编号：_____

实验	一	二	三	四	五	六	七	八	总评	教师签名
成绩										

武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： 系统安全与可信计算

实验内容 ： 实验一 Linux 访问控制机制

专业(班) ： 网安试验班

学 号 ： 2023302181081

姓 名 ： 林水利

任课教师 ： 王鹃

2025 年 10 月 25 日

目 录

实验一　Linux 访问控制机制

1.1 实验名称

《Linux 访问控制机制》

1.2 实验目的

（1）熟悉 Linux 基本访问控制机制使用和原理

（2）熟悉 Linux S 位的作用和使用

（3）熟悉强制访问控制 Selinux 原理及其使用

1.3 实验步骤及内容

1、Linux 基本访问控制机制

（1）在/home 下创建一个文件夹 test

（2）在文件夹下创建一个文本文件

（3）利用两种方法将文本文件的权限设置为属主完全控制、组可读和执行、其它用户只读

（4）将系统新建文件和文件夹的权限设置为 644 和 755，重新创建文件观察权限值

（5）建立一个可执行文件，设置其 Suid 位；执行该文件，并通过命令观察该文件权限的变化。

（6）编译执行课件 suid 部分提权到 root shell 的例子，分析其原理

2、强制访问控制 Selinux 使用

（1）启动 Selinux

（2）查看系统的安全上下文

（3）安装 httpd

（4）修改 /var/www 文件夹的类型，使其为 httpd 不允许访问的类型，然后检查 web 服务器是否还可以正常使用。

3、Linux 访问控制机制实现原理分析

（1）安装 linux 源代码

（2）从源代码分析 linux 基本访问控制和 Selinux 实现原理

1.4 实验关键过程及其分析

1、Linux 基本访问控制机制

（1）在/home 下创建一个文件夹 test

（2）在文件夹下创建一个文本文件

（3）利用两种方法将文本文件的权限设置为属主完全控制、组可读和执行、其它用户只读

（4）将系统新建文件和文件夹的权限设置为 644 和 755，重新创建文件观察权限值

（5）建立一个可执行文件，设置其 Suid 位；执行该文件，并通过命令观察该文件权限的变化。

（6）编译执行课件 suid 部分提权到 root shell 的例子，分析其原理

2、关键过程

（1）利用两种方法将文本文件的权限设置为属主完全控制、组可读和执行、其它用户只读。

方法 1.使用字母的修改方法

修改前：如图 1，属主有读写权限，组和其他为只读权限。

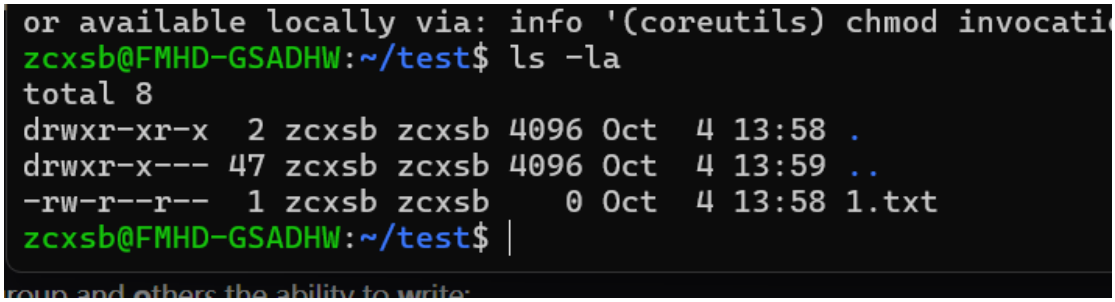


图 1 修改前 1.txt 的权限

修改命令以及修改如图 2：

chmod u+x 1.txt 是给拥有者添加 x 权限。因此拥有者对这个有了执行权限。且为完全控制权限（因为 rwx 都为 1 了）

chmod g+x 1.txt 是给拥有者对应的组添加 x 权限。此时显示为组可读和执行。而其他用户还是只读。

```
zcxsb@FMHD-GSADHW:~/test$ chmod u+x 1.txt
zcxsb@FMHD-GSADHW:~/test$ chmod g+x 1.txt
zcxsb@FMHD-GSADHW:~/test$ ls -la
total 8
drwxr-xr-x  2 zcxsb zcxsb 4096 Oct  4 13:58 .
drwxr-x--- 47 zcxsb zcxsb 4096 Oct  4 13:59 ..
-rwxr-xr--  1 zcxsb zcxsb    0 Oct  4 13:58 1.txt
zcxsb@FMHD-GSADHW:~/test$
```

图 2 修改后 1.txt 的权限

方法 2.使用数字的修改方法，如图 3

修改前：属主可读写，组和其他为只读。

```
zcxsb@FMHD-GSADHW:~/test$ ls -la
total 8
drwxr-xr-x  2 zcxsb zcxsb 4096 Oct  4 14:11 .
drwxr-x--- 47 zcxsb zcxsb 4096 Oct  4 13:59 ..
-rw-r--r--  1 zcxsb zcxsb    0 Oct  4 14:11 1.txt
zcxsb@FMHD-GSADHW:~/test$ chmod 754 1.txt
zcxsb@FMHD-GSADHW:~/test$ ls -la
total 8
drwxr-xr-x  2 zcxsb zcxsb 4096 Oct  4 14:11 .
drwxr-x--- 47 zcxsb zcxsb 4096 Oct  4 13:59 ..
-rwxr-xr--  1 zcxsb zcxsb    0 Oct  4 14:11 1.txt
```

图 3 使用数字的修改方法

修改后为 rwxr-xr--，即与第一种方法的结果一样。7=111，即一种权限用 3 位二进制数表示，第 i 位为 1 表示拥有这个位上的权限。结合后面的源码阅读我们知道其底层就是这样实现的。

（2）将系统新建文件和文件夹的权限设置为 644 和 755，重新创建文件观察权限值。

使用 umask 022 命令设置目录默认权限 755 和文件默认权限 644。

命令解释：

umask 会将系统默认创建权限减去设置的数字，系统创建文件默认权限为 666，文件夹默认 777，则减去 022 就是 644 和 755.

```
drwxr-xr-x  2 zcxsb zcxsb        4096 Oct  4 14:14 test
```

图 4 新创建的默认目录权限

文件 644（110 100 100）代表读写、读、读，755 代表完全控制、读和执行、

读和执行。

验证如图 4 和图 5。

```
drwxr-x--- 47 zcxsb zcxsb 4096 Oct  4 14:14 ..
-rw-r--r--  1 zcxsb zcxsb    0 Oct  4 14:14 1.txt
zcxsb@FMHD-GSADHW:~$ ls -la
```

图 5 新创建的默认文件权限

（3）建立一个可执行文件，设置其 Suid 位；执行该文件，并通过命令观察该文件权限的变化。

建立一个源文件，读取 rootfile，如图 6

```
#include <iostream>
using namespace std;
int main() {
        freopen("rootfile.txt","r",stdin);

        int x;
        cin >> x;

        cout << x << endl;

        return 0;
}
```

图 6 可执行文件的源代码

rootfile 权限如图 7，只允许 root 进行读写。

```
-rw-------  1 root  root     6 Oct  4 14:30 rootfile.txt
```

