 创建配对函数在调用层级、调用关系、调用路径等信息的调用上下文场景;

5) 根据配对函数原型列表及调用上下文场景验证内存资源的申请释放是否严格按层级性匹配.

3.1 配对函数提取

3.1.1 函数原型列表

本文使用 C 语言编写设备驱动源程序预处理模块, 使后续需要进行深入分析的驱动函数提取工作变的更轻量级. 主要设计的存放函数原型列表及相关信息的数据结构如下:

```
/* 为存放提取出的简要函数原型所定义的数据结构 */
struct FunctionPrototype
{
char mFunName[ID_MAX_LEN];
char mFunType[ID_MAX_LEN];
int paramCount;
struct FormalParam mParam[PARAMETER_COUNT_MAX];
};
struct Function
{
int mFunctionIndex;
struct FunctionPrototype mFunctionPrototype;
int mPreCompilingLevel;
struct PreCondition mPrecompilingCondition[PRECONDITION_COUNT_MAX];
int mStartLine;
int m_bDefined;
};
```

3.1.2 配对函数识别

如图 4 所示, 我们以 Linux4.8.8 版本内核下的 pcnet32 网卡驱动程序为例, 在 Linux 内核机制中, 每个网卡都由一个 net_device 结构来描述, pcnet32.c 中的 pcnet32_get_link() 函数是用来判断当前网络连接状态的驱动程序, 当执行单元访问共享资源之前, 需要用 729 行的 spin_lock_irqsave 来保存中断标志, 给中断当前的开启或关闭状态上锁, 相当于失效了当前的中断; 而 739 行的 spin_unlock_irqrestore 则是要恢复访问共享资源前的中断标志, 相当于释放掉自旋锁, 恢复到之前的中断状态. Linux 内核的中断机制中大量使用了自旋锁机制, 可以看出在该函数体内部 spin_lock_irqsave 和 spin_unlock_irqrestore 是先后分别被调用的, 它们的调用次序是固定的, 并且有上锁的操作则必须有解锁的操作.

```
724.static u32 pcnet32_get_link(struct net_device *dev)
725.{
726.    struct pcnet32_private *lp = netdev_priv(dev);
727.    unsigned long flags;
728.    spin_lock_irqsave(&lp->lock, flags);
        ……
739.    spin_unlock_irqrestore(&lp->lock, flags);
740.    return r;
741.}
```

图 4 Linux4.8.8 下 pcnet32 驱动部分代码段

本文主要在静态分析方法的基础上事先通过对设备驱动程序源码进行分析处理, 得到可能的资源操作配对函数信息. 图 5 显示了在 pcnet32 网卡驱动程序的不同函数定义体下相关资源操作的函数. pcnet32_probe1() 函数是加载和初始化的网卡驱动程序, 1695 行的 alloc_etherdev 是在该函数体中创建网络设备, 禁用网卡之后, 网卡程序则在 pcnet32_remove() 函数体内调用 2892 行的 free_netdev 删除已分配的网络设备. 这两个函数在对网络设备资源操作时呈现申请/释放的对应操作, 也即: 对内存资源申请之后必须相应地释放掉, 并且调用次序和主调函数都是固定和相对应的.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

```
1542. static int
1543.pcnet32_probe1(unsigned long ioaddr, int shared,
struct pci_dev *pdev)
1544.{
        ……
1695.   dev = alloc_etherdev(sizeof(*lp));
1696.   if (!dev) {
1697.       ret =-ENOMEM;
1698.       goto err_release_region;
1699.   }
        ……
1951. err_release_region:
1952.   release_region(ioaddr, PCNET32_TOTAL_S
IZE);
1953.   return ret;
1954.}
```

```
2880.static void pcnet32_remove (struct pci_dev *pdev)
2881.{
        ……
2892.   free_netdev(dev);
2983.   pci_disable_device(pdev);
2984.}
```

图 5　Linux4.8.8 下 pcnet32 驱动部分代码段

3.1.3　获取配对函数列表

Linux 内核开发人员对于内核函数的命名是十分规范的, 我们所归纳的这些配对函数的函数名是由规则的语义词、字符串及下划线组成的, 部分配对函数名称由一些加上前缀和后缀的字符串拼接组成, 每个字符串由下划线连接, 如图 5 所示, 整个函数名不仅仅有 release, 还有这个关键词前后的字符串 pci 和 device, 因此对于配对函数的识别就需要进行整个字符串语义匹配的综合判断. 针对预处理设备驱动源程序之后得到的函数列表, 对其进行手工分析总结得到一些涉及到内存资源操作, 且操作均为对资源的申请/释放的函数对, 表 1 列出了一些高频使用的配对函数部分关键词及其相关描述.

