基于LSM框架的文件安全管控技术调研

文件编号	UT-RD-SXXX	版本	1.0.0
编制日期	2021-9-13	密级	保密

版本变更记录

版本	修订说明	修订人	修订时间
0.0.1	创建	王现利	2021-08-16
0.0.2	增加整体框架说明	王现利 张亚	2021-08-16
0.0.3	增加策略解析匹配说明	王现利 张亚	2021-08-22
0.0.4	补充权限控制流程图	王现利	2021-08-26
0.0.5	根据二审建议修改实验验证章节	王现利	2021-09-13

目录

1 相关术语 **1**

2 问题 **2**

3 现状 **3**

4 技术方案 **4**

4.1 整体框架 4

4.2 关键技术点 4

4.2.1 策略解析 4

4.2.2 hook 管控 7

5 实验验证 **8**

5.1 模块兼容性 8

5.2 策略加载解析测试 9

5.3 文件夹防删除 9

5.4 读文件权限 11

6 小结 **14**

7 参考文献引用 **15**

1 相关术语

LSM: Linux Secrity Module, Linux 安全模块是 Linux 内核中用于支持各种计算机安全模型的框架。

DAC: Discretionary Access Control 自主访问控制。它是根据主体（如用户、进程或 I/O 设备等）的身份和他所属的组来限制对客体的访问。

MAC: Mandatory Access Control 强制访问控制。安全策略由安全策略管理员集中控制, 用户无权覆盖策略。

VFS： 是一个内核软件层，在具体的文件系统之上抽象的一层，用来处理与 Posix 文件系统相关的所有调用，表现为能够给各种文件系统提供一个通用的接口，使上层的应用程序能够使用通用的接口访问不同文件系统，同时也为不同文件系统的通信提供了媒介。

AppArmor: 管理应用对文件权限的模块

FileArmor: 目前内部研发中的针对文件安全的管控模块，后续描述 FileArmor 来代替将要实现的文件安全管控模块。

DFA： 确定性有限状态自动机

eHFA: extened Hybrid Finite Automata, 拓展混合有穷自动机，也就是经过改造的 DFA。

2 问题

在日常工作由于操作失误，使用者可能会误删除系统的重要文件，影响系统的正常运行。 对于一些商业机密，用户信息等重要数据，由于未进行用户权限的限制，发生信息泄露事件，造成严重的经济损失。

对于系统核心的文件，或者一些重要办公文档，如交易系统的配置文件，银行客户端要缓存的敏感信息，办公文档应用的私密文件等。往往还需要更细的管理粒度，需要设置仅允许该应用访问，禁止别的应用进行读写或删除，包括有 root 权限的应用。

目前系统里支持按照用户来设置访问权限，可以设置该目录或文件所属者的权限、所属组的权限以及其他人的权限，但是对于 root 用户来说，拥有对所有文件的访问权限，存在一定的安全隐患。

本次调研主要是为了实现如下目标：

1. 重要数据防删除。
2. 做到应用之间的数据隔离，限制不同的应用操作同一个文件的权限。

3 现状

现有方案一：使用 chattr 命令修改文件属性，定义文件的隐藏属性，那么该文件的拥有者和 root 用户也无权操作该文件，只能解除文件的隐藏属性。才可以写该文件。

图 1: chattr 实验过程

进行如下测试：新建的文件 test.c 可以写内容，使用 chattr 修改属性，进行写和删除测试，撤销 chattr 对文件属性修改。

标注 1： chattr 设置文件属性，达到控制删除目的，但是影响写权限执行。

标注 2： 取消 chattr 设置文件属性，写权限得到恢复。

存在问题：使用 chattr 修改文件属性，限制写的权限。对于一些需要修改权限，或者特权用户想要删除，都是无法进行操作，因为 chattr 无法区分用户、应用进行权限管理。

