Тема 3. Механизмы разграничения доступа ОС Linux

Table of Contents

Безопасность операционных систем
Осень 2025 г.

Веричев Александр Владимирович
@xanchelavederiver alexanderverichev@gmail.com

1. Классическая модель разграничения доступа Unix

1.1. О модели

Классическая модель управления доступом в ОС Unix разрабатывалась в 1970-х годах. Одна из главных задач, решавшихся инженерами - организация многопользовательской работы в ОС.

Управление доступом ОС Unix реализует модель ***избирательного управления доступом***, также именуемую *дискреционная модель*. Избирательное управление доступом (англ. *discretionary access control, DAC*) - управление доступом субъектов к объектам на основе списков управления доступом или матрицы доступа [1].

Основные принципы избирательного доступа:

1. **Владелец ресурса**: пользователь, создавший ресурс, является его владельцем и имеет право устанавливать на него права доступа - кто может получить доступ к этому ресурсу и какие действия (чтение, запись, выполнение) ему разрешены.
2. **Делегирование прав**: владелец может передавать свои права доступа другим пользователям или группам пользователей.
3. **Контроль над действиями**: для каждого субъекта и объекта могут быть явно определены допустимые действия (например, чтение, запись, выполнение).
4. **Идентификация**: все субъекты (пользователи, программы) и объекты (файлы, базы

данных) должны быть однозначно идентифицированы.

Возможны несколько подходов к построению дискреционного управления доступом:

1. Каждый объект системы имеет привязанного к нему субъекта, называемого **владельцем**. Именно владелец устанавливает права доступа к объекту.
2. Система имеет одного выделенного субъекта - **суперпользователя**, который имеет право устанавливать права владения для всех остальных субъектов системы.
3. Субъект с определённым правом доступа может передать это право любому другому субъекту.
4. **Смешанный вариант**: в системе присутствуют как владельцы, устанавливающие права доступа к своим объектам, так и суперпользователь, имеющий возможность изменения прав для любого объекта и/или изменения его владельца (именно такой подход реализован в Unix).

1.2. Объекты, субъекты и виды прав

Субъектами доступа являются **программы**, запущенные от имени какого-либо пользователя. Основные виды объектов доступа показаны в таблице [2, p. 5.4][3, гл. 9].

Table 1. Основные виды объектов доступа

Вид объекта (формат `ls`)	Описание
-	Обычный файл
d	Каталог
l	Символическая ссылка **N.B.** Для символических ссылок права доступа файла всегда отображаются как `rwxrwxrwx` и являются фиктивными. Реальные права хранятся с файлом, на который указывает символическая ссылка
c	Символьный специальный файл Этот тип файла относится к устройству, которое обрабатывает данные как поток байтов, например, терминал или `/dev/null`
b	Блочный специальный файл Этот тип файла относится к устройству, которое обрабатывает данные блоками, например, жесткий диск или DVD-привод

Права доступа к объекту (также называемые режим доступа, *file mode*) представляются в классической модели доступа ОС Unix виде т.н. триплетов прав (*permissions triplet*). Триплеты описывают права доступа к объекту, выданные соответствующей категории субъектов.

Определяются три категории субъектов [4, гл. 15]:

1. **владелец** (также называемый пользователь, **user**) - единственный пользователь, обозначенный как владелец данного файла;

2. **группа** (также группа владельцев, **group**) - совокупность пользователей, объединённых в одну группу;
3. **остальные** (**other**) - прочие пользователи системы.

Каждой категории пользователей могут быть предоставлены следующие три права доступа [4, гл. 15]:

1. **чтение** (**read**, символ **r**) - содержимое файла можно читать;
2. **запись** (**write**, символ **w**) - содержимое файла можно изменять;
3. **выполнение**(**execute**, символ **x**) - данный файл можно исполнить (для программы или сценария).

Таким образом, права доступа трём группам субъектов представляются в виде девяти атрибутов - трёх триплетов прав:

user	group	other
`rwx`	`rwx`	`rwx`

1.3. Права доступа к файлам и директориям

Интерпретация прав на чтение, запись и исполнение для файлов и директорий описаны в таблице ниже и следующих за ней комментариях [2, p. 5.5][3, гл. 9][4, p. 15.4][5].

Атрибут	Файлы	Каталоги
`r`	Разрешает открытие и чтение файла	Разрешает получение списка файлов каталога **N.B.** Для вывода содержимого файлов каталога или информации о них также требуется атрибут исполнения `x`
`w`	Разрешает запись в файл или его усечение **N.B.** Атрибут `w` не позволяет переименовывать или удалять файлы - права на эти операции определяется атрибутами каталога	Разрешает создание, удаление и переименование файлов в каталоге **N.B.** Только при наличии установленного атрибута `x` на каталог!
`x`	Позволяет интерпретировать файл как программу и исполнять его	Разрешает доступ к каталогу: позволяет перейти в каталог (`cd каталог`), читать и изменять метаданные файлов (т.е. создавать, удалять, переименовывать файлы) Разрешение на выполнение применительно к каталогу также иногда называют разрешением на *поиск*

Комментарии по доступу к **файлам**:

1. Атрибут `r` на файл необходим для выполнения копирования файла (т.к. копируемое содержимое необходимо прочитать).
2. Атрибут `w` не позволяет переименовывать или удалять файл, для этого требуются атрибуты `w` и `x` каталога, содержащего файл (см. ниже).
3. Для запуска бинарного исполняемого файла (ELF) достаточно установленного атрибута `x`, права на чтения `r` не нужны:

```
$ cp /bin/ls ./ls-copy
$ chmod 0111 ls-copy
$ ls -la ls-copy
---x--x--x 1 x x 142312 Nov 24 13:06 ls-copy
$ ./ls-copy -la ls-copy
---x--x--x 1 x x 142312 Nov 24 13:06 ls-copy
```