对于配对函数的关键词语义集合构建, 需要包含全部内核函数资源操作的关键词并归纳出每一对具有相反语义的关键词对. 我们给出配对函数的判定条件如下:

1) 对于资源进行操作的函数名满足一定的命名规则;

2) 对内存中相同的资源进行操作, 且操作的语义是相反的;

3) 成对地出现在一个完整的驱动程序执行场景中.

针对以上配对函数的判定条件, 提取配对函数需要建立两个存放不同语义的关键词集, 将手工分析得出的关键词分别放入两个关键词集中.

表 1　配对函数关键词描述

配对函数	描述
{alloc,free}	划分或回收内核空间内存
{open,close}	打开或关闭设备
{ lock,unlock}	对中断状态的上锁或解锁
{request,release}	申请或释放资源
{ioremap,iounmap}	为 I/O 内存分配或回收虚拟内存
{add,del}	添加或删除资源
{ enable,disable }	启用或禁用设备

图 6 给出了提取配对函数的算法.

```
1.begin
2:  create requestSet， releaseSet， pairSet;
3:  foreach keyword in list do
4:    if !MatchKeyWords (keyword, requestSet) then
5:      continue;
6:    end if
7:    while keyword_nt in list do
8:      if !MatchKeyWords(keyword, releaseSet) then
9:        continue;
10:     end if
11:     pairSet := ExtractPairFunc(keyword, keyword_nt);
12:     while pairKeyWords in pairSet do
13:       D:= PairCheck(keyword, keyword _nt);
14:       if StrMatch(func.name,func_nt.name) then
15:         D := PairCheck(func.name,func_nt.name)*M1;
16:       end if
17:       else if !StrMatch(func.name,func_nt.name)
18:         D := 0;
19:       end if
20:       else if IndexOfStrKMP(func.name, func_nt.name) then
21:         D := PairCheck(func.name,func_nt.name)*M2;
22:       end if
23:     end while
24:     if D > T then
25:       if !MatchFunc(func,func_nt, pairSet) then
26:         pairNum := pairNum+1;
27:         AddFunc(dependency, func, func_nt, pairSet);
28:       end if
29:     end if
30:   end while
31: end foreach
32:end
```

图 6　提取配对函数算法

根据以上的对配对函数的介绍和判定条件, 对配对函数的提取和匹配主要有图 6 中所描述的以下几个关键过程: 首先创建两个关键词集分别为内存资源申请关键词集 requestSet、内存资源释放关键词集 releaseSet, 以及一个配对函数词集 pairSet. 针对遍历预处理设备驱动源程序之后得到的函数原型列表文件, 扫描得到配对函数词集中的关键词记录, 以此来进一步发掘可能需要处理的函数名列表. 对于关键词记录表中每个关键字段, 将其分别和内存资源申请关键词集requestSet、

© 中国科学院软件研究所 http://www.c-s-a.org.cn

内存资源释放关键词集 releaseSet 相匹配, 以此判断它是否为资源操作相关函数的关键字段. 然后, 对每个存在配对可能性的资源申请函数 func 之后, 我们遍历检查找出与之对应的函数 func_nt. 如果 func_nt 是 func 的配对函数, 则它必须首先必须是一个资源释放函数, 并与函数 func 对相同的内存数据进行操作. 接着我们对资源释放函数 func_nt 以及资源申请函数 func 的匹配程度进行计算, 两个函数名配对的关联性决定了匹配度, 计算主要过程如下:

1) 初始的匹配度 D 设为内存资源申请/释放关键词集在关键词完全匹配的状态下的匹配度;

2) 若关键词所在函数名为单个字符串, 直接计算对应的匹配度, 匹配失败则匹配度 D 为 0;

3) 若关键词是多组字符串和下划线构成的函数名, 对关键词所在函数名的字段进行分割, 若都含有更多相同的子串则匹配度越高, 若子串不完全相同则匹配两个函数名的最长子串, 根据相应结果计算不同的匹配度 D.