现有方案二：基于 LSM 的强制访问控制模块进行权限管控，比如 SELinux，AppArmor 等

存在问题：SELinux 规则配置复杂，不容易使用。AppArmor 是基于文件路径，限制应用的权限，禁止访问一个文件，需要每一个应用都进行配置。

结合目前现有方案和技术，考虑通过 LSM 框架来修改系统原有的处理逻辑，解决现有文件安全管控技术方案存在的问题。

4 技术方案

4.1 整体框架

本方案主要包含两个模块，FileArmor 内核模块和 FileArmor 应用层模块。

FileArmor 内核模块用于文件权限管控。

FileArmor 应用层模块用于提供应用开发者设置权限的接口。

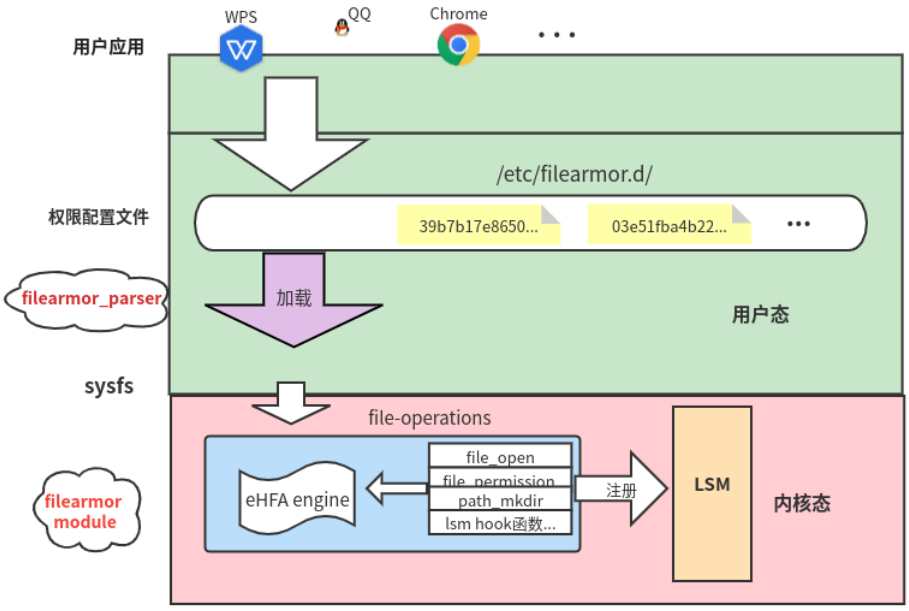

图 2: 整体框架结构图

用户态的作用是将文件的权限规则配置文件加载到内核，包含 filearmor_parser, libfilearmor, 启动服务等模块。

用户的权限规则配置文件保存在/etc/filearmor.d 文件夹。该配置文件一般随软件包安装，也可以由应用通过调用 libapparmor 库直接创建、修改。

filearmor_parser 是对配置文件解析、加载到 sysfs 文件系统的工具。

filearmor module 内核模块的功能是挂载 LSM 文件相关 hook 函数。读取 sysfs 中的配置数据，建立 eHFA 状态机引擎，供 hook 函数查询权限配置。

4.2 关键技术点

4.2.1 策略解析

FileArmor 内核运行时会针对管控规则进行策略检查，结果只有两个，匹配上或者没匹配上。

filearmor_parser 基于 flex 和 bison 语法词法解析，读取配置文件中文件路径正则表达式字符串，允许访问的程序和 uid、权限等数据。

图 3 所示为策略解析大致流程。加载权限配置文件以后，根据《编译原理（龙书）》中的转换规则，通过词法语法解析，用正则表达式字符串构建 DFA 表达树。随后根据 DFA 表达树，压缩状态，进行矩阵的拆解。转换得到 eHFA table，eHFA table 写入 sysfs 文件。

eHFA table 方便快速进行应用的路径匹配检查。未匹配到的路径，不进行管控。匹配到的路径，提取权限信息，进行校验，判断是否禁止申请的权限。

eHFA 同常规的数据结构主要的区别是 eHFA 可以支持正则表达式，并且对相同的数据，进行压缩处理。

当字符串的数量更大时，采用常规的数据结构进行规则校验，存配置数据没有压缩，冗余信息多，校验速度慢。占用内存也会比进行压缩 eHFA table 更多，因为 eHFA 做了压缩，降低内存开销，提高匹配速度。