4. При исполнении скрипта - текстового файла, содержащего код программы на интерпретируемом языке (e.g. bash, Python) - в начале файла должен располагаться путь к интерпретатору (т.н. *шебанг*, *shebang*), а также файл должен быть доступен для исполнения `x` и для чтения `r`:

```
$ echo "/bin/ls -la" >> ls.sh
$ chmod 0111 ls.sh
$ ls -la ls.sh
---x--x--x 1 x x 12 Nov 24 13:13 ls.sh
$ ./ls.sh
bash: ./ls.sh: Permission denied
$ chmod 0555 ls.sh
$ ls -la ls.sh
-r-xr-xr-x 1 x x 12 Nov 24 13:14 ls.sh
$ ./ls.sh
total 12
drwxrwxr-x  2 x    x    4096 Nov 24 13:14 .
drwxrwxrwt 20 root root 4096 Nov 24 13:15 ..
-r-xr-xr-x  1 x    x      12 Nov 24 13:14 ls.sh

$ echo "import os; os.system('/bin/ls -la')" >  ls.py
$ chmod 0111 ls.py
bash: ./ls.py: Permission denied
$ chmod 0555 ls.py
$ ./ls.py
./ls.py: line 1: syntax error near unexpected token `''/bin/ls -la''
./ls.py: line 1: `import os; os.system('/bin/ls -la')'

$ echo '#!/bin/python3' | cat - ls.py > temp && mv temp ls.py
$ chmod 0555 ls.py
$ ./ls.py
total 12
```

```
drwxrwxr-x  2 x    x    4096 Nov 24 13:26 .
drwxrwxrwt 20 root root 4096 Nov 24 13:15 ..
-r-xr-xr-x  1 x    x      51 Nov 24 13:26 ls.py
```

`bash` смог исполнить bash-скрипт без шебанга, но только когда у `ls.sh` были выставлены оба атрибута - `x` и `r` . Python-программу запустить без указания пути к интерпретатору не получилось.

Комментарии по доступу к **директориям**:

1. Атрибут `r` на директорию необходим для получения списка названий хранимых в ней файлов, однако без атрибута `x` невозможно получить содержимое этих файлов (даже при наличии прав доступа `r` на файлы), также недоступна информация из индексных дескрипторов файлов (метаданные файлов, в т.ч. триплеты прав):

```
$ mkdir f
$ touch f/{1,2}.txt
$ echo "Hello" > f/3.txt
$ ls -la f
total 8
drwxrwxr-x 2 x x 4096 Nov 24 13:37 .
drwxrwxr-x 3 x x 4096 Nov 24 13:37 ..
-rw-rw-r-- 1 x x    0 Nov 24 13:37 1.txt
-rw-rw-r-- 1 x x    0 Nov 24 13:37 2.txt
-rw-rw-r-- 1 x x    6 Nov 24 13:37 3.txt
$ cat f/3.txt
Hello

$ chmod 0400 f
$ ls -la .
total 12
drwxrwxr-x  3 x    x    4096 Nov 24 13:37 ./
drwxrwxrwt 20 root root 4096 Nov 24 13:15 ../
dr--------  2 x    x    4096 Nov 24 13:37 f/
$ ls -l f/
ls: cannot access 'f/2.txt': Permission denied
ls: cannot access 'f/3.txt': Permission denied
ls: cannot access 'f/1.txt': Permission denied
total 0
-????????? ? ? ? ?            ? 1.txt
-????????? ? ? ? ?            ? 2.txt
-????????? ? ? ? ?            ? 3.txt
$ cat f/3.txt
cat: f/3.txt: Permission denied
```

2. При доступе к файлу разрешение на выполнение необходимо для всех каталогов, которые содержатся в имени пути: например, для чтения файла `/home/x/file.txt` требуется разрешение на выполнение каталогов `/`, `/home` и `/home/x` (а также разрешение

на чтение самого файла `file.txt`).

Если путь задаётся относительный путь к файлу, то права на исполнение нужны только на директории, явно ли неявно заданные в пути: например, пусть текущим рабочим каталогом является `/home/x/sub1` а мы осуществляем доступ по относительному имени пути `../sub2/file.txt`, то в этом случае необходимо иметь разрешение на выполнение для каталогов `/home/x` и `/home/x/sub2` (но не для каталога `/` или `/home`).

3. Обратно, в случае наличия права на исполнение `x` на директорию, но отсутствии права на чтение `r`, доступ к содержимому файла возможен по пути (т.е. в случае, если его название известно), а чтение списка файлов директории невозможно - поиск по имени невозможен:

```
$ mkdir f2
$ echo "Hello!" > f2/file.txt
$ chmod 0100 f2
$ ls -la .
total 16
drwxrwxr-x  4 x    x    4096 Nov 24 14:17 .
drwxrwxrwt 20 root root 4096 Nov 24 13:51 ..
d--x------  2 x    x    4096 Nov 24 14:17 f2
$ ls -la f2/
ls: cannot open directory 'f2/': Permission denied
$ cat f2/file.txt
Hello!
```

Такой подход нередко используется для контроля доступа к содержимому общедоступной директории.

4. Атрибут `w` даёт права на изменение содержимого директории: добавление (создание) или удаление файлов. Однако создание/удаление файла подразумевает редактирование соответствующих метаданных, следовательно, также требуется право на исполнение `x` на директорию!

```
$ mkdir f3
$ echo "Hello" > f3/1.txt
$ ls -la f3/
total 12
drwxrwxr-x 2 x x 4096 Nov 24 14:23 .
drwxrwxr-x 3 x x 4096 Nov 24 14:23 ..
-rw-rw-r-- 1 x x    6 Nov 24 14:23 1.txt
$ chmod 0200 f3
$ ls -la .
total 12
drwxrwxr-x  3 x    x    4096 Nov 24 14:23 .
drwxrwxrwt 20 root root 4096 Nov 24 13:51 ..
d-w-------  2 x    x    4096 Nov 24 14:23 f3
```

```
$ touch f3/2.txt
touch: cannot touch 'f3/2.txt': Permission denied
$ cp f3/1.txt f3/1-copy.txt
cp: cannot create regular file 'f3/1-copy.txt': Permission denied
$ mv f3/1.txt f3/1-move.txt
mv: cannot move 'f3/1.txt' to 'f3/1-move.txt': Permission denied
$ rm f3/1.txt
rm: cannot remove 'f3/1.txt': Permission denied

$ chmod 0700 f3
$ touch f3/2.txt
$ cp f3/1.txt f3/1-copy.txt
$ mv f3/1.txt f3/1-move.txt
$ rm f3/1-move.txt
$ ls f3/
1-copy.txt  2.txt
```

1.4. Алгоритм проверки прав доступа

Проверка прав доступа осуществляется ядром ОС при выполнении каждого системного вызова, содержащего путь к файлу в списке аргументов. Зачастую проверки также затрагивают каталог, в котором располагается данный файл, а также директории, указанные в задаваемом пути. В проверках прав доступа используются идентификаторы пользователя/владельца и группы пользователя/владельца.