根据每一对资源操作函数 func 与 func_nt, 匹配度 D 超过设置的阈值 D, 则配对成功, 添加到配对函数对当中; 匹配度小于设定阈值 D 的函数, 根据最后得到的配对程度报告文件, 对匹配程度低的资源操作函数对进行人工检查验证, 来确认其是否为真正的配对函数. 通过多次挖掘并修正结果, 优化子串的匹配过程, 找出真正对内存资源进行操作的配对函数, 可以为后续检查函数调用关系和路径工作减少大量不必要的工作量, 从而更高效全面地来检测设备驱动中内存资源操作的潜在违规行为: 如果当前的函数是配对函数词集 PairSet 里的一个资源申请函数 func, 搜索 pairSet 中是否有与其对应的、对相同数据进行操作的资源释放函数 func_nt, 如果未找到满足条件的函数, 此时极有可能出现内存资源的违规操作现象.

3.2 配对函数调用上下文场景

3.2.1 调用关系与调用层级

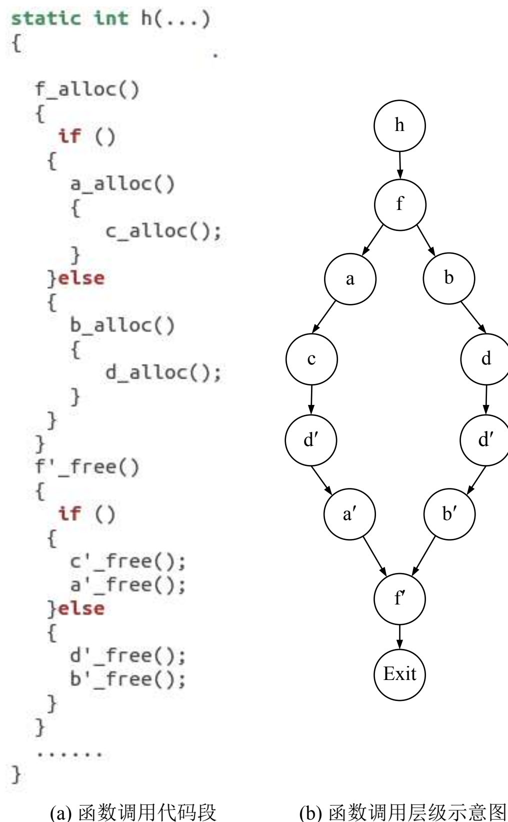
本阶段的任务主要是获取各驱动函数的调用情况、内存资源操作函数的调用关系以及调用层级, 建立并维护每个资源操作函数的在各个驱动函数体内部的调用状态列表以及跨驱动函数调用情况下的调用状态列表. 该阶段主要设计的函数调用原型列表及相关信息的数据结构如下:

```
/* 为存放提取出的真实函数调用原型所定义的数据结构 */
struct FunctionInvokation
{
int mIndexOfFunInvoker;
int paramCount;
char mActualParameter[PARAMETER_COUNT_MAX];
struct ContextOfInvokingFunction mContextOfInvokingFunction;
};
/*函数调用场景数据结构 */
struct ContextOfInvokingFunction
{
int mIndexOfFunctionInvoker;
int mContextLevel;
struct PreCondition mPreCondition[CIF_LEVELS_MAX];
};
/*函数调用预先条件数据结构 */
struct PreCondition
{
char mPreConditionDesrcription[PRECONDITION_MAX_LEN];
int mLineNo;
int mYes;
};
```

其中 ContextOfInvokingFunction 里 mContextLevel 为当前的调用层级, 也就是在源程序里的驱动函数体内或函数间的整个调用关系中, 按照操作的次序来给资源申请函数/资源释放函数编号, 并且在每个资源操作函数的状态列表中记录当前的条件分支. 如图 7(a) 的函数调用简略示意代码段所示, 在整个调用层级模式以及条件语句结构的基础上构建逻辑次序, a_alloc()~f_alloc() 表示资源申请函数, a'_free()~f'_free() 表示资源释放函数, 根据调用关系及其对称性, 可以大致归结出: 在每个完整的程序分支中, 配对函数的调用必须是成对的, 对于某一内存资源, 有申请必有释放; 在调用层级上, 每个资源操作函数列表及该函数下列表的调用关系是对称的, 因为对于内存资源满足先申请后释放的次序, 若驱动函数申请资源失败, 必须确保执行到最后可以把该资源申请操作之前的所申请的内存资源

© 中国科学院软件研究所 http://www.c-s-a.org.cn

按照对称的次序全部释放掉, 以确保没有资源操作的漏洞, 驱动源程序中的多重分支也是如此. 图 7(a) 代码段所对应的整个资源操作函数调用的次序如图 7(b) 所示.