图 7 rootfile 的只允许属主读写的权限

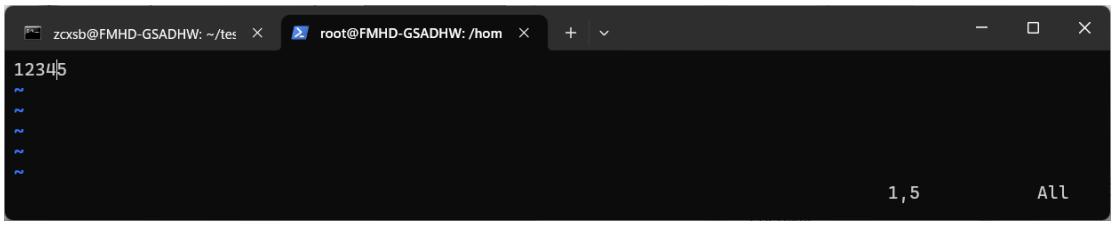


图 8 rootfile 的内容

如图 9、图 10、图 11，2.out 权限修改前，此时 root 执行该程序可以读出其中内容，而其他用户不能（即读不到数字则输出未初始化的变量值，为 UB）

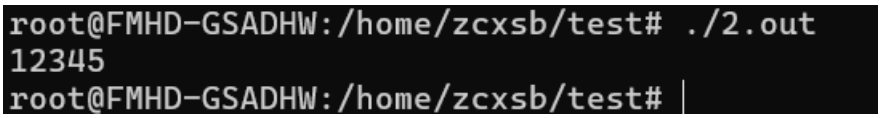


图 9 root 执行 2.out 正确输出

图 10 2.out 的权限

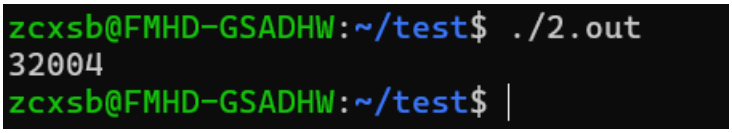


图 11 其他用户执行 2.out 不能正确输出

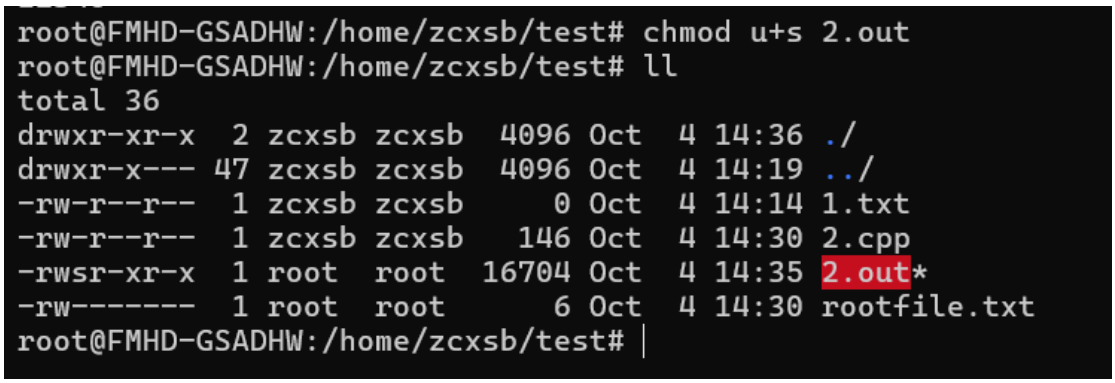


图 12 2.out 加了 S 位

如图 12 为 2.out 修改后的权限。加上 s 位，意思是能执行这个程序的用户将在执行时获得这个文件所有者的身份。那我们用其他用户验证一下。验证如图 13。

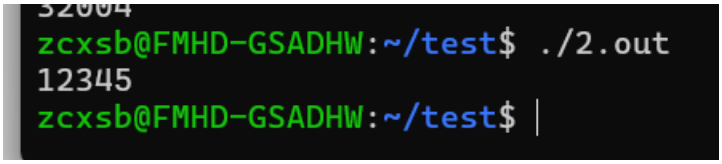


图 13 使用其他用户运行被加了 S 位的 2.out

此时该程序以 root 的权限对 rootfile 进行读取。

（4）编译执行课件 suid 部分提权到 root shell 的例子，分析其原理。

例子如 PPT（图 14）所示，该 shell 调用了 setuid(0)，即将当前用户的 uid 设置为 0（root），然后执行一个 shell。

- **提权到root shell**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
 char *name[2];
name[0]="/bin/sh";
name[1]=0x0;
setuid(0);
execve(name[0], name, 0x0);
 exit(0); }
```

注意：
1、以非root身份建立shell.c文件，修改shell，使其可执行
2、修改shell，使其属主为root
3、添加S位
此时当非root用户执行shell时刻获得root shell,拥有root权限

图 14 PPT 的 shell 例子

以非 root 身份创建 shell，修改 shell 使其可以执行，并修改 shell 属主为 root，并添加 s 位。按如上步骤操作后的 shell.out 如图 15。

图 15 shell.out 的权限

此时执行 shell 后，shell 的身份应该为 root。

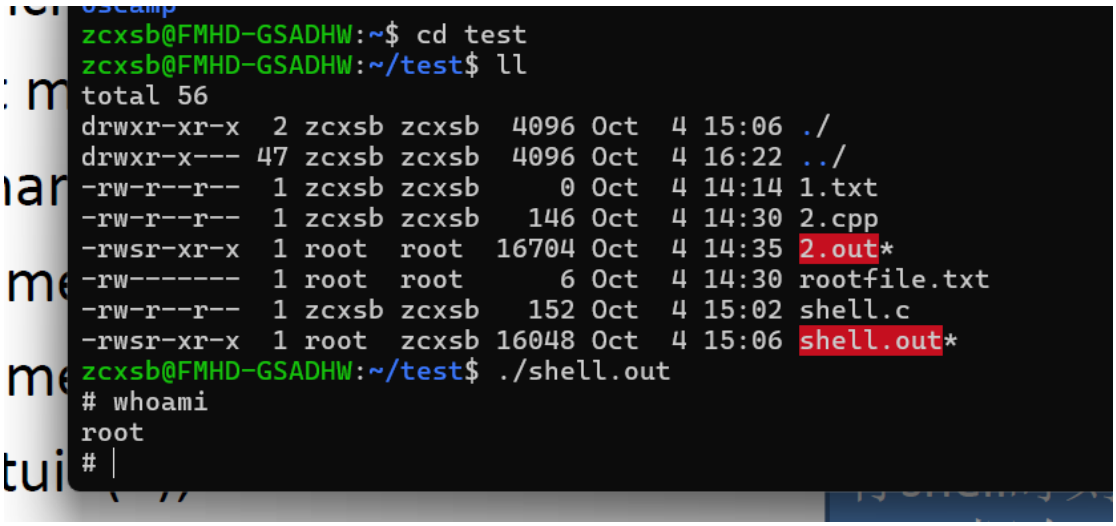


图 16 设置了 S 位，属主 root 的 shell 执行结果

分析：

普通用户执行这个 shell，因为 s 位置位，说明这个用户执行这个程序，这个程序的权限是属主 root 的权限，所以这个程序以 root 的身份打开了 sh，导致 whoami 为 root。

那么根据 setuid 的限制，即只有 root 或者设置了 S 位并且属主为 root 的程序可以成功设置 uid=0，那么我们修改这个属主为 zcxsb，看看会发生什么，如图 17。

```
zcxsb@FMHD-GSADHW:~/test$ sudo chown zcxsb ./shell.out
zcxsb@FMHD-GSADHW:~/test$ ll
total 56
drwxr-xr-x  2 zcxsb zcxsb  4096 Oct  4 15:06 ./
drwxr-x--- 47 zcxsb zcxsb  4096 Oct 25 12:44 ../
-rw-r--r--  1 zcxsb zcxsb     0 Oct  4 14:14 1.txt
-rw-r--r--  1 zcxsb zcxsb   146 Oct  4 14:30 2.cpp
-rwsr-xr-x  1 root  root  16704 Oct  4 14:35 2.out*
-rw-------  1 root  root      6 Oct  4 14:30 rootfile.txt
-rw-r--r--  1 zcxsb zcxsb   152 Oct  4 15:02 shell.c
-rwxr-xr-x  1 zcxsb zcxsb 16048 Oct  4 15:06 shell.out*
zcxsb@FMHD-GSADHW:~/test$ ./shell.out
$ whoami
zcxsb
$ ^[[2~ex
/bin/sh: 2: ex: not found
$ exit
zcxsb@FMHD-GSADHW:~/test$ chmod u+s shell.out
zcxsb@FMHD-GSADHW:~/test$ ./shell.out
$ whoami
zcxsb
$
```

图 17 属主为非 root，执行有 setuid(0)的程序的结果

在这里，我们进一步验证了 setuid 的用法。我们查看一下 setuid 的源代码，如图 17.2。

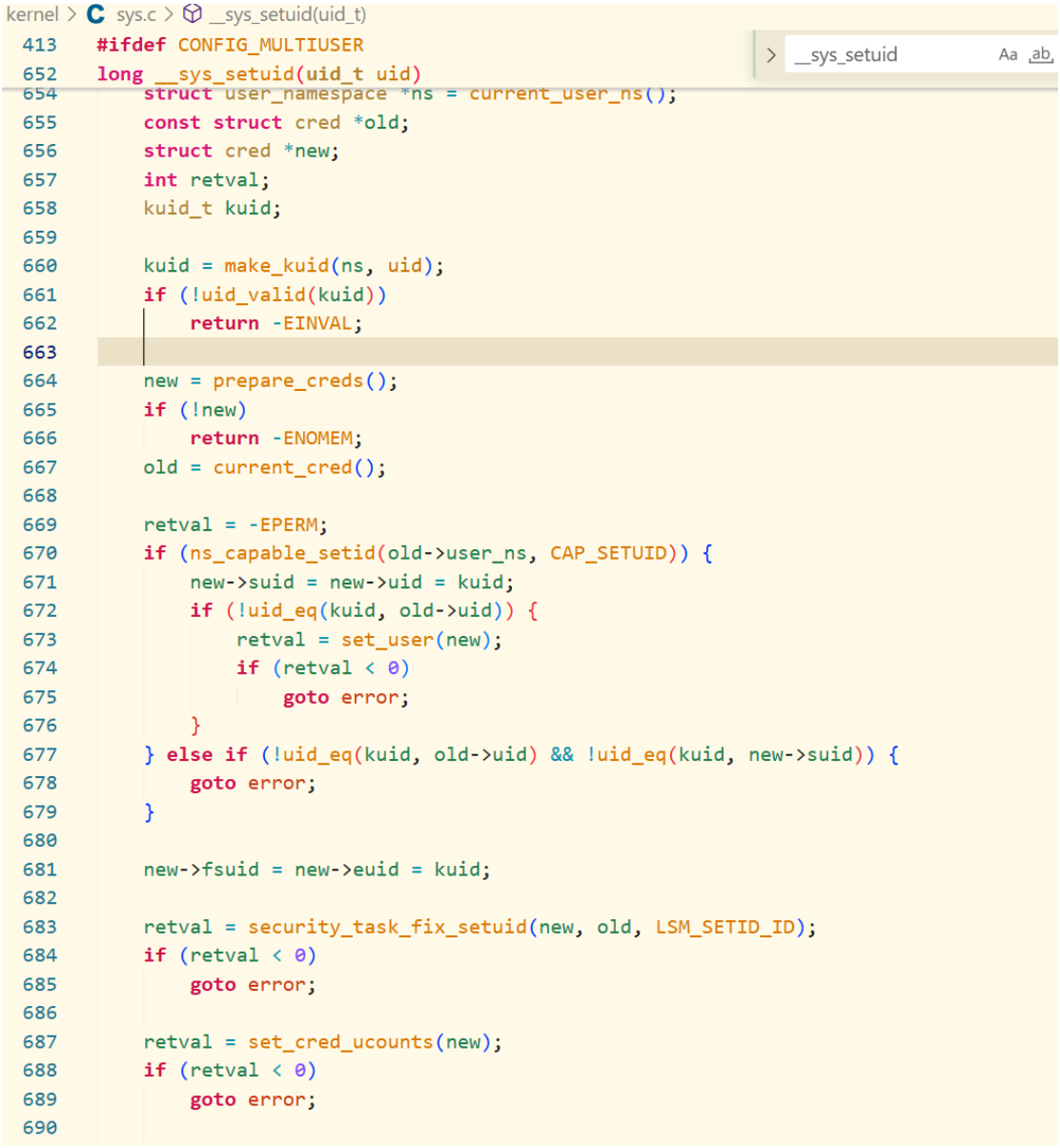


图 17.2 setuid 的视线

首先获取用户 namespace，然后检查设置的 uid 是否合法，然后准备 credential，并准备替换。其中检查 setuid 权限的 ns_capable_setid 的最底层函数为 cap_capable_helper，如图 17.3。