Как только файл открывается с помощью системного вызова `open()`, последующие системные вызовы (такие как `read()`, `write()`, `fstat()`, `fcntl()` и `mmap()`), которые работают с возвращенным дескриптором файла, не выполняют проверку прав доступа [4, p. 15.4.3].

Схематично, алгоритм проверки прав доступа выглядит следующим образом [4, p. 15.4.3].

1. Если процесс привилегирован (запущен от `root` или имеет необходимые Capabilities), предоставляется полный доступ.
2. Если эффективный UID процесса совпадает с идентификатором владельца файла, то предоставляется доступ в соответствии с правами доступа владельца файла.
3. Если эффективный GID или идентификатор любой дополнительной группы процесса совпадает с идентификатором группы владельцев файла, то доступ предоставляется в соответствии с правами доступа группы для данного файла.
4. В остальных случаях доступ к файлу предоставляется в соответствии с правами доступа для остальных пользователей.

В ядре перечисленные выше проверки организованы таким образом, чтобы проверка привилегированности процесса выполнялась только в том случае, если процессу не предоставлены необходимые права доступа в результате какой-либо другой проверки.

Проверки прав доступа для владельца, группы и остальных выполняются по порядку и прекращаются, как только обнаруживается применимое правило. Это может привести к неожиданным последствиям: если, например, права доступа для группы превышают права владельца, то последний будет фактически иметь меньше прав доступа к файлу, чем участники группы!

1.5. Специальные разрешения: setuid, setgid и sticky bit

1.5.1. setuid

Бит **setuid** (бит `04000`, часто также обозначаемый как **SUID**) - специальное разрешение доступа, выполняющее единственную функцию: исполняемый файл с SUID всегда выполняется от имени владельца файла, независимо от того, какой пользователь выполнил запуск [6].

В выводе утилиты `ls` и аналогичных setuid обозначается символом `s` в первом триплете. Символ `S` используется в случае, если владелец файла не имеет прав на выполнение (т.е. вместо `x` в первом триплете стоит `s` или `S`).

setuid-программы в директории `/usr/bin`

```
$ find /usr/bin -perm -4000 -type f -exec ls -la {} \;
-rwsr-xr-x 1 root root 39296 Apr  8  2024 /usr/bin/fusermount3
-rwsr-xr-x 1 root root 76248 May 30  2024 /usr/bin/gpasswd
-rwsr-xr-x 1 root root 44760 May 30  2024 /usr/bin/chsh
-rwsr-xr-x 1 root root 51584 Jun  5 16:17 /usr/bin/mount
-rwsr-xr-x 1 root root 55680 Jun  5 16:17 /usr/bin/su
-rwsr-xr-x 1 root root 40664 May 30  2024 /usr/bin/newgrp
-rwsr-xr-x 1 root root 72792 May 30  2024 /usr/bin/chfn
-rwsr-xr-x 1 root root 277936 Jun 25 16:42 /usr/bin/sudo
-rwsr-xr-x 1 root root 64152 May 30  2024 /usr/bin/passwd
-rwsr-xr-x 1 root root 30952 Dec  2  2024 /usr/bin/pkexec
-rwsr-xr-x 1 root root 39296 Jun  5 16:17 /usr/bin/umount
-rwsr-xr-x 1 root root 14656 Sep 23 20:03 /usr/bin/vmware-user-suid-wrapper
```

В случае, если владельцем setuid-программы является `root`, такие исполняемые файлы иногда называют **setuid-root**.

Пример установления setuid на бинарный исполняемый файл

```
$ nano userinfo.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>

int main() {
```

```
    uid_t real_uid = getuid();      // Реальный пользователь (кто запустил)
    uid_t eff_uid = geteuid();      // Эффективный пользователь (под кем выполняется)

    struct passwd *real_pw = getpwuid(real_uid);
    struct passwd *eff_pw = getpwuid(eff_uid);

    printf("Реальный пользователь (кто запустил): %s\n",
           real_pw ? real_pw->pw_name : "неизвестно");
    printf("Эффективный пользователь (под кем выполняется): %s\n",
           eff_pw ? eff_pw->pw_name : "неизвестно");

    return 0;
}

$ gcc -o userinfo userinfo.c
$ chmod u-x userinfo
$ ls -la userinfo
-rw-rwxr-x 1 x x 16096 Nov 24 23:36 userinfo
$ chmod u+s userinfo
$ ls -la userinfo
-rwSrwxr-x 1 x x 16096 Nov 24 23:36 userinfo
$ ./userinfo
-bash: ./userinfo: Permission denied
$ sudo -u user ./userinfo
Реальный пользователь (кто запустил): x
Эффективный пользователь (под кем выполняется): x


$ chmod u+x userinfo
$ chmod u-s userinfo
$ ls -l userinfo
-rwxrwxr-x 1 x x 16096 Nov 24 23:36 userinfo
$ ./userinfo
Реальный пользователь (кто запустил): x
Эффективный пользователь (под кем выполняется): x
$ sudo -u user ./userinfo
Реальный пользователь (кто запустил): user
Эффективный пользователь (под кем выполняется): user

$ sudo chown user:user userinfo
$ sudo chmod u+s userinfo
$ ls -l userinfo
-rwsrwxr-x 1 user user 16096 Nov 24 23:37 userinfo
$ ./userinfo
Реальный пользователь (кто запустил): user
Эффективный пользователь (под кем выполняется): user
$ sudo -u user2 ./userinfo
Реальный пользователь (кто запустил): user
Эффективный пользователь (под кем выполняется): user
$ sudo -u x ./userinfo
Реальный пользователь (кто запустил): user
```

```
Эффективный пользователь (под кем выполняется): user
```

В случае, если у владельца исполняемого файла нет права на исполнение `x`, однако выставлен бит setuid, наблюдается следующее поведение:

1. в выводе утилиты `ls` setuid показан с помощью символа `S` (uppercase)
2. несмотря на бит setuid, запуск от владельца (`x` в примере выше) **невозможен** - т.к. пользователю-владельцу не выданы права на исполнение
3. запуск от иного пользователя **возможен**, причём эффективным пользователем будет
   владелец (`x`)!