(a) 函数调用代码段　　(b) 函数调用层级示意图

图 7　函数层级示例

在与内核进行交互时, 设备驱动程序可能会遇到突发的异常情况. 为了保证设备驱动代码的可靠性, 必须要提供处理这些突发情况的异常处理代码分支. 因此, 大多数设备驱动程序都会在这种情况下有对应的异常处理代码. 多数设备驱动程序基本都是用 C 语言编写的, 因此无法使用 C ++和 Java 中的一些异常处理或垃圾回收机制. 在设备驱动程序中最常用的异常处理机制就是基于 goto 语句的代码异常处理机制[15]. 在该机制中, goto 语句用于处理不同状况下的异常, 并且所有异常处理代码基本都位于每个设备驱动函数体内的末尾位置, 放置于每一个独立的代码段中. 例如图 5 中, 当 1695 行的申请以太网设备函数 alloc_etherdev() 返回异常时, 驱动程序将会跳转到 1951 行起始的 err_release_region 代码段.

在异常处理代码段中的每个资源释放函数都应该与该状态前的正常执行的资源申请函数形成层级性配对, 也即必须在异常处理部分逆序释放当前调用状态列表中所有已分配内存资源的申请函数.

在异常处理代码段中的每个资源释放函数都应该与该状态前的正常执行的资源申请函数形成层级性配对, 也即必须在异常处理部分逆序释放当前调用状态列表中所有已分配内存资源的申请函数.

3.2.2　建立执行路径

本阶段的任务是基于每个资源操作函数的调用状态列表建立完整的配对函数执行路径. 如果函数调用列表中的某个驱动函数体内没有调用其他驱动函数, 就只在该驱动函数体内建立配对函数的执行路径, 如果存在其他驱动函数, 则需要跨函数建立配对函数的整个执行路径.

针对单个驱动函数体内建立配对函数执行路径的情况, 首先要从调用状态列表取出当前资源申请函数调用起始位置所在的驱动函数, 也即其主调函数信息; 然后依据该驱动函数体内相应的最近层级配对函数调用状态列表提取出语义相反的资源释放函数, 综合二者调用状态列表中的调用层级 (驱动函数体内首个资源申请函数和最后一个资源释放函数, 对应层级为 0 层)、操作参数, 在执行过程中每多加载一个资源申请函数, 则对应层级加 1, 而资源释放函数则是每多加载一个则层级减 1. 最后分析完毕之后若形成完全对称性匹配, 就综合这些配对函数的状态形成一条资源申请与释放的完整路径信息, 整个调用路径记录对应着驱动源程序中的行号. 若遍历整个函数体内未找到对应层级的资源释放函数, 则形成不完整路径的报告.

在分析完单个驱动函数体内的执行路径之后, 则需要针对跨驱动函数的情况建立配对函数执行路径. 此时, 如图 7(b) 所示, 会有一个分析调用路径入口的主函数 *h*, 在该驱动主函数体内, 每当分析到出现对另外的驱动函数进行调用的情况, 此时要从函数调用状态列表中提取并记录调用的位置, 然后提取被调用驱动函数体中的配对函数列表, 载入该列表信息后, 再综合当前的主函数建立配对函数总的执行路径.

在得到了每个驱动函数体内的配对函数并确立了每个资源操作函数的对应层级、调用信息、相关参数之后, 对配对函数集合中的申请资源函数名进行遍历时, 从每个资源申请函数开始作为根节点, 子节点和叶节点分别定义为资源申请函数涉及的内存变量、执行路径终点的资源释放函数和相关参数. 从根节点开始

© 中国科学院软件研究所 http://www.c-s-a.org.cn

依次往后搜索寻找子节点、叶节点, 对于每个子节点, 重复进行向后搜索的流程直到抵达叶节点, 最终得出每条完整的执行路径集合及其所对应的资源操作集合. 因此每条完整的路径集合可以看做一棵包含参数信息的以顺序执行序列构成的结构化数据流树, 每棵由最外层资源申请函数作为数据流起点的树中包含了其内部所有以被调资源申请函数为起点的子树, 子树数量为 n. 每棵树的深度为 d, d 的值为满足以下条件的最小值:

$$2^d - 2 \geq n \tag{1}$$

将真实的执行路径集合占整个路径数量的比例 λ 进行统计, 由于实验的源码量大, λ 的值一般均不超过 20%, 并且 λ 越小, 子树数量 n 越小, 相应的平均执行时间越少. 因此在各运行路径上构造数据流树的平均执行时间都是比较少的, 基本上处于平稳增加的状态, 整个数据流树的构造过程不会造成树的深度 d 取值过大的问题.