```
 * See cap_capable for more details.
 */
static inline int cap_capable_helper(const struct cred *cred,
                     struct user_namespace *target_ns,
                     const struct user_namespace *cred_ns,
                     int cap)
{
    struct user_namespace *ns = target_ns;

    /* See if cred has the capability in the target user namespace
     * by examining the target user namespace and all of the target
     * user namespace's parents.
     */
    for (;;) {
        /* Do we have the necessary capabilities? */
        if (likely(ns == cred_ns))
            return cap_raised(cred->cap_effective, cap) ? 0 : -EPERM;

        /*
         * If we're already at a lower level than we're looking for,
         * we're done searching.
         */
        if (ns->level <= cred_ns->level)
            return -EPERM;

        /*
         * The owner of the user namespace in the parent of the
         * user namespace has all caps.
         */
        if ((ns->parent == cred_ns) && uid_eq(ns->owner, cred->euid))
            return 0;

        /*
         * If you have a capability in a parent user ns, then you have
         * it over all children user namespaces as well.
         */
        ns = ns->parent;
    }
```

图 17.3 cap_capable_helper

从该函数中可以发现，ns->level <= cred_ns->level 也就是说目标 ns 的权限要高于当前 ns 的权限才能转换。

2、强制访问控制 Selinux 使用

（1）启动 Selinux

（2）查看系统的安全上下文

（3）安装 httpd

（4）修改 /var/www 文件夹的类型，使其为 httpd 不允许访问的类型，然后检查 web 服务器是否还可以正常使用。

关键步骤：

1. 在 Ubuntu 上安装 selinux