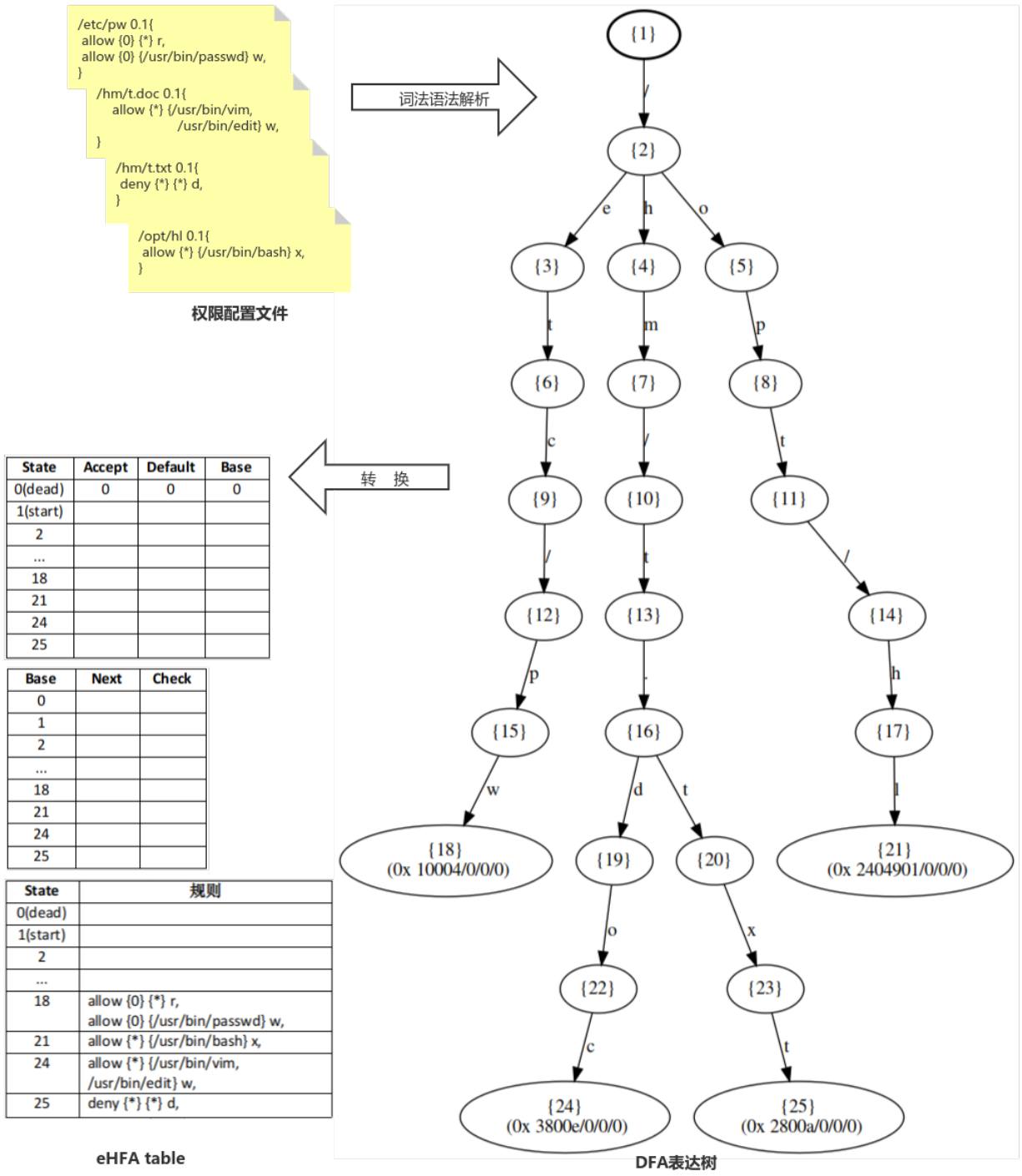

图 3: 策略解析流程

策略格式简介：

```
<filepath> <own app>[可选] <version>[可选] {      # 规则头
        <action> < uid list> <app list> <permission>,  # 规则，可以有多条
        <action> < uid list> <app list> <permission>,
        ...
}
```

<filepath> 文件绝对路径, 例如 hometest.sshrsa_key, 支持中文路径。

<own app> 归属应用的绝对路径, 可选项, 设置后只有该应用可以修改本配置文件, FileArmor 会限制本配置文件的访问应用。

<version> 版本号，可选项，例如 1.0，方便后期配置规则变更。

<action> allow 或 deny。allow 表示允许满足该条规则的操作，deny 表示拒绝满足该条规则的操作。

<uid list> 允许的用户 uid 列表。’ *’ 表示任何用户，用’’ 包裹，例如 0, 1000。

<app list> 允许的应用的绝对路径列表。‘*’ 表示任何应用功。用’’包裹,例如 /usr/bin/wps, /usr/bin/hello。

<permission> 权限， r|w|x|d，对应读、写、执行、删除权限。

注释 以 # 开头的内容表示被注释。

策略示例：

```