Пример установления setuid бита на bash- или Python-скрипт

```
$ ls /home
user  user2  x
$ whoami
x

$ printf '#!/usr/bin/python3\nimport os\nimport
pwd\n\nprint(pwd.getpwuid(os.geteuid()).pw_name)' | tee id.py
$ chmod u+x id.py
$ ls -la id.py
-rwxrw-r-- 1 x x 81 Nov 24 22:38 id.py
$ ./id.py
x
$ sudo -u user ./id.py
sudo: unable to execute ./id.py: Permission denied
$ sudo -u user python3 id.py
user

$ sudo chown user:user id.py
$ sudo chmod u+x id.py
$ ls -la id.py
-rwsrw-r-- 1 user user 81 Nov 24 22:38 id.py
$ ./id.py
-bash: ./id.py: Permission denied


$ echo -e '#!/bin/bash\n\nid' > id.sh
$ chmod u+x id.sh
$ ls -la id.sh
-rwxrw-r-- 1 x x 16 Nov 24 22:46 id.sh
$ ./id.sh
uid=1001(x) gid=1001(x) groups=1001(x),100(users)
$ chmod u+s id.sh
$ ls -la id.sh
-rwsrw-r-- 1 x x 16 Nov 24 22:46 id.sh
```

```
$ ./id.sh
uid=1001(x) gid=1001(x) groups=1001(x),100(users)

$ sudo chown user:user id.sh
$ sudo chmod u+s id.sh
$ ls -la id.sh
-rwsrw-r-- 1 user user 16 Nov 24 22:46 id.sh
$ ./id.sh
-bash: ./id.sh: Permission denied
$ sudo -u user ./id.sh
uid=1000(user) gid=1000(user)
groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),100(users),114(lpadmin
)
```

Руководствуясь соображениями безопасности (времени исполнения, runtime), установленный на скрипт (текстовый файл с шебангом) setuid-бит **игнорируется системой** [7].

1.5.2. setgid

Бит **setgid** (бит `02000`, часто также обозначаемый как **SGID**) - специальное разрешение доступа, выполняющее три функции [6]:

1. при установлении на файл позволяет выполнять файл с правами группы владельцев файла (аналогично setuid)
2. при установлении на каталог, всем файлам, созданным в этом каталоге, будут назначаться группа владельцев, соответствующая владельцу каталога
3. осуществление принудительной блокировки файла [4, p. 51.4]

Аналогично setuid, в выводе утилиты `ls` setgid обозначается символом `s`, но во втором триплете. Символ `S` используется в случае, если группа владельцев файла не имеет прав на выполнение.

Пример установления setgid на бинарный исполняемый файл

```
$ nano groupinfo.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <grp.h>

int main() {
    gid_t real_gid = getgid();      // Реальная группа (которая запустила)
    gid_t eff_gid = getegid();      // Эффективная группа (под которой выполняется)

    struct group *real_gr = getgrgid(real_gid);
    struct group *eff_gr = getgrgid(eff_gid);

    printf("Реальная группа (которая запустила): %s\n",
```

```
            real_gr ? real_gr->gr_name : "неизвестно");
    printf("Эффективная группа (под которой выполняется): %s\n",
            eff_gr ? eff_gr->gr_name : "неизвестно");

    return 0;
}
```

```
$ gcc -o groupinfo groupinfo.c
$ chmod g-x groupinfo
$ ls -la groupinfo
-rwxrw-r-x 1 x x 16096 Nov 25 00:20 groupinfo
$ chmod g+s groupinfo
$ ls -la groupinfo
-rwxrwSr-x 1 x x 16096 Nov 25 00:20 groupinfo
$ ./groupinfo
Реальная группа (которая запустила): x
Эффективная группа (под которой выполняется): x
$ sudo -u user -g x ./groupinfo
sudo: unable to execute ./groupinfo: Permission denied
$ sudo -g user ./groupinfo
Реальная группа (которая запустила): user
Эффективная группа (под которой выполняется): user


$ chmod g+x-s groupinfo
$ ls -l groupinfo
-rwxrwxr-x 1 x x 16096 Nov 25 00:20 groupinfo
$ ./groupinfo
Реальная группа (которая запустила): x
Эффективная группа (под которой выполняется): x
$ sudo -g user ./groupinfo
Реальная группа (которая запустила): user
Эффективная группа (под которой выполняется): user

$ sudo chgrp user groupinfo
$ sudo chmod g+x+s groupinfo
$ ls -l groupinfo
-rwxrwsr-x 1 x user 16096 Nov 25 00:20 groupinfo
$ ./groupinfo
Реальная группа (которая запустила): user
Эффективная группа (под которой выполняется): user
$ sudo -u user2 ./groupinfo
Реальная группа (которая запустила): user
Эффективная группа (под которой выполняется): user
$ sudo -u x ./groupinfo
Реальная группа (которая запустила): user
Эффективная группа (под которой выполняется): user
```

Поведение при запуске исполняемого файла с установленным setgid аналогично setuid:

1. в выводе утилиты `ls` setgid показан с помощью символа `S` (uppercase)
2. несмотря на бит setgid, запуск от группы владельцев (`x` в примере выше) **невозможен** - т.к. группе владельцев не выданы права на исполнение (см запуск `sudo -u user -g x ./groupinfo`)
   N.B. Запуск из-под пользователя `x` возможен из-за бита `x` в первом триплете - т.к. это запуск от владельца
3. запуск от иной группы пользователей **возможен**, причём эффективной группой будет (`x`)!