```
$ sudo apt install policycoreutils selinux-utils selinux-basics
$ sudo selinux-activate
$ sudo selinux-config-permissive
```

重启（因为可能没有配置某些必要程序的规则，所以如果设置为 enforcing，系统将无法启动，因此使用 permissive 进行实验，对 selinux 的行为观察 selinux 的日志即可）

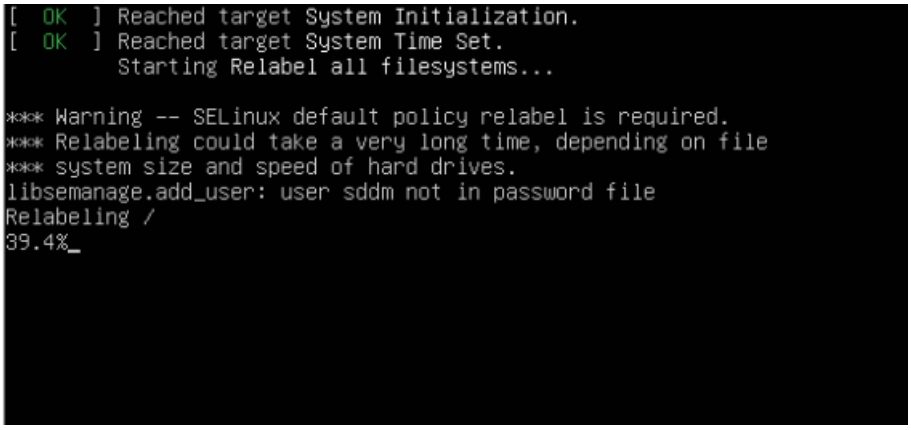


图 18 启用 Selinux 重启过程显示正在 relabel

2. 查看安全上下文

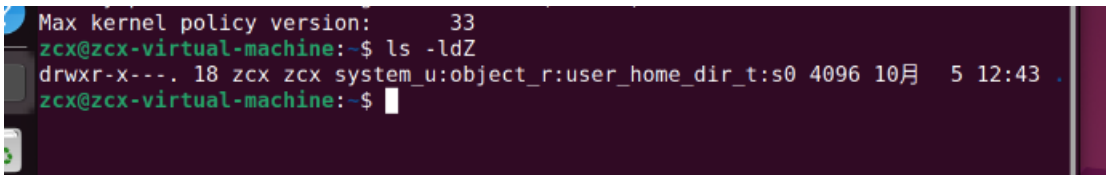


图 19 用户 home 的安全上下文

如图 19，可以发现增加了列条目，system_u 为 selinux 安全用户，object_r 是安全上下文的角色，说明这是一个对象，user_home_dir_t 为安全上下文类型，决定对这个文件的访问决策。

3. 安装 httpd 查看安全上下文。

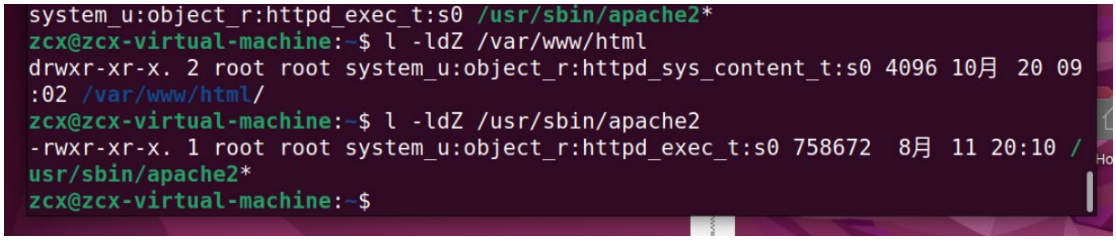


图 20 httpd 的相关目录默认安全上下文

如图 20，apache2 的二进制文件被赋予了 httpd_exec_t 的类型，其工作目录（即网站目录）被赋予 httpd_sys_content_t 的类型。这些类型将与访问控制相关。

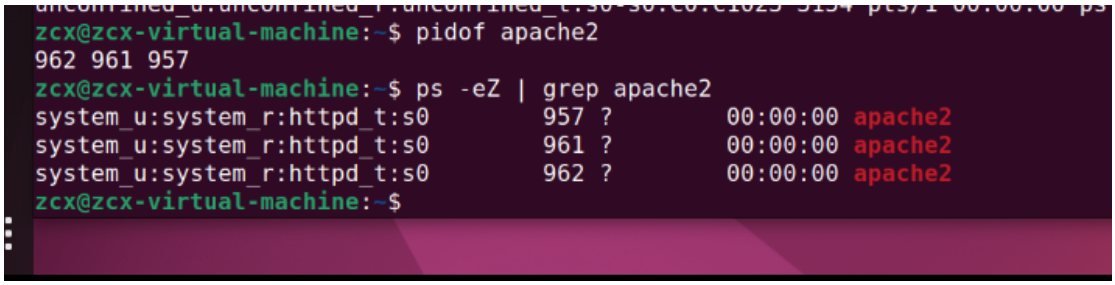


图 21 进程安全上下文

从图 21 我们可以发现，进程的安全上下文跟文件是不同的，我们用 sesearch -A -s httpd_t 查看一下这个类型的权限规则，如图 22。

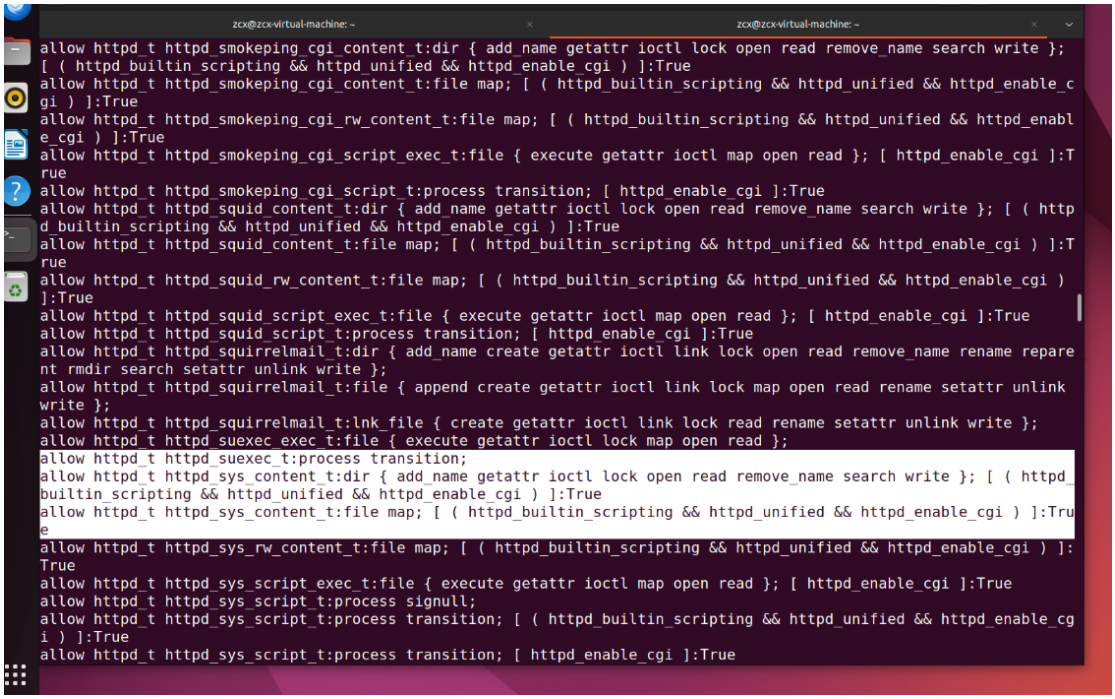


图 22 httpd 的 allow 规则

我们只看目标类型为 httpd_sys_content_t 的规则，可以发现对这个类型的

dir 具有很多权限，符合进程对网站的控制。后面为 filemap，是某种内存映射文件，我们在这里就不解释了。

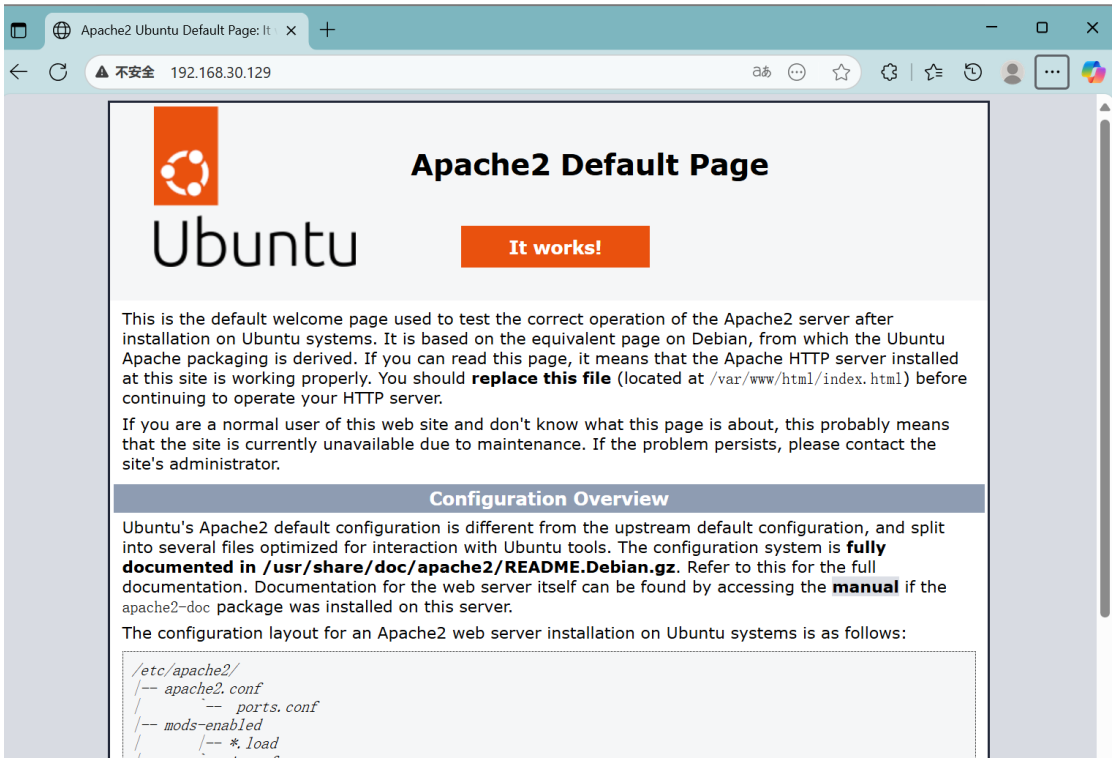


图 21 网页正常访问

而且根据撰写报告时的验证，其没有产生 AVC（即拒绝访问）的 log。

4. 修改 /var/www 文件夹的类型，使其为 httpd 不允许访问的类型，然后检查 web 服务器是否还可以正常使用。

如图 22，我们修改/var/www 为 default_t，该 type 是未分类文件的默认类型，httpd_t 没有对其允许访问的 allow 规则。chcon -R 表示递归修改，-t 为要修改到的目标类型。

```
zcx@zcx-virtual-machine:/var/log$ sudo chcon -R -t default_t /var/www
```

图 22 修改权限的命令

```
type=AVC msg=audit(1760924190.920:658): avc:  denied  { search } for  pid=3815 comm="apache2" name="www" dev="sda3" ino=661779 scontext=syste
m_u:system_r:httpd_t:s0 tcontext=system_u:object_r:default_t:s0 tclass=dir permissive=1
type=AVC msg=audit(1760924190.920:659): avc:  denied  { getattr } for  pid=3815 comm="apache2" path="/var/www/html" dev="sda3" ino=661780 sco
ntext=system_u:system_r:httpd_t:s0 tcontext=system_u:object_r:default_t:s0 tclass=dir permissive=1
type=AVC msg=audit(1760924190.920:660): avc:  denied  { getattr } for  pid=3815 comm="apache2" path="/var/www/html/index.html" dev="sda3" ino
=661782 scontext=system_u:system_r:httpd_t:s0 tcontext=unconfined_u:object_r:default_t:s0 tclass=file permissive=1
type=AVC msg=audit(1760924190.920:661): avc:  denied  { read } for  pid=3815 comm="apache2" name="index.html" dev="sda3" ino=661782 scontext=
system_u:system_r:httpd_t:s0 tcontext=unconfined_u:object_r:default_t:s0 tclass=file permissive=1
type=AVC msg=audit(1760924190.920:662): avc:  denied  { open } for  pid=3815 comm="apache2" path="/var/www/html/index.html" dev="sda3" ino=66
1782 scontext=system_u:system_r:httpd_t:s0 tcontext=unconfined_u:object_r:default_t:s0 tclass=file permissive=1
type=AVC msg=audit(1760924190.920:663): avc:  denied  { map } for  pid=3815 comm="apache2" path="/var/www/html/index.html" dev="sda3" ino=661
782 scontext=system_u:system_r:httpd_t:s0 tcontext=unconfined_u:object_r:default_t:s0 tclass=file permissive=1
zcx@zcx-virtual-machine:/var/log$
```

图 23 再次访问网站时 log 的记录

图 23 检测到 apache2 对 default_t 的访问，并记录在 log 中，denied 字样

说明如果在强制模式下，就不能访问。说明 selinux 检测到了越权访问，但是由于是 permissive 模式，因此只记录在日志中而没有真正阻止访问。

3、Linux 访问控制机制实现原理分析

（1）安装 linux 源代码

（2）从源代码分析 linux 基本访问控制和 Selinux 实现原理

关键步骤：

1.基础访问控制：

include/linux/fs.h

```
/*
 * Keep mostly read-only and often accessed (especially for
 * the RCU path lookup and 'stat' data) fields at the beginning
 * of the 'struct inode'
 */
struct inode {
    umode_t         i_mode;
    unsigned short      i_opflags;
    kuid_t          i_uid;
    kgid_t          i_gid;
    unsigned int        i_flags;
```

图 24 inode 数据结构关键字段

如图 24，inode 结点数据结构中有 i_mode 字段，存储文件的类型和三类用户（所有者、组、其他）的权限 rwx，以及 uid、gid。权限检查主要用 i_mode 跟 i_uid,i_gid 结合检查。