/home/test/test.doc /usr/bin/vim 1.0 {
        allow {*} {/usr/bin/cat}  r,
        allow {1000} {/usr/bin/wps,/usr/bin/office}  rw,
}

/home/test/.ssh/rsa_key {
        deny {1000} {/usr/bin/wps,/usr/bin/office} w,
}
```

4.2.2 hook 管控

进行文件的权限管控，最关键的是进行权限的校验。权限校验的过程基于 LSM 框架实现。

LSM 框架的设计初衷是为了在 Linux Kernel 中实现一个 MAC Hook 框架，LSM 已经作为 Linux Kernel 的一部分随着内核一起发布了。使用框架的好处就在于，安全人员可以直接基于 LSM 框架进行上层审计模块的开发，专注于业务逻辑，而不需要关注底层的内核差异兼容性和稳定性。

内核文件安全相关的关键对象有：inode(管道、文件或者 socket 套接字)、file(文件操作)、对这些对象的系统调用操作就是关键路径。LSM 在这些关键路径上，使用静态插桩法，插入了一批预置的 Hook 点。

当应用对文件发生读写删除等操作时，触发 filearmor module 注册的 LSM hook 函数，hook 函数中通过对文件的绝对路径进行 eHFA table 匹配，匹配不到说明该文件没有做权限限制，返回 0，操作被允许。匹配到说明该文件有权限配置，进一步获取该配置，同调用的进程路径和 uid 做进一步对比，判断操作是否被允许。