3.2.3 内存资源操作违规性检测

上述 PFED 原型方法主要是建立在对设备驱动源码进行静态分析的基础上去实现的, 在提取驱动源程序中的配对函数之后, 再针对性的对预处理之后的源程序分析获取每个资源操作函数的调用上下文场景信息, 最后建立多条完整的配对函数执行路径. 在验证设备驱动内存资源申请释放的层级匹配过程中, 要注意的一点是, 同类设备驱动程序大多数情况下的执行逻辑相同, 例如网卡设备驱动的整个工作流程基本上遵循这样的流程: 探测、启动、发送和接受数据包、关闭、注销. 因此对于驱动函数体的整个检测顺序和每个驱动函数调用场景的建立也要遵循具体的运行逻辑.

通过对 Linux2.6.20 及 4.8.8 内核版本下的网卡、声卡、USB 等设备驱动源程序的分析处理, 我们在多次修正了子串的匹配过程及调整了匹配阈值 T 之后, 综合各驱动程序中的提取配对函数结果, PFED 分别可提取出共计 54 和 57 对由不同前缀或后缀字符、关键词以及下划线组成的配对函数, 然后通过人工检查验证, 分别确定了 49 和 52 对真正的配对函数.

在结合配对函数调用状态列表之后, 最后确立的不完整调用场景报告在整个函数执行场景报告中大约占据约 2.5% 的比例. 如图 8 所示的 Linux 下的 USB 设备驱动代码段示例中, 存在配对函数在调用层级上不匹配的异常情况: 在驱动函数 zd_op_start () 中的 332 行、334 行、336 行处的异常处理代码中, 分别调用了资源释放函数, 这三处释放函数严格按调用层级与前面的资源申请函数 zd_chip_enable_rxtx(), zd_chip_switch_radio_on(), zd_chip_enable_int() 相对应, 若申请资源失败, 则跳转到异常处理部分把前面调用的资源申请函数按照对称的次序释放掉. 但是在正常执行路径上, 与 zd_op_start() 所对应的驱动关闭函数 zd_op_stop() 中的 zd_chip_disable_rxtx(), zd_chip_disable_hwint() 这两处的释放函数的层级颠倒了, 由于先调用了 zd_chip_switch_radio_off 释放函数, 可能导致 zd_chip_disable_rxtx 释放函数引发内存资源操作不当的问题, 进一步造成死锁状态.

```
282. int zd_op_start(struct ieee80211_hw *hw)
283.{
     ……
294.   r = zd_chip_enable_int(chip);
     ……
315.   r = zd_chip_switch_radio_on(chip);
316.   if (r < 0) {
317.     dev_err(zd_chip_dev(chip), "%s: failed to
318.          set radio on\n", __func__);
319.     goto disable_int;
320.   }
321.   r = zd_chip_enable_rxtx(chip);
     ……
324.   r = zd_chip_enable_hwint(chip);
     ……
332. disable_rxtx:
333.   zd_chip_disable_rxtx(chip);
334. disable_radio:
335.   zd_chip_switch_radio_off(chip);
336. disable_int:
337.   zd_chip_disable_int(chip);
338. out:
339    return r;
340.}
342. void zd_op_stop(struct ieee80211_hw *hw)
343.{
      ……
356.   zd_chip_disable_rxtx(chip);
      ……
361.   zd_chip_disable_hwint(chip);
362.   zd_chip_switch_radio_off(chip);
363.   zd_chip_disable_int(chip);
      ……
368.}
```

图 8 Linux4.8.8 下 via-rhine 驱动部分代码段

4 实验结果与分析

4.1 实验方法

1) 实验环境: 硬件环境为 Intel(R) Core(TM) i7-4710MQ CPU @ 2.50 GHz, 8.00 GB 内存, 500 GB 硬盘; 软件环境为 Ubuntu 14.04 LTS 操作系统; 开发和编译工具为 Gedit、GCC4.8, C 语言.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

2) 实验样本: 本实验使用 Linux 内核版本为 4.8.8 下的六个网卡、声卡及 USB 驱动: pcnet32, ens1370, e100, sky2, zd_mac 以及 via-rhine.