```
98              ssize_t bytes, void *private);
99
100 #define MAY_EXEC        0x00000001
101 #define MAY_WRITE       0x00000002
102 #define MAY_READ        0x00000004
103 #define MAY_APPEND      0x00000008
104 #define MAY_ACCESS      0x00000010
105 #define MAY_OPEN        0x00000020
106 #define MAY_CHDIR       0x00000040
107 /* called from RCU mode, don't block */
108 #define MAY_NOT_BLOCK       0x00000080
109
```

图 25 mode 的掩码

fs/inode.c

```c
bool inode_owner_or_capable(struct mnt_idmap *idmap,
                const struct inode *inode)
{
    vfsuid_t vfsuid;
    struct user_namespace *ns;

    vfsuid = i_uid_into_vfsuid(idmap, inode);
    if (vfsuid_eq_kuid(vfsuid, current_fsuid()))
        return true;

    ns = current_user_ns();
    if (vfsuid_has_mapping(ns, vfsuid) && ns_capable(ns, CAP_FOWNER))
        return true;
    return false;
}
EXPORT_SYMBOL(inode_owner_or_capable);
```

图 26 判断一个进程是否可以被视为文件所有者。

```c
/**
 * in_group_or_capable - check whether caller is CAP_FSETID privileged
 * @idmap:  idmap of the mount @inode was found from
 * @inode:  inode to check
 * @vfsgid: the new/current vfsgid of @inode
 *
 * Check whether @vfsgid is in the caller's group list or if the caller is
 * privileged with CAP_FSETID over @inode. This can be used to determine
 * whether the setgid bit can be kept or must be dropped.
 *
 * Return: true if the caller is sufficiently privileged, false if not.
 */
bool in_group_or_capable(struct mnt_idmap *idmap,
             const struct inode *inode, vfsgid_t vfsgid)
{
    if (vfsgid_in_group_p(vfsgid))
        return true;
    if (capable_wrt_inode_uidgid(idmap, inode, CAP_FSETID))
        return true;
    return false;
}
EXPORT_SYMBOL(in_group_or_capable);
```

图 27 对应的组判断。

```
 * Return: the new mode to use for the file
 */
umode_t mode_strip_sgid(struct mnt_idmap *idmap,
                        const struct inode *dir, umode_t mode)
{
    if ((mode & (S_ISGID | S_IXGRP)) != (S_ISGID | S_IXGRP))
        return mode;
    if (S_ISDIR(mode) || !dir || !(dir->i_mode & S_ISGID))
        return mode;
    if (in_group_or_capable(idmap, dir, i_gid_into_vfsgid(idmap, dir)))
        return mode;
    return mode & ~S_ISGID;
}
EXPORT_SYMBOL(mode_strip_sgid);

#ifdef CONFIG_DEBUG_VFS
/*
```

图 28 S 位的转译

```
static inline bool execute_ok(struct inode *inode)
{
    return (inode->i_mode & S_IXUGO) || S_ISDIR(inode->i_mode);
}
```

图 29 判断能否执行

```
void inode_set_flags(struct inode *inode, unsigned int flags,
                     unsigned int mask)
{
    WARN_ON_ONCE(flags & ~mask);
    set_mask_bits(&inode->i_flags, mask, flags);
}
```

图 30 设置标志位

我们以 open 为例，分析一下其安全机制。

如图 31，do_dentry_open 中，包含对 security_file_open 的调用

```
static int do_dentry_open(struct file *f,
                          int (*open)(struct inode *, struct file *))
{
    static const struct file_operations empty_fops = {};
    struct inode *inode = f->f_path.dentry->d_inode;
    int error;

    path_get(&f->f_path);
    f->f_inode = inode;
    f->f_mapping = inode->i_mapping;
    f->f_wb_err = filemap_sample_wb_err(f->f_mapping);
    f->f_sb_err = file_sample_sb_err(f);

    if (unlikely(f->f_flags & O_PATH)) {
        f->f_mode = FMODE_PATH | FMODE_OPENED;
        file_set_fsnotify_mode(f, FMODE_NONOTIFY);
        f->f_op = &empty_fops;
        return 0;
    }

    if ((f->f_mode & (FMODE_READ | FMODE_WRITE)) == FMODE_READ) {
        i_readcount_inc(inode);
    } else if (f->f_mode & FMODE_WRITE && !special_file(inode->i_mode)) {
        error = file_get_write_access(f);
        if (unlikely(error))
            goto cleanup_file;
        f->f_mode |= FMODE_WRITER;
    }

    /* POSIX.1-2008/SUSv4 Section XSI 2.9.7 */
    if (S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode))
        f->f_mode |= FMODE_ATOMIC_POS;

    f->f_op = fops_get(inode->i_fop);
    if (WARN_ON(!f->f_op)) {
        error = -ENODEV;
        goto cleanup_all;
    }

    error = security_file_open(f);
    if (error)
        goto cleanup_all;
```

图 31 do_entry_open

namei.c 包含 permissioncheck 的核心函数，其他文件系统在访问的时候调用这个进行 check。

```
 */
int generic_permission(struct mnt_idmap *idmap, struct inode *inode,
                       int mask)
{
    int ret;

    /*
     * Do the basic permission checks.
     */
    ret = acl_permission_check(idmap, inode, mask);
    if (ret != -EACCES)
        return ret;

    if (S_ISDIR(inode->i_mode)) {
        /* DACs are overridable for directories */
        if (!(mask & MAY_WRITE))
            if (capable_wrt_inode_uidgid(idmap, inode,
                                         CAP_DAC_READ_SEARCH))
                return 0;
        if (capable_wrt_inode_uidgid(idmap, inode,
                                     CAP_DAC_OVERRIDE))
            return 0;
        return -EACCES;
    }

    /*
     * Searching includes executable on directories, else just read.
     */
    mask &= MAY_READ | MAY_WRITE | MAY_EXEC;
    if (mask == MAY_READ)
        if (capable_wrt_inode_uidgid(idmap, inode,
                                     CAP_DAC_READ_SEARCH))
            return 0;
    /*
     * Read/write DACs are always overridable.
     * Executable DACs are overridable when there is
     * at least one exec bit set.
     */
    if (!(mask & MAY_EXEC) || (inode->i_mode & S_IXUGO))
        if (capable_wrt_inode_uidgid(idmap, inode,
                                     CAP_DAC_OVERRIDE))
            return 0;

    return -EACCES;
}
EXPORT_SYMBOL(generic_permission);

/**
```

图 32 内核导出的权限检查接口

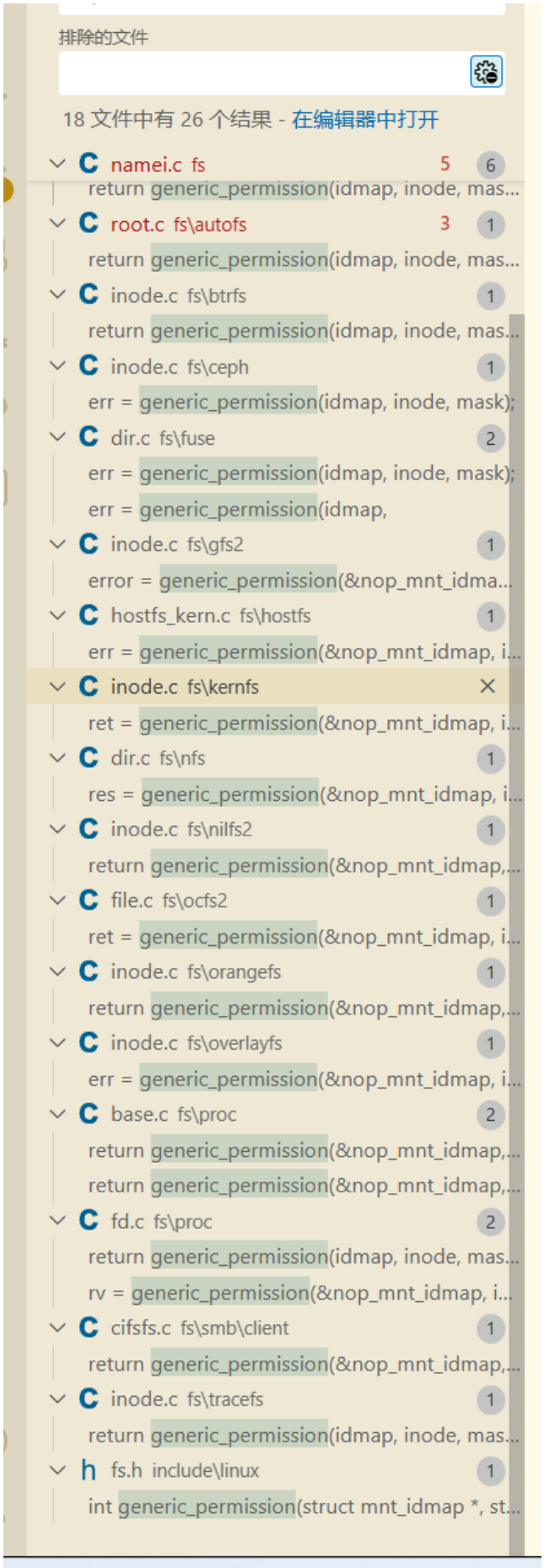

排除的文件
18 文件中有 26 个结果 - 在编辑器中打开
namei.c fs 5 6
return generic_permission(idmap, inode, mas...
root.c fs\autofs 3 1
return generic_permission(idmap, inode, mas...
inode.c fs\btrfs 1
return generic_permission(idmap, inode, mas...
inode.c fs\ceph 1
err = generic_permission(idmap, inode, mask);
dir.c fs\fuse 2
err = generic_permission(idmap, inode, mask);
err = generic_permission(idmap,
inode.c fs\gfs2 1
error = generic_permission(&nop_mnt_idma...
hostfs_kern.c fs\hostfs 1
err = generic_permission(&nop_mnt_idmap, i...
inode.c fs\kernfs
ret = generic_permission(&nop_mnt_idmap, i...
dir.c fs\nfs 1
res = generic_permission(&nop_mnt_idmap, i...
inode.c fs\nilfs2 1
return generic_permission(&nop_mnt_idmap,...
file.c fs\ocfs2 1
ret = generic_permission(&nop_mnt_idmap, i...
inode.c fs\orangefs 1
return generic_permission(&nop_mnt_idmap,...
inode.c fs\overlayfs 1
err = generic_permission(&nop_mnt_idmap, i...
base.c fs\proc 2
return generic_permission(&nop_mnt_idmap,...
return generic_permission(&nop_mnt_idmap,...
fd.c fs\proc 2
return generic_permission(idmap, inode, mas...
rv = generic_permission(&nop_mnt_idmap, i...
cifsfs.c fs\smb\client 1
return generic_permission(&nop_mnt_idmap,...
inode.c fs\tracefs 1
return generic_permission(idmap, inode, mas...
fs.h include\linux 1
int generic_permission(struct mnt_idmap *, st...

图 33 此函数的引用表

如图 34，最外层由 inode_permission 包装

fs > namei.c > inode_permission(mnt_idmap *, inode *, int)