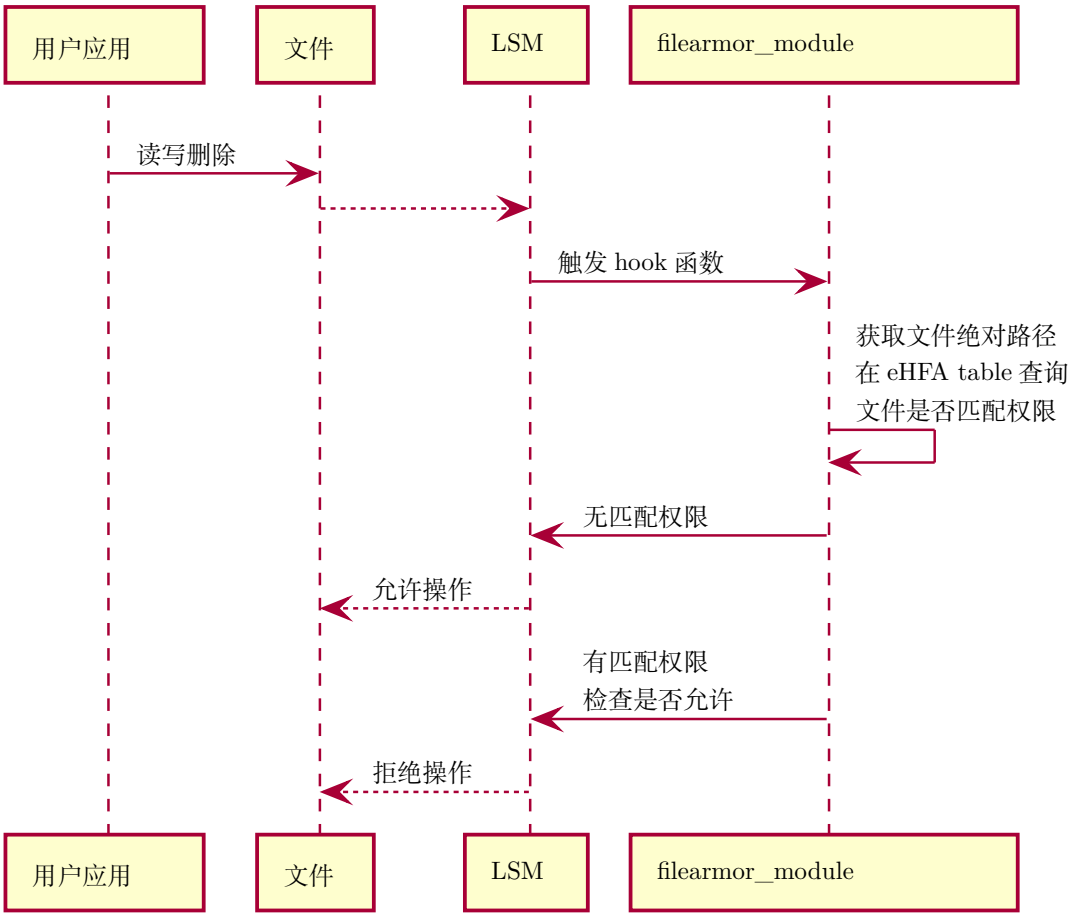

图 4: 文件操作权限的流程

表 1: 文件或文件夹操作与 hook 函数对应关系

文件操作	Hook 函数	钩子作用
重命名、拷贝	path_rename	检查重命名权限。
打开 创建	path_truncate	检查文件截取权限。
创建	file_open	检查打开文件权限。
删除	path_rmdir/inode_rmdir	检查删除文件夹权限。
创建	path_mknod/inode_mknod	检查创建文件权限。
创建	path_mkdir/inode_mkdir	检查创建文件夹权限。
读写执行删除	file_permission	检查文件修改权限。
删除	path_unlink/inode_unlink	检查删除文件硬链接或符号链接权限。

重复挂载的 hook 函数保存在链表中，按顺序执行，直到返回非 0 表示操作被拒绝，或所有挂载的 hook 函数均返回 0 表示操作被允许。也就是对文件的操作只有 FileArmor 的规则和其他安全模块的规则都满足时，才会被允许。

5 实验验证

5.1 模块兼容性

目前的安全模块之间存在冲突，无法同时运行，新增加文件安全管控模块实现与原有安全模块的兼容运行。

冲突原因：AppArmor、SELinux、Smack、TOMOYO 这几个默认模块会存在冲突，只能启动一个。安全模块 SELinux，AppArmor 等互斥的原因就是都使用了 LSM security 指针。

```
1 struct file 中的 void *f_security;
2 struct cred 中的 void *security;
3 struct inode 中的 void *i_security;
4 struct task_struct 中的 void *security;
```