Пример создания файлов с заданной группой владельцев с помощью setgid на директорию

```
$ mkdir f
$ sudo chgrp user2 f
$ sudo chmod g+s f
$ ls -l .
total 4
drwxrwsr-x 2 x user2 4096 Nov 25 00:42 f

$ groups
x users
$ echo "Hello" > f/1.txt
$ echo "Hello" > f/2.txt
$ sudo -u user2 touch f/3.txt
$ ls -l f/
total 0
-rw-rw-r-- 1 x     user2 0 Nov 25 00:46 1.txt
-rw-rw-r-- 1 x     user2 0 Nov 25 00:46 2.txt
-rw-rw-r-- 1 user2 user2 0 Nov 25 00:46 3.txt

$ sudo -u user cat f/1.txt
Hello
```

1.5.3. sticky-bit

Бит **sticky** (бит `01000`, изредка также называют *бит закрепления*) - специальное разрешение доступа к каталогам [4, p. 15.4.5].

В ранних реализациях UNIX данный бит служил как средство более быстрого запуска часто применяемых программ: при установке на исполняемый файл, во время первого запуска его копия сохранялась в области подкачки - закреплялась в ней и загружалась быстрее при последующих выполнениях. Такое поведение дало название данному биту (sticky - липкий).

В современных реализациях UNIX-подобных систем управление памятью существенно иное, поэтому данное назначение sticky-бита устарело.

В современных реализациях UNIX (в том числе в Linux) sticky-бит игнорируется для файлов,

а для директорий интерпретируется как флаг запрещения удаления файлов: в случае наличия установленного бита непривилегированный пользователь может удалять только те файлы, владельцем которых он является.

Такое поведение позволяет создать каталог, который задействуют одновременно несколько пользователей. Каждый из них может создавать и удалять собственные файлы в данном каталоге, но не может удалять чужие файлы.

В выводе утилиты `ls` sticky-бит обозначается символом `t` в третьем триплете. Символ `T` используется в случае, если для остальные пользователи (other) не имеют прав на исполнение.

Типичный пример использования sticky-бита - директория `/tmp`

```
$ ls -la / | grep tmp
drwxrwxrwt  20 root root       4096 Nov 25 00:30 tmp

$ mkdir -p /tmp/experiments
$ echo "Hello" > /tmp/experiments/file.txt
$ sudo chown user /tmp/experiments/file.txt
$ ls -la /tmp/experiments
total 12
drwxrwxr-x  2 x    x    4096 Nov 25 01:19 .
drwxrwxrwt 20 root root 4096 Nov 25 01:19 ..
-rw-rw-r--  1 user x       6 Nov 25 01:19 file.txt
$ whoami
x
$ rm /tmp/experiments/file.txt
$ ls -l /tmp/experiments/file.txt
ls: cannot access '/tmp/experiments/file.txt': No such file or directory

$ echo "Hello" > /tmp/file2.txt
$ sudo chown user /tmp/file2.txt
$ ls -l /tmp/file2.txt
-rw-rw-r-- 1 user x 6 Nov 25 01:19 /tmp/file2.txt
$ whoami
x
$ rm /tmp/file2.txt
rm: cannot remove '/tmp/file2.txt': Operation not permitted
```

В примере выше первая серия команд демонстрирует возможность удалить файл, владельцем которого является `user`, из-под пользователя `x`; обратите внимание, что для этого пользователю `x` требуются права на исполнение на директорию `/tmp/experiments` (получает по первому триплету, будучи владельцем этой директории).

Вторая серия команд демонстрирует эффект sticky-бита: пользователь `x` не может удалить файл `/tmp/file2.txt`, несмотря на то, что на папку `/tmp`

имеются права на исполнение по третьему триплету.

1.6. Маска режима создания файла процесса: umask()

Рассмотрим процесс назначения прав доступа для создаваемых файлов и директорий [4, p. 15.4.6].

Для новых файлов ядро использует права, указанные в аргументе `mode` системного вызова `open()` или `creat()`. Для новых каталогов права устанавливаются в соответствии с аргументом `mode` команды `mkdir()`. Однако указанные параметры изменяются с помощью т.н. **маски** режима создания файла, известной как **umask**. Этот атрибут процесса указывает, какие биты прав доступа следует всегда **отключать** при создании данным процессом новых файлов или директорий.

Зачастую процесс задействует атрибут `umask`, который он наследует от своей родительской оболочки (например, `/bin/bash`). Как следствие, пользователь может управлять данным атрибутом в запускаемых из командного интерпретатора программах, изменяя атрибут `umask` с помощью специальной утилиты `umask`.

Файлы инициализации в большинстве оболочек по умолчанию устанавливают для атрибута `umask` восьмеричное значение `022` (`----w-w-`). Оно указывает на то, что право на запись должно быть всегда отключено для группы и для остальных пользователей.

Обычно аргумент `mode` системного вызова `open()` равен `0666` (чтение и запись разрешены для всех пользователей), то есть новые файлы создаются с правами доступа на чтение и запись для владельца, а для всех остальных - только с правом доступа на чтение (команда ls –l покажет это как rw-r—r--).

Аналогично, обычно аргумент `mode` системного вызова `mkdir()` устанавливается равным `0777` (все права всем пользователям), поэтому новые каталоги создаются с предоставлением всех прав доступа владельцу, а группам и остальным пользователям предоставляются только права доступа на чтение и выполнение (`rwxr-xr-x`).

1.7. Реализация классической модели разграничения доступа Unix

1.7.1. Принцип хранения и представления атрибутов доступа

Возможной реализацией хранения атрибутов доступа является создание единой таблицы, задающей права доступа к каждому объекту (по строкам) для каждого субъекта (по столбцам) - как показано в примере ниже.