```
554  /**
556   * @idmap:  idmap of the mount the inode was found from
557   * @inode:  Inode to check permission on
558   * @mask:   Right to check for (%MAY_READ, %MAY_WRITE, %MAY_EXEC)
559   *
560   * Check for read/write/execute permissions on an inode.  We use fs[ug]id for
561   * this, letting us set arbitrary permissions for filesystem access without
562   * changing the "normal" UIDs which are used for other things.
563   *
564   * When checking for MAY_APPEND, MAY_WRITE must also be set in @mask.
565   */
566  int inode_permission(struct mnt_idmap *idmap,
567                       struct inode *inode, int mask)
568  {
569      int retval;
570
571      retval = sb_permission(inode->i_sb, inode, mask);
572      if (unlikely(retval))
573          return retval;
574
575      if (unlikely(mask & MAY_WRITE)) {
576          /*
577           * Nobody gets write access to an immutable file.
578           */
579          if (unlikely(IS_IMMUTABLE(inode)))
580              return -EPERM;
581
582          /*
583           * Updating mtime will likely cause i_uid and i_gid to be
584           * written back improperly if their true value is unknown
585           * to the vfs.
586           */
587          if (unlikely(HAS_UNMAPPED_ID(idmap, inode)))
588              return -EACCES;
589      }
590
591      retval = do_inode_permission(idmap, inode, mask);
592      if (unlikely(retval))
593          return retval;
594
595      retval = devcgroup_inode_permission(inode, mask);
596      if (unlikely(retval))
597          return retval;
598
599      return security_inode_permission(inode, mask);
600  }
601  EXPORT_SYMBOL(inode_permission);
602
603  /**
```

图 34 inode_permission

以 exec 为例分析，如图 35。

```
}

/*
 * Calling this is the point of no return. None of the failures will be
 * seen by userspace since either the process is already taking a fatal
 * signal (via de_thread() or coredump), or will have SEGV raised
 * (after exec_mmap()) by search_binary_handler (see below).
 */
int begin_new_exec(struct linux_binprm * bprm)
{
    struct task_struct *me = current;
    int retval;

    /* Once we are committed compute the creds */
    retval = bprm_creds_from_file(bprm);
    if (retval)
        return retval;

    /*
     * This tracepoint marks the point before flushing the old exec where
     * the current task is still unchanged, but errors are fatal (point of
     * no return). The later "sched_process_exec" tracepoint is called after
     * the current task has successfully switched to the new exec.
     */
    trace_sched_prepare_exec(current, bprm);

    /*
```

图 35 begin_new_exec

如图 36，exec 调用的检查 credential，后为一系列的调用链。

```
/*
 * Compute brpm->cred based upon the final binary.
 */
static int bprm_creds_from_file(struct linux_binprm *bprm)
{
    /* Compute creds based on which file? */
    struct file *file = bprm->execfd_creds ? bprm->executable : bprm->file;

    bprm_fill_uid(bprm, file);
    return security_bprm_creds_from_file(bprm, file);
}

/*
```

图 36 bprm_creds_from_file

```
 */
int cap_bprm_creds_from_file(struct linux_binprm *bprm, const struct file *file)
{
    /* Process setpcap binaries and capabilities for uid 0 */
    const struct cred *old = current_cred();
    struct cred *new = bprm->cred;
    bool effective = false, has_fcap = false, id_changed;
    int ret;
    kuid_t root_uid;

    if (WARN_ON(!cap_ambient_invariant_ok(old)))
        return -EPERM;

    ret = get_file_caps(bprm, file, &effective, &has_fcap);
    if (ret < 0)
        return ret;

    root_uid = make_kuid(new->user_ns, 0);

    handle_privileged_root(bprm, has_fcap, &effective, root_uid);

    /* if we have fs caps, clear dangerous personality flags */
    if (__cap_gained(permitted, new, old))
        bprm->per_clear |= PER_CLEAR_ON_SETID;

    /* Don't let someone trace a set[ug]id/setpcap binary with the revised
     * credentials unless they have the appropriate permit.
     *
     * In addition, if NO_NEW_PRIVS, then ensure we get no new privs.
     */
    id_changed = !uid_eq(new->euid, old->euid) || !in_group_p(new->egid);

    if ((id_changed || __cap_gained(permitted, new, old)) &&
        ((bprm->unsafe & ~LSM_UNSAFE_PTRACE) ||
         !ptracer_capable(current, new->user_ns))) {
        /* downgrade; they get no more than they had, and maybe less */
        if (!ns_capable(new->user_ns, CAP_SETUID) ||
            (bprm->unsafe & LSM_UNSAFE_NO_NEW_PRIVS)) {
            new->euid = new->uid;
            new->egid = new->gid;
        }
        new->cap_permitted = cap_intersect(new->cap_permitted,
                                           old->cap_permitted);
    }

    new->suid = new->fsuid = new->euid;
    new->sgid = new->fsgid = new->egid;

    /* File caps or setid cancels ambient. */
```

图 37 cap_bprm_creds_from_file

```
    read_sequnlock_excl(&p->fs->seq);
}

static void bprm_fill_uid(struct linux_binprm *bprm, struct file *file)
{
    /* Handle suid and sgid on files */
    struct mnt_idmap *idmap;
    struct inode *inode = file_inode(file);
    unsigned int mode;
    vfsuid_t vfsuid;
    vfsgid_t vfsgid;
    int err;

    if (!mnt_may_suid(file->f_path.mnt))
        return;

    if (task_no_new_privs(current))
        return;

    mode = READ_ONCE(inode->i_mode);
    if (!(mode & (S_ISUID|S_ISGID)))
        return;

    idmap = file_mnt_idmap(file);

    /* Be careful if suid/sgid is set */
    inode_lock(inode);

    /* Atomically reload and check mode/uid/gid now that lock held. */
    mode = inode->i_mode;
    vfsuid = i_uid_into_vfsuid(idmap, inode);
    vfsgid = i_gid_into_vfsgid(idmap, inode);
    err = inode_permission(idmap, inode, MAY_EXEC);
    inode_unlock(inode);

    /* Did the exec bit vanish out from under us? Give up. */
    if (err)
        return;