实现兼容:

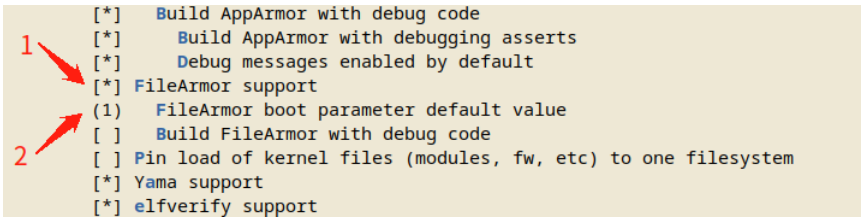

图 5: FileArmor 编译选项

FileArmor 与现存的 Linux security module 不冲突，可以兼容运行。FileArmor 编译选项和启动项独立于 Linux 安全模块。编译选项 ： SECURITY_FILEARMOR。如图 5 所示：

箭头标注 1 选择是否将 FileArmor 编译进内核。

箭头标注 2 FileArmor 内核 boot 启动项： 配置为 0 不启动 ； 配置为 1 启动。

进行功能验证，AppArmor 与 FileArmor 都可以正常启动，运行情况如图 6 所示，功能测试未出现正常。

```
[  134.558747] audit: type=1400 audit(1629781022.430:61): app
armor="ALLOWED" operation="open" namespace="root" profile="nm
bd" name="/proc/1508/cmdline" pid=2523 comm="nmbd" requested_
mask="r" denied_mask="r" fsuid=0 ouid=0
[  134.558812] audit: type=1400 audit(1629781022.430:62): app
armor="ALLOWED" operation="open" namespace="root" profile="nm
bd" name="/proc/1508/cmdline" pid=2523 comm="nmbd" requested_
mask="r" denied_mask="r" fsuid=0 ouid=0
```

图 6: AppArmor 运行日志

5.2 策略加载解析测试

策略解析参数说明：参数 D 转储内部信息以进行调试，通过 rule-exprs、expr-stats...、compressed-dfa 等参数，可以进行策略的加载解析过程分析，test 为构造的测试策略文件

```
sudo ./filearmor_parser  "-D" "rule-exprs" "-D" "expr-stats" "-D" "expr-tree"
↪  "-D" "expr-simplified" "-D" "stats" "-D" "progress" "-D" "dfa-states" "-D"
↪  "dfa-graph" "dfa-minimize" "-D" "dfa-unreachable" "-D" "dfa-node-map" "-D"
↪  "dfa-uniq-perms" "-D" "dfa-minimize-uniq-perms" "-D"
↪  "dfa-minimize-partitions" "-D" "compress-progress" "-D" "compressed-dfa"
↪  test
```

表 2: 参数简单说明

参数	作用
expr-tree	输出 dfa 表达树
expr-simplified	dfa 表达树简化结果
dfa-states	输出 DFA 状态下状态信息
rule-exprs	转储内部信息以进行调试
dfa-unreachable	状态到达情况
dfa-node-map	dfa 节点图信息
dfa-minimize-partitions	dfa 最小化分区
compressed-dfa	dfa 数据的压缩比例

```
DFA did not generate an equivalence class
Compressed trans table: states 54, next/check 285, optimal next/check 72 avg/state 5.28, compression 1464/27648
size=54 (accept, default, base):  {state} -> {default state}
0: (0, 0, 0) {0} -> {0}
1: (0, 0, 0) {1} -> {0}
2: (4, 0, 0) {2} -> {0}
```

图 7: 策略加载测试结果

通过进行策略的加载调试，很容易发现策略加载的过程是否有异常，以及异常出现的具体过程。从**红色箭头**标注可以看到，加载策略进行了数据压缩，压缩率为：1-1464/27648*100%=94.7%，大大降低了资源的消耗。