Объект доступа	user1	user2	user3	…
file.txt	rwx	r--	r-x	…
docs/	r-x	rwx	r--	…

Объект доступа	user1	user2	user3	...
script.sh	rwx	r-x	r-x	...
data/	rw-	r--	rw-	...
...	...	...	...	...

Однако такой подход сопряжён с целым рядом ограничений и недостатков, в частности:

1. Даже для относительно "небольших" систем таблица файлов будет содержать минимум десятки тысяч файлов, в реальности количество исчисляется миллионами (см. кол-во `Inodes`).

   ```
   $ sudo df --inodes /
   Filesystem      Inodes  IUsed   IFree IUse% Mounted on
   /dev/sda2      6553600 258474 6295126    4% /
   ```

   Как следствие, поисковые операции в такой таблице будут небесплатными - для **каждой** проверки прав доступа к конкретному файлу будет необходимо найти файл в таблице и извлечь триплеты для заданного пользователя (также, возможно, для группы и остальных).
   Также стоит отметить, что операции, которые могли бы выполняться эффективно с помощью такой таблицы (например, поиск всех файлов, принадлежащих заданному пользователю `user`), весьма редки по сравнению с описанной выше проверкой прав доступа к конкретному файлу.

2. Одна большая таблица представляет собой единую точку отказа - в случае повреждения этого файла самые важные метаданные файловой системы будут утеряны.

3. Иерархическая файловая система Unix-подобных ОС, в том числе ОС Linux, собирается из кусков: различные файловые системы (в том числе FS разного типа - например, ext4 и FAT) монтируются в разные точки (например, "основная" FS в `/`, отдельная FS в `/home`). Причём, в общем случае, точка монтирования FS может изменяться.
   Как следствие, невозможно точно адресовать файл с помощью его абсолютного пути (от корня `/`), и потому невозможно использовать путь как идентификатор файла в таблице.

Как справедливо может заметить внимательный читатель, существуют файловые системы, организованные именно на концепции хранения метаданных файлов в единой таблице - очевидным примером является файловая система NTFS и её *Master File Table*, `$MFT`.

Однако в ОС Windows, в которой преимущественно используется эта FS, не используется концепция единой иерархии файловой системы ОС - разные экземпляры NTFS монтируются в виде отдельных логических дисков `C:`, `D:`, etc, а также механизм управления доступом в ОС Windows значительно отличается от классической модели Unix.

В связи с обозначенными выше проблемами, атрибуты файлов и директорий, в частности, триплеты прав и специальные биты, хранятся отдельно для каждого файла вместе с

другими метаданными файлового объекта - в структуре, называемой **inode**.

1.7.2. Реализация модели Unix в ядре Linux

Хранение идентификаторов пользователей и прав доступа

ОС Linux поддерживает внушительное число файловых систем: ext-{2,3,4}, btrfs, zft, FAT*, NTFS и др. С целью унификации процедур проверок прав доступа и выполнения базовых операций FS детали реализации каждой из поддерживаемых файловых систем скрываются за единым интерфейсом, называемым **виртуальная файловая система** ОС (*Virtual File System*, *VFS*).

Центральной структурой, предназначенной для хранения метаданных (данных о данных - информации, описывающей хранимые в файлах данные) является т.н. `inode`. Структура `inode` слоя VFS содержит базовую информацию о файле или директории, а [дополнительные] метаданные, характерные для конкретной используемой FS, представляются в инкапсулируемой структуре `{fstype}_inode` в виде поля `inode::i_private`.

Идентификаторы владельца файла `uid`, группы владельцев файла `gid`, а также прав доступа `mode` хранятся в структуре `inode` VFS.

include/uapi/asm-generic/posix_types.h

```
...
#ifndef __kernel_uid32_t
typedef unsigned int    __kernel_uid32_t;
typedef unsigned int    __kernel_gid32_t;
#endif
...
```

include/linux/types.h

```
...
typedef unsigned short      umode_t;
...
typedef __kernel_uid32_t    uid_t;
typedef __kernel_gid32_t    gid_t;
...
```

include/linux/uidgid_types.h

```
...
typedef struct {
    uid_t val;
} kuid_t;

typedef struct {
    gid_t val;
} kgid_t;
```

```
...
```

include/linux/fs.h

```
...
struct inode {
    umode_t             i_mode;       ①
    unsigned short      i_opflags;
    kuid_t              i_uid;        ②
    kgid_t              i_gid;        ②
    unsigned int        i_flags;

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl    *i_acl;
    struct posix_acl    *i_default_acl;
#endif
...
```

① 16-битное поле (`unsigned short`), хранящее триплеты прав, биты спец. разрешений, тип файла

② 32-битные поля (`unsigned int`), хранящие идентификаторы владельца и группы владельцев

Поле `i_mode` представляет собой битовое поле, по структуре частично совпадающее с выводом прав доступа и типа файлового объекта утилитой `ls`.

include/uapi/linux/stat.h

```
#define S_IFMT  00170000  ⑦
#define S_IFSOCK 0140000
#define S_IFLNK  0120000
#define S_IFREG  0100000
#define S_IFBLK  0060000
#define S_IFDIR  0040000
#define S_IFCHR  0020000
#define S_IFIFO  0010000
#define S_ISUID  0004000  ④
#define S_ISGID  0002000  ⑤
#define S_ISVTX  0001000  ⑥

#define S_ISLNK(m)  (((m) & S_IFMT) == S_IFLNK)
#define S_ISREG(m)  (((m) & S_IFMT) == S_IFREG)
#define S_ISDIR(m)  (((m) & S_IFMT) == S_IFDIR)
#define S_ISCHR(m)  (((m) & S_IFMT) == S_IFCHR)
#define S_ISBLK(m)  (((m) & S_IFMT) == S_IFBLK)
#define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO)
#define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK)

#define S_IRWXU 00700  ①
#define S_IRUSR 00400
```

```
#define S_IWUSR 00200
#define S_IXUSR 00100

#define S_IRWXG 00070  ②
#define S_IRGRP 00040
#define S_IWGRP 00020
#define S_IXGRP 00010

#define S_IRWXO 00007  ③
#define S_IROTH 00004
#define S_IWOTH 00002
#define S_IXOTH 00001
```