    /* We ignore suid/sgid if there are no mappings for them in the ns */
    if (!vfsuid_has_mapping(bprm->cred->user_ns, vfsuid) ||
```

图 38 bprm_fill_uid

从图 38 看到这里也检查了 inode_permission。

2. SeLinux 的分析

security/selinux/ss/context.h

如图 39，在 security.c 中可以看到 open 也 call 了一个 hook，从上文我们知道 open 调用了这个 security_file_open。

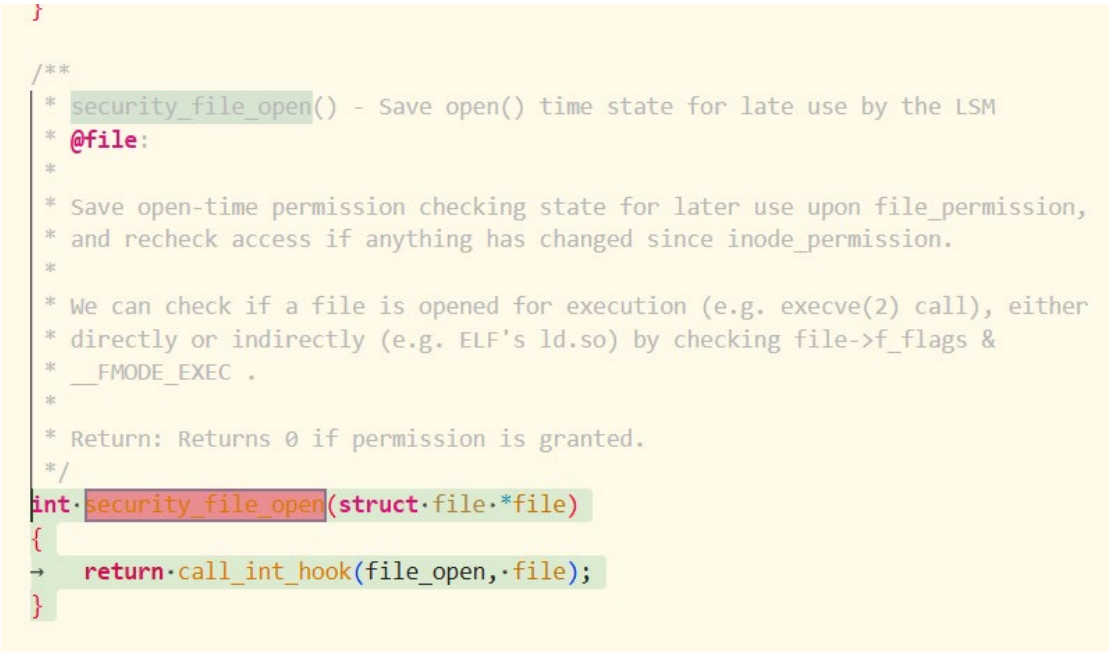
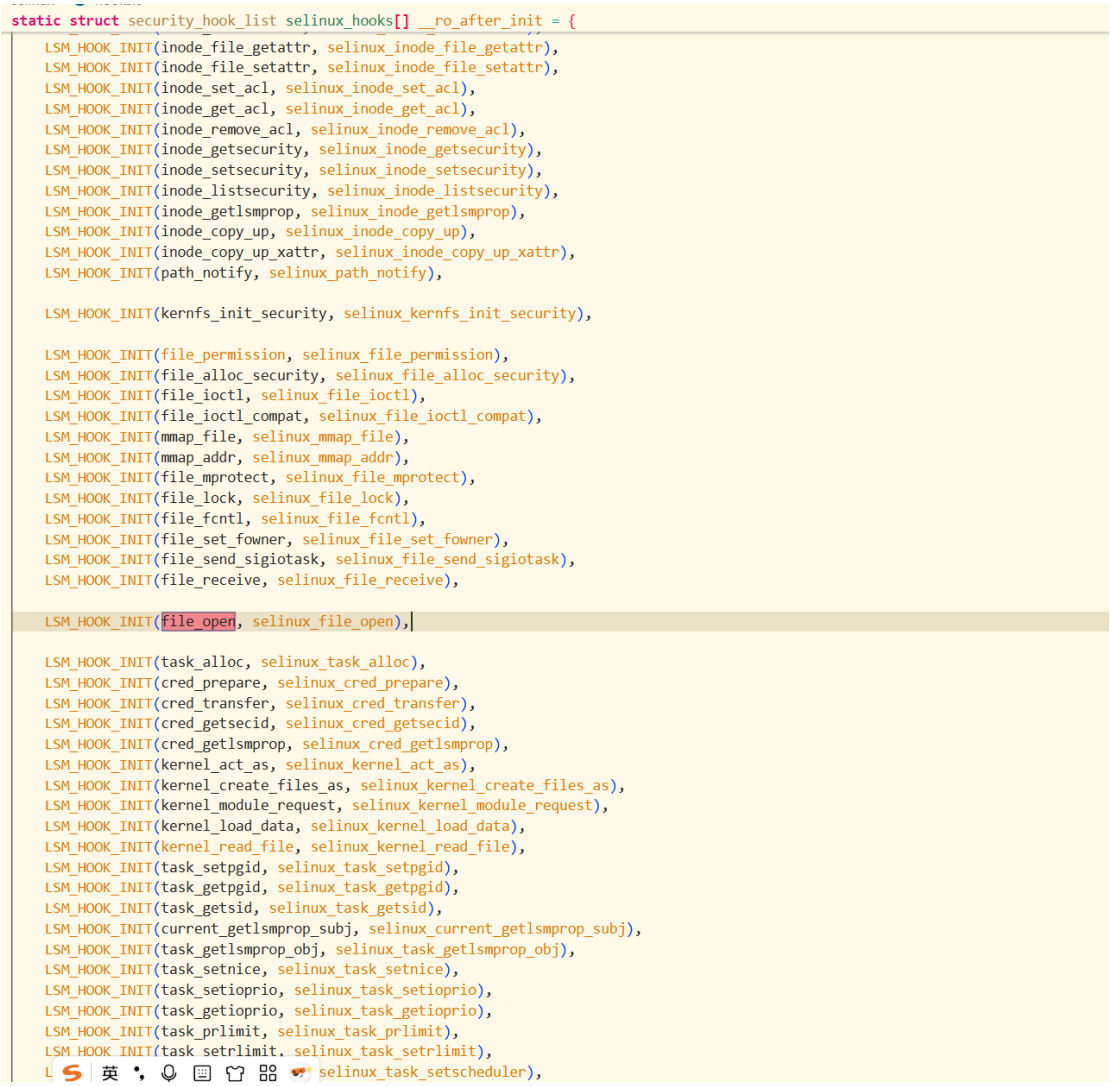


图 39 security_file_open

这个 call 会调用所有模块的 file_open 的 hook，图 40 为 selinux 的 hook 列表。



```
static struct security_hook_list selinux_hooks[] __ro_after_init = {
    LSM_HOOK_INIT(inode_file_getattr, selinux_inode_file_getattr),
    LSM_HOOK_INIT(inode_file_setattr, selinux_inode_file_setattr),
    LSM_HOOK_INIT(inode_set_acl, selinux_inode_set_acl),
    LSM_HOOK_INIT(inode_get_acl, selinux_inode_get_acl),
    LSM_HOOK_INIT(inode_remove_acl, selinux_inode_remove_acl),
    LSM_HOOK_INIT(inode_getsecurity, selinux_inode_getsecurity),
    LSM_HOOK_INIT(inode_setsecurity, selinux_inode_setsecurity),
    LSM_HOOK_INIT(inode_listsecurity, selinux_inode_listsecurity),
    LSM_HOOK_INIT(inode_getlsmprop, selinux_inode_getlsmprop),
    LSM_HOOK_INIT(inode_copy_up, selinux_inode_copy_up),
    LSM_HOOK_INIT(inode_copy_up_xattr, selinux_inode_copy_up_xattr),
    LSM_HOOK_INIT(path_notify, selinux_path_notify),

    LSM_HOOK_INIT(kernfs_init_security, selinux_kernfs_init_security),

    LSM_HOOK_INIT(file_permission, selinux_file_permission),
    LSM_HOOK_INIT(file_alloc_security, selinux_file_alloc_security),
    LSM_HOOK_INIT(file_ioctl, selinux_file_ioctl),
    LSM_HOOK_INIT(file_ioctl_compat, selinux_file_ioctl_compat),
    LSM_HOOK_INIT(mmap_file, selinux_mmap_file),
    LSM_HOOK_INIT(mmap_addr, selinux_mmap_addr),
    LSM_HOOK_INIT(file_mprotect, selinux_file_mprotect),
    LSM_HOOK_INIT(file_lock, selinux_file_lock),
    LSM_HOOK_INIT(file_fcntl, selinux_file_fcntl),
    LSM_HOOK_INIT(file_set_fowner, selinux_file_set_fowner),
    LSM_HOOK_INIT(file_send_sigiotask, selinux_file_send_sigiotask),
    LSM_HOOK_INIT(file_receive, selinux_file_receive),