针对设备驱动内存资源申请与释放相关操作的检测问题, 我们将本文提出的 PFED 原型工具在不同种类设备驱动程序上进行了测试比较, 实验展示了在 6 个不同的测试用例上最终分析得到结果与人工分析得到结果之间的误差, 进行准确性与可靠性评估. 其中对于提取配对函数实验的结果, 漏报率 (false negative) 指标是对比手工分析结果, 最终自动提取结果中未出现的配对函数对数占实际配对函数对数的比例; 误报率 (false positive) 指标是对比两份配对函数结果报告, 自动提取结果中含有手工分析报告未出现的配对函数对数占实际配对函数对数的比例.

4.2 实验结果及分析

通过在不同类型的设备驱动程序上提取到的配对函数对数与手工分析得到的真实结果进行对比, 以及对最后结果的漏报率、误报率进行统计, 来评估本方法的准确性.

针对预处理源程序提取的配对函数结果如表 2 所示, 按照对每个驱动函数体内不去重的方式记录提取结果. 由表 2 可以看出, 由于本文提出的方法是在对源码的高覆盖程度下进行分析, 并且多次纠正优化了函数名子串匹配过程, 通过人工验证检测结果, 最终结果的漏报率和误报率都比较低, 均不超过 15%.

表 2 提取配对函数结果

驱动	配对函数 (对)		漏报率 (%)	误报率 (%)
	手工分析	自动提取		
e100	37	35	5.41	5.41
pcnet32	41	38	9.76	13.63
ens1370	45	48	6.67	12.50
via-rhine	39	41	5.13	9.75
sky2	55	57	5.45	9.09
zd_mac	31	32	3.23	9.68

本方法的可靠性通过对不同版本下的驱动源程序的漏洞检测结果报告和人工验证真实的设备驱动资源操作漏洞结果进行对比评估, 结果如表 3 所示.

表 3 漏洞检测部分结果

资源申请函数	资源释放函数	所在路径	层级	检测结果
pci_alloc_consistent	pci_free_consistent	via-rhine.c/alloc_ring	0	缺少资源释放函数
pci_request_regions	pci_release_regions	via-rhine/rhine_init_one	1	缺少资源释放函数
pci_map_single	pci_unmap_single	pcnet32/pcnet32_realloc_rx_ring	3	缺少资源释放函数
pci_enable_device	pci_disable_device	sky2/sky2_probe	2	缺少资源释放函数
zd_chip_enable_hwint	zd_chip_disable_hwint	zd_mac/zd_op_stop	3	资源释放函数层级不匹配

实验检测到的跨驱动函数调用配对函数情况共 22 例, 由于驱动程序特有的结构性基础, 在静态分析过程中涉及到的跨函数调用场景配对检查情况, 我们在此处花费了更多的人工验证时间. 基于我们提取出的全部配对函数列表, 以及建立的函数调用场景, 根据不同的调用层级和申请释放层级, 形成对应层级的资源操作漏洞检测结果. 表 3 中给出了部分检测报告, 其中对应层级表示的是该对资源操作函数处于函数体的哪一层. 最终总计得到了 6 处可能出现资源匹配问题的有关漏洞检测结果, 经过人工验证, 发现有 1 处误报, 1 处漏报. 因此该方法在满足对源程序的高覆盖度情况下, 可以有效地检测出潜在的设备驱动资源操作漏洞.

由于部分设备驱动函数的调用列表有时会不满足规范的资源操作逻辑流程、驱动函数出现的个别接口函数私有化命名现象、驱动程序设计开发时违背编码规范等问题, 会导致分析结果中出现误判. 一些设备驱动程序中用以获取当前设备状态的内核接口函数, 比如 netif_carrier_ok/netif_carrier_on/netif_carrier_off 这组用来判断网络通路是否为正常连接状态的接口函数, 网卡驱动会通过它们和内核中的网络子系统传递消息, 但是这些接口函数并未涉及到内存资源操作. 因此在检测过程中我们会剔除这些无法应用正常分析流程的特殊实例, 在关键词集中建立特殊名单机制并通过迭代检测结果更新名单以提高整个检测过程的效率, 降低误判率.