① Триплет прав доступа для владельца файла

② Триплет прав доступа для группы владельцев файла

③ Триплет прав доступа для остальных пользователей

④ setuid-бит

⑤ setgid-бит

⑥ sticky-бит (`S_ISVTX` отражает изначальное название бита - saved-text)

⑦ Биты, хранящие тип файла: обычный файл (`S_IFREG`), директория (`S_IFDIR`), символическая ссылка (`S_IFLNK`), символьное/блочное устройство (`S_IFCHR`/`S_IFBLK`), сокет (`S_IFSOCK`) или именованный канал (`S_IFIFO`)

Пример выполнения проверки прав доступа: функция удаления файла `vfs_unlink`

Разберём ***часть*** реализации проверки прав доступа при удалении файла, релевантных классической модели Unix (проверки, связанные с прочими атрибутами файла и содержащей его директории игнорируем).

Напомним, для удаления файла из директории пользователь должен иметь права на запись `w` и исполнение `x` на эту директорию.

Удаление файла выполняет с помощью функции `vfs_unlink`, права на выполнение удаления проверяется в функции `may_delete`, причём права, заданные триплетами прав, проверяются в функции `acl_permission_check`. Стек вызова функции `acl_permission_check` выглядит *примерно* так:

```
1 acl_permission_check(..., *dir_inode, MAY_WRITE | MAY_EXEC)
2 generic_permission(..., *dir_inode, MAY_WRITE | MAY_EXEC)    472
3 do_inode_permission(..., *dir_inode, MAY_WRITE | MAY_EXEC)   533
4 inode_permission(..., *dir_inode, MAY_WRITE | MAY_EXEC)     593  ①
5 may_delete(..., *dir_inode, ...)                             3274
6 vfs_unlink(..., *dir_inode, ...)                             4654
```

① Для удаления файла необходимы права на запись и исполнение (задаются флагом `MAY_WRITE | MAY_EXEC`) на директорию, хранящую файл (`dir_inode` описывает её `inode`)

linux/fs/namei.c

```
int vfs_unlink(struct mnt_idmap *idmap, struct inode *dir,
               struct dentry *dentry, struct inode **delegated_inode)
{
    struct inode *target = dentry->d_inode;
    int error = may_delete(idmap, dir, dentry, 0);  ①
    ...
```

① Проверки прав на удаление выполняются в функции `may_delete`

linux/fs/namei.c

```
static int may_delete(struct mnt_idmap *idmap, struct inode *dir,
                      struct dentry *victim, bool isdir)
{
    ...
    /* Inode writeback is not safe when the uid or gid are invalid. */
    if (!vfsuid_valid(i_uid_into_vfsuid(idmap, inode)) ||
        !vfsgid_valid(i_gid_into_vfsgid(idmap, inode)))
        return -EOVERFLOW;

    audit_inode_child(dir, victim, AUDIT_TYPE_CHILD_DELETE);

    error = inode_permission(idmap, dir, MAY_WRITE | MAY_EXEC);  ①
    if (error)
        return error;
    if (IS_APPEND(dir))
        return -EPERM;

    if (check_sticky(idmap, dir, inode) || IS_APPEND(inode) ||
        IS_IMMUTABLE(inode) || IS_SWAPFILE(inode) ||
        HAS_UNMAPPED_ID(idmap, inode))
        return -EPERM;
    ...
```

① Проверки прав, заданных в структуре `inode`, вынесены в функцию `inode_permission`
N.B. Проверяются права `MAY_WRITE` (`w`) и `MAY_EXEC` (`x`) на директорию, которую описывает `inode *dir`

linux/fs/namei.c

```
int inode_permission(struct mnt_idmap *idmap,
                     struct inode *inode, int mask)
{
    ...
    retval = do_inode_permission(idmap, inode, mask);  ①
    ...
```

① Проверки прав UNIX выполняются в функции `do_inode_permission`

linux/fs/namei.c

```
static inline int do_inode_permission(struct mnt_idmap *idmap,
                      struct inode *inode, int mask)
{
    ...
    return generic_permission(idmap, inode, mask);  ①
}
```

① В случае, если для заданной `inode` не определена специальная функция проверки прав, используется общая функция `generic_permission`, которая реализует проверки для Posix-like файловой системы

linux/fs/namei.c

```
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);     ①
    if (ret != -EACCES)                                 ②
        return ret;

    if (S_ISDIR(inode->i_mode)) {
        /* DACs are overridable for directories */
        if (!(mask & MAY_WRITE))
            if (capable_wrt_inode_uidgid(idmap, inode,  ②
                         CAP_DAC_READ_SEARCH))
                return 0;
        if (capable_wrt_inode_uidgid(idmap, inode,  ②
                       CAP_DAC_OVERRIDE))
            return 0;
        return -EACCES;
    }
    ...
```

① Основные проверки вынесены в функцию `acl_permission_check`

② Проверки Unix-прав выполняются первыми, если проверка неуспешна (т.е. триплеты не дают прав) выполняется проверка прочих условий, в частности, имеет ли пользователь соответствующие Capability (является ли он `root`)