    LSM_HOOK_INIT(file_open, selinux_file_open),

    LSM_HOOK_INIT(task_alloc, selinux_task_alloc),
    LSM_HOOK_INIT(cred_prepare, selinux_cred_prepare),
    LSM_HOOK_INIT(cred_transfer, selinux_cred_transfer),
    LSM_HOOK_INIT(cred_getsecid, selinux_cred_getsecid),
    LSM_HOOK_INIT(cred_getlsmprop, selinux_cred_getlsmprop),
    LSM_HOOK_INIT(kernel_act_as, selinux_kernel_act_as),
    LSM_HOOK_INIT(kernel_create_files_as, selinux_kernel_create_files_as),
    LSM_HOOK_INIT(kernel_module_request, selinux_kernel_module_request),
    LSM_HOOK_INIT(kernel_load_data, selinux_kernel_load_data),
    LSM_HOOK_INIT(kernel_read_file, selinux_kernel_read_file),
    LSM_HOOK_INIT(task_setpgid, selinux_task_setpgid),
    LSM_HOOK_INIT(task_getpgid, selinux_task_getpgid),
    LSM_HOOK_INIT(task_getsid, selinux_task_getsid),
    LSM_HOOK_INIT(current_getlsmprop_subj, selinux_current_getlsmprop_subj),
    LSM_HOOK_INIT(task_getlsmprop_obj, selinux_task_getlsmprop_obj),
    LSM_HOOK_INIT(task_setnice, selinux_task_setnice),
    LSM_HOOK_INIT(task_setioprio, selinux_task_setioprio),
    LSM_HOOK_INIT(task_getioprio, selinux_task_getioprio),
    LSM_HOOK_INIT(task_prlimit, selinux_task_prlimit),
    LSM_HOOK_INIT(task_setrlimit, selinux_task_setrlimit),
    L                                                  selinux_task_setscheduler),
```

图 40 selinux 的 hook 列表

我们看下 selinux hook 过去的 file_open 函数，如图 41。

```
119  }
120
121  static int selinux_file_open(struct file *file)
122  {
123      struct file_security_struct *fsec;
124      struct inode_security_struct *isec;
125
126      fsec = selinux_file(file);
127      isec = inode_security(file_inode(file));
128      /*
129       * Save inode label and policy sequence number
130       * at open-time so that selinux_file_permission
131       * can determine whether revalidation is necessary.
132       * Task label is already saved in the file security
133       * struct as its SID.
134       */
135      fsec->isid = isec->sid;
136      fsec->pseqno = avc_policy_seqno();
137      /*
138       * Since the inode label or policy seqno may have changed
139       * between the selinux_inode_permission check and the saving
140       * of state above, recheck that access is still permitted.
141       * Otherwise, access might never be revalidated against the
142       * new inode label or new policy.
143       * This check is not redundant - do not remove.
144       */
145      return file_path_has_perm(file->f_cred, file, open_file_to_av(file));
146  }
147
```

图 41 selinux_file_open

```
1723      return inode_has_perm(cred, inode, av, &ad);
1724  }
1725
1726  /* Same as path_has_perm, but uses the inode from the file struct. */
1727  static inline int file_path_has_perm(const struct cred *cred,
1728                                       struct file *file,
1729                                       u32 av)
1730  {
1731      struct common_audit_data ad;
1732
1733      ad.type = LSM_AUDIT_DATA_FILE;
1734      ad.u.file = file;
1735      return inode_has_perm(cred, file_inode(file), av, &ad);
1736  }
1737
```

图 42 file_path_has_perm

```c
/* Check whether a task has a particular permission to an inode.
   The 'adp' parameter is optional and allows other audit
   data to be passed (e.g. the dentry). */
static int inode_has_perm(const struct cred *cred,
                          struct inode *inode,
                          u32 perms,
                          struct common_audit_data *adp)
{
    struct inode_security_struct *isec;
    u32 sid;

    if (unlikely(IS_PRIVATE(inode)))
        return 0;

    sid = cred_sid(cred);
    isec = selinux_inode(inode);

    return avc_has_perm(sid, isec->sid, isec->sclass, perms, adp);
}
```

图 43 inode_has_perm

security > selinux > C avc.c > avc_has_perm(u32, u32, u16, u32, common_audit_data *)

```c
/**
 * avc_has_perm - Check permissions and perform any appropriate auditing.
 * @ssid: source security identifier
 * @tsid: target security identifier
 * @tclass: target security class
 * @requested: requested permissions, interpreted based on @tclass
 * @auditdata: auxiliary audit data
 *
 * Check the AVC to determine whether the @requested permissions are granted
 * for the SID pair (@ssid, @tsid), interpreting the permissions
 * based on @tclass, and call the security server on a cache miss to obtain
 * a new decision and add it to the cache.  Audit the granting or denial of
 * permissions in accordance with the policy.  Return %0 if all @requested
 * permissions are granted, -%EACCES if any permissions are denied, or
 * another -errno upon other errors.
 */
int avc_has_perm(u32 ssid, u32 tsid, u16 tclass,
                 u32 requested, struct common_audit_data *auditdata)
{
    struct av_decision avd;
    int rc, rc2;

    rc = avc_has_perm_noaudit(ssid, tsid, tclass, requested, 0,
                              &avd);

    rc2 = avc_audit(ssid, tsid, tclass, requested, &avd, rc,
                    auditdata);
    if (rc2)
        return rc2;
    return rc;
}
```

图 44 avc_has_perm

图 41-43 为调用链，图 44 的函数进行权限检查，调用了如图 45 的核心函数。

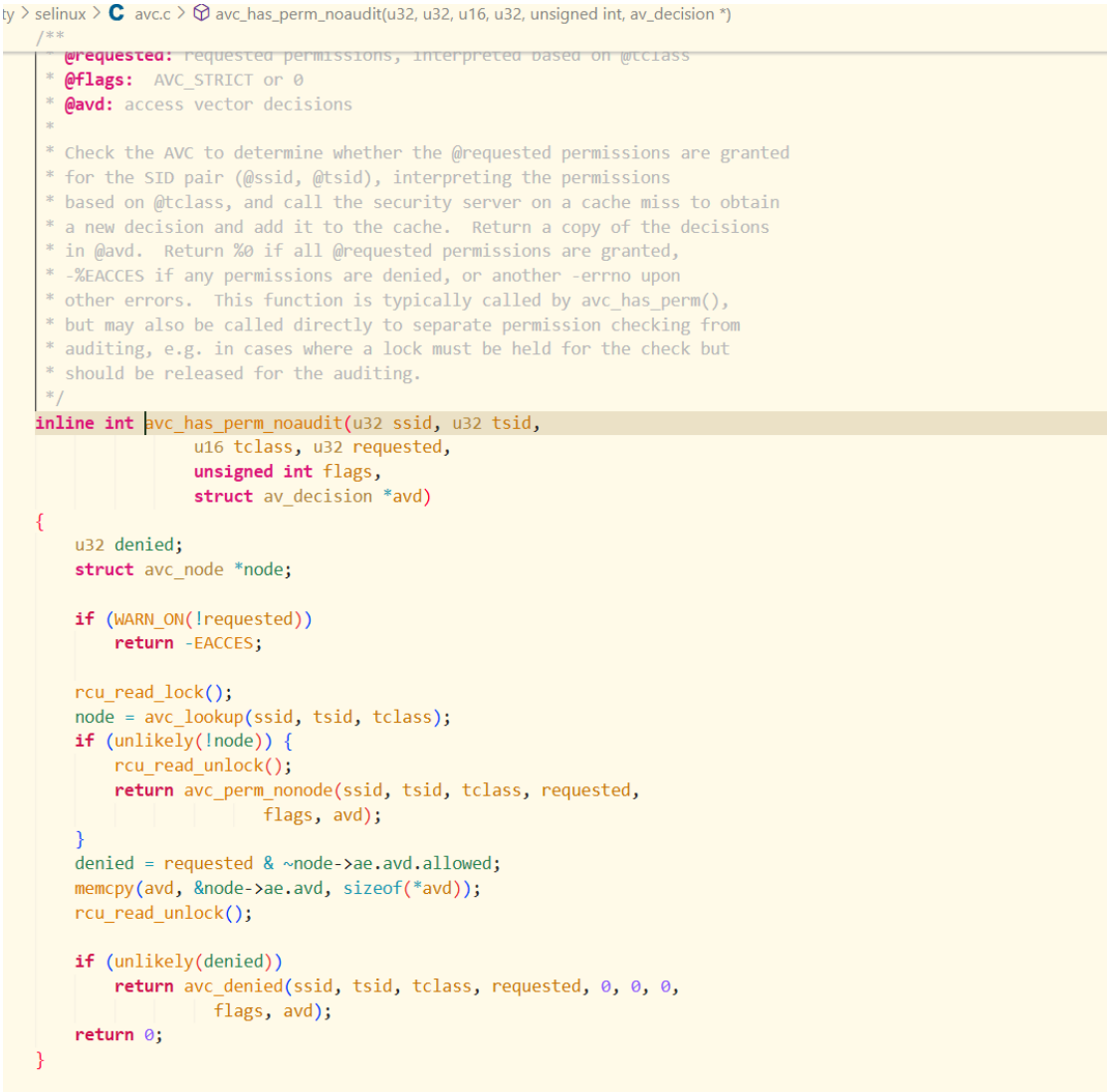


图 45 avc_has_perm_noaudit

这是 avc 的核心函数，可以看关键行，denied = requested & ~node->ae.avd.allowed 说明如果请求了并且没有 allow，则就是 deny。符合其访问控制机制。