5 结论与展望

本文提出一种基于配对函数调用场景的设备驱动漏洞检测的研究方法及相关原型 PFED, 与现有的驱动漏洞检测方法相比, 我们的方法不依赖于编译后形成的中间语言和真实情境下的硬件设备, 在结合了设备驱动工作流程及相关调用函数信息的基础之后, 主要

© 中国科学院软件研究所 http://www.c-s-a.org.cn

针对提取出的配对函数在执行路径上的调用场景信息,增加了需要进行分析的信息量,并有效检测出配对函数在调用层级上不匹配的潜在驱动漏洞. 实验结果表明, PFED 提取的配对函数结果更为精确, 具有较低的漏报率、误报率, 在检测设备驱动资源操作漏洞方面具有较高的可靠性, 并且进一步提高了源程序的覆盖度和检测的准确度. 然而, 本文的方法也存在许多不足,对于配对函数筛选结果中匹配度低的函数需要人工检查验证, 今后的研究应考虑设置自动验证和纠错反馈机制, 将匹配度低的配对函数分类别处理, 使配对函数提取的整个过程实现完全自动化. 并且如何进一步地结合更多种类的设备驱动结构、将符号执行技术应用于函数调用场景中以提高检测效率, 也将成为未来的研究工作.

参考文献

1 Chou A, Yang JF, Chelf B, *et al*. An empirical study of operating systems errors. Proceedings of the 18th ACM Symposium on Operating Systems Principles. Banff, Alberta, Canada. 2001. 73–88.

2 Chen Y, Wu FG, Yu KL, *et al*. Instant bug testing service for linux kernel. Proceedings of the IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing. Zhangjiajie, China. 2013. 1860–1865.

3 Chen HG, Mao YD, Wang X, *et al*. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. Proceedings of the 2nd Asia-pacific Workshop on Systems. Shanghai, China. 2011. 5.

4 Liu HQ, Wang YP, Jiang LB, *et al*. PF-Miner: A new paired functions mining method for android kernel in error paths. Proceedings of the IEEE 38th Annual Computer Software and Applications Conference. Vasteras, Sweden. 2014. 33–42. [doi: 10.1109/COMPSAC.2014.10]

5 Engler D, Chen DY, Hallem S, *et al*. Bugs as deviant behavior: A general approach to inferring errors in systems code. Proceedings of the 18th ACM Symposium on Operating Systems Principles. Banff, Alberta, Canada. 2001. 57–72.

6 刘虎球, 白家驹, 王瑀屏. 一种面向内核接口的顺序依赖规则挖掘与违例检测方法. 计算机学报, 2015, 38(5): 1007–1019.

7 Chittimalli PK, Shah V. GEMS: A generic model based source code instrumentation framework. Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation. Montreal, QC, Canada. 2012. 909–914.

8 Zhou F, Condit J, Anderson Z, *et al*. SafeDrive: Safe and recoverable extensions using language-based techniques. Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Seattle, WA, USA. 2006. 45–60.

9 Artzi S, Dolby J, Tip F, *et al*. Fault localization for dynamic web applications. IEEE Transactions on Software Engineering, 2012, 38(2): 314–335. [doi: 10.1109/TSE.2011.76]

10 de Moura L, Bjørner N. Z3: An efficient SMT solver. Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Budapest, Hungary. 2008. 337−340. [doi: 10.1007/978-3-540-78800-3_24]

11 Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, CA, USA. 2008. 209–224.

12 Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. Proceedings of 2004 International Symposium on Code Generation and Optimization. San Jose, CA, USA. 2004. 75–86.

13 Wang X, Chen HG, Jia ZH, *et al*. Improving integer security for systems with KINT. Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, CA, USA. 2012. 163–177.

14 Bai JJ, Liu HQ, Wang YP, *et al*. Runtime checking for paired functions in device drivers. Proceedings of the 21st Asia-pacific Software Engineering Conference. Jeju, South Korea. 2014. 407–414. [doi: 10.1109/APSEC.2014.66]

15 Saha S, Lawall J, Muller G. An approach to improving the structure of error-handling code in the linux kernel. ACM SIGPLAN Notices, 2011, 46(5): 41–50. [doi: 10.1145/2016603]

© 中国科学院软件研究所 http://www.c-s-a.org.cn