linux/fs/namei.c

```
381 static int acl_permission_check(struct mnt_idmap *idmap,
382                  struct inode *inode, int mask)
383 {
```

```
384     unsigned int mode = inode->i_mode;
385     vfsuid_t vfsuid;
386
387     /*
388      * Common cheap case: everybody has the requested
389      * rights, and there are no ACLs to check. No need
390      * to do any owner/group checks in that case.
391      *
392      *  - 'mask&7' is the requested permission bit set
393      *  - multiplying by 0111 spreads them out to all of ugo
394      *  - '& ~mode' looks for missing inode permission bits
395      *  - the '!' is for "no missing permissions"
396      *
397      * After that, we just need to check that there are no
398      * ACL's on the inode - do the 'IS_POSIXACL()' check last
399      * because it will dereference the ->i_sb pointer and we
400      * want to avoid that if at all possible.
401      */
402     if (!((mask & 7) * 0111 & ~mode)) {  ①
403         if (no_acl_inode(inode))
404             return 0;
405         if (!IS_POSIXACL(inode))
406             return 0;
407     }
408
409     /* Are we the owner? If so, ACL's don't matter */
410     vfsuid = i_uid_into_vfsuid(idmap, inode);                ②
411     if (likely(vfsuid_eq_kuid(vfsuid, current_fsuid()))) {  ②
412         mask &= 7;   ③
413         mode >>= 6;
414         return (mask & ~mode) ? -EACCES : 0;
415     }
416
417     /* Do we have ACL's? */
418     if (IS_POSIXACL(inode) && (mode & S_IRWXG)) {
419         int error = check_acl(idmap, inode, mask);
420         if (error != -EAGAIN)
421             return error;
422     }
423
424     /* Only RWX matters for group/other mode bits */
425     mask &= 7;
426
427     /*
428      * Are the group permissions different from
429      * the other permissions in the bits we care
430      * about? Need to check group ownership if so.
431      */
432     if (mask & (mode ^ (mode >> 3))) {  ④
433         vfsgid_t vfsgid = i_gid_into_vfsgid(idmap, inode);  ⑤
434         if (vfsgid_in_group_p(vfsgid))
```

```
435             mode >>= 3;
436     }
437
438     /* Bits in 'mode' clear that we require? */
439     return (mask & ~mode) ? -EACCES : 0;  ⑥
440 }
```

① Проверка случая, когда требуемые права установлены для всех пользователей (в этом случае нет необходимости получать владельца и группу владельцев файла)

② Проверка, является ли текущий пользователь владельцем
N.B. При проверке прав используется `FSUID`

③ Текущий пользователь - владелец, потому проверяем, установлены ли требуемые права в первом триплете прав (его получаем сдвигом на два триплета вправо: `mode >>= 6`)

④ Проверяем, отличаются ли права доступа для группы владельцев от прав для всех остальных

⑤ Права отличаются, проверяем по второму триплету - перезаписываем третий триплет вторым (`mode >>= 3`)

⑥ Проверяем права по третьему триплету (если на самом деле проверка идёт по группе владельцев, на месте третьего триплета записан второй - см. шаг 5)

Как можно видеть в приведённых выше листингах кода, проверка прав доступа к объекту в современной ОС Linux весьма сложна и включает себя кроме UNIX-проверок (триплетов прав и специальных битов) прочие проверки - в частности, проверки правил списка контроля доступа (см. Списки контроля доступа) и привилегий пользователя (является ли он суперпользователем `root` или владеет соответствующими Capabilities).

Обратите внимание, что первыми выполняются Unix-проверки. Такой подход позволяет избежать извлечения дополнительных атрибутов и метаданных в случае, если права доступа могут быть предоставлены непосредственно на основе триплетов прав.

Пример выполнения проверки прав доступа: функция создания файла `vfs_create`

С точки зрения проверки прав операция создания файла аналогична удалению, т.к. для создания файла в директории пользователь также должен иметь права на запись `w` и исполнение `x` на эту директорию.

Создание файла выполняет с помощью функции `vfs_create`, права на выполнение удаления проверяется в функции `may_create`, а триплеты прав проверяются с помощью той же функции `acl_permission_check`. Стек вызова функции `acl_permission_check` выглядит *примерно* так:

```
1 acl_permission_check(..., *dir_inode, MAY_WRITE | MAY_EXEC)
2 generic_permission(..., *dir_inode, MAY_WRITE | MAY_EXEC)    472
3 do_inode_permission(..., *dir_inode, MAY_WRITE | MAY_EXEC)   533
4 inode_permission(..., *dir_inode, MAY_WRITE | MAY_EXEC)      593  ①
```

```
5 may_create(..., *dir_inode, ...)                    3274
6 vfs_create(..., *dir_inode, ...)                    4654
```

① Для создания файла необходимы права на запись и исполнение (задаются флагом `MAY_WRITE | MAY_EXEC`) на директорию, в которой создаётся файл (`dir_inode` описывает её `inode`)

linux/fs/namei.c

```
3477 int vfs_create(struct mnt_idmap *idmap, struct inode *dir,
3478            struct dentry *dentry, umode_t mode, bool want_excl)
3479 {
3480     int error;
3481
3482     error = may_create(idmap, dir, dentry);  ①
3483     if (error)
3484         return error;
3485
3486     if (!dir->i_op->create)
3487         return -EACCES; /* shouldn't it be ENOSYS? */
3488
3489     mode = vfs_prepare_mode(idmap, dir, mode, S_IALLUGO, S_IFREG);
3490     error = security_inode_create(dir, dentry, mode);
3491     if (error)
3492         return error;
3493     error = dir->i_op->create(idmap, dir, dentry, mode, want_excl);
3494     if (!error)
3495         fsnotify_create(dir, dentry);
3496     return error;
3497 }
```

① Проверки прав на создание выполняются в функции `may_create`

linux/fs/namei.c

```
3307 static inline int may_create(struct mnt_idmap *idmap,
3308                  struct inode *dir, struct dentry *child)
3309 {
3310     audit_inode_child(dir, child, AUDIT_TYPE_CHILD_CREATE);
3311     if (child->d_inode)
3312         return -EEXIST;
3313     if (IS_DEADDIR(dir))
3314         return -ENOENT;
3315     if (!fsuidgid_has_mapping(dir->i_sb, idmap))
3316         return -EOVERFLOW;
3317
3318     return inode_permission(idmap, dir, MAY_WRITE | MAY_EXEC);  ①
3319 }
```

① Проверки прав вынесены в функцию `inode_permission`; проверяются права `MAY_WRITE` (`w`) и

`MAY_EXEC` (`x`) на директорию `inode *dir`

Проверка прав доступа при открытии файла

Требуемые для открытия файла права зависят от режима открытия, задаваемого с помощью аргумента `mode_t mode` системных функций open(2), openat(2), openat2(2). Системные функции исполняются в user space и взывают один из системных вызовов (