5.3 文件夹防删除

想要达到的禁止 uid 为 1000 用户，执行应用/usr/bin/rm，去删除/home/test/test 效果，设置策略格式如下：

```
/home/test/test {
        deny {1000} {/usr/bin/rm} d,
}
```

hook 函数检查逻辑实现：

```
        //          省略部分代码
        int check_inode_rmdir(struct inode* dir, struct dentry* dentry)
        {
                if ( unlikely(dentry == NULL)) {
                        pr_err("[%s] [%s] dentry is NULL!", MODULE_NAME,
                        ↪  __func__);
                        return 0;
                }

                char* kbuf = kmalloc(PATH_MAX, GFP_KERNEL);
                if ( unlikely(kbuf == NULL) ) {
                        pr_err("[%s] [%s] kmalloc fail!", MODULE_NAME,
                        ↪  __func__);
                        return -ENOMEM;
                }
                char* raw_path = dentry_path_raw(dentry, kbuf, PATH_MAX);
                if (IS_ERR(raw_path)) {
                        kfree(kbuf);
                        pr_err("[%s] [%s] failed to get dentry_path_raw",
                        ↪  MODULE_NAME, __func__);
                        return -ENOMEM;
                }

                char *app_path = executable_path(current);
                if( 0 != strcmp(app_path,
                ↪  "/usr/lib/systemd/systemd")){

                        printk("app_path = %s \n",app_path);
                        int process_owner_uid = (current->cred)->uid.val;
                        if(0 == strcmp(raw_path, "/home/wxl/test")){

                       // 检查用户 id, 用户 id 不是 1000 不控制
                        if( current_uid().val != 1000)  {
                                kfree(kbuf);
                                return 0;
                        }
                        printk(" [%s] uid = %d", __func__, current_uid().val);

                        if(0 == strcmp(app_path, "/usr/bin/rm")){
                                printk("check current uid = %d  can't delete
                                ↪  file \n", current_uid().val );
                                return -EPERM;
                        }
                return 0;
```

```
                }
        }
        return 0;

}
```

文件夹防删除验证结果：

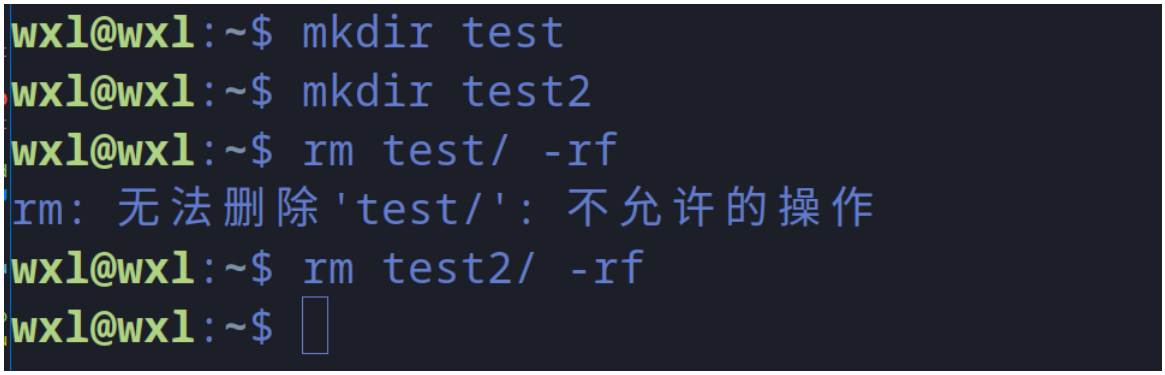

图 8: 文件夹防删除测试结果

当进行删除文件夹的操作时用户名、文件路径、应用，完全匹配，无法删除该文件，当用户名、文件路径、应用，未完全匹配，可以删除该文件夹。

5.4 读文件权限

想要达到仅允许 uid 为 1000 的用户，执行应用/usr/bin/more，去读/home/test/test.c 内容效果，设置策略格式如下：

```
/home/wxl/test.c {
        allow {1000} {/usr/bin/more} r,
}
```

读写权限管控的 hook 函数实现：

```
//  省略部分代码
int check_file_permission(struct file* file, int mask)
{

        char* kbuf;
        char* raw_path;

        if ( unlikely(file == NULL) ) {
                pr_err(" [%s] dentry is NULL!",  __func__);
                return 0;
                return 0
        }

        // 非路径返回 0
        kbuf = kmalloc(PATH_MAX, GFP_KERNEL);
```

```
        if ( unlikely(kbuf == NULL) ) {
                pr_err(" [%s] kmalloc failed!", __func__);
                return -ENOMEM;
        }

        raw_path = d_path( &(file->f_path), kbuf, PATH_MAX);
        if ( IS_ERR(raw_path) ) {
                kfree(kbuf);
                return 0;
        }

        // 测试路径下的文件进行权限控制
        if( 0 != strncmp( raw_path, "/home/wxl/test", strlen("/home/wxl/test")) )  {
                kfree(kbuf);
                return 0;
        }

        // 检查用户 id 用户 id不是 1000不允许访问
        if( current_uid().val != 1000)  {
                kfree(kbuf);
                return -EPERM;
        }
        printk(" [%s] uid = %d", __func__, current_uid().val);

        // test.c仅允许/usr/bin/more访问
        char* kbuf2 = kmalloc(PATH_MAX, GFP_KERNEL);
        char *app_path = executable_path(current);
        if( 0 == strcmp( app_path,  "/usr/bin/more") ) {
                kfree(kbuf);
                kfree(kbuf2);
                printk("check current uid = %d App = %s can't delete path = %s
                ↪  \n", process_owner_uid ,current->comm,raw_path);
                return 0;
        }

        kfree(kbuf);
        kfree(kbuf2);
        return 0;
}
```

不允许的应用操作结果：

```
cat test.c
cat: test.txt  Operation not permitted
```

允许的应用操作结果：

```
$ more test.c
hello world
```

基于管控实现原理，还可以继续扩展文件管控的其他功能，比如通过获取文件属性，进行实

现文件有效期管理，超出有效日期，限制文件打开操作。或者进行文件的清理，释放机器的存储资源。

6 小结

本方案可以有效的实现文件的防删除控制，对用户的重要文件提供可靠保护，做到针对用户应用的强制权限控制。实现应用的权限隔离，提高了文件的安全性。

本方策略配置简单，方便操作，相比目前的管控方案，更加容易使用。

本方案兼容性好，实现与现有安全方案的兼容运行，对原有的实现方案没有影响。

本方案采用的策略加载解析方式，通过测试验证，策略加载效率高，匹配速度快。

本方案可扩展性强，实现方式简单，结合具体的需求，可以实现不同的文件安全管控方案。

本方基于 LSM 框架实习，相比其他 hook 实现方式，因为编译进内核，安全级别更高，相比应用层 preload hook 方式，不会被绕过，管控更加严格，能够实现具体的应用权限的控制。

本方案管控模块匹配规则是直接在代码中硬编码，验证文件安全管控可行性。实现基本策略加载解析功能，测试策略加载的性能。

进行产品化的应用，还需要进一步的开发，将模块进行组合联调，优化实现细节。

7 参考文献引用

LSM [2] [3] [4] 策略解析实现 [1]

参考文献

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools*. 2020.

[2] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, pages 6–16. Citeseer, 2002.

[3] 刘瑜. Linux 安全分析与系统增强的研究. 电子科技大学学报, 2004.

[4] 杨宗德, 邓玉春, 曾庆华, et al. *Linux* 高级程序设计. 人民邮电出版社, 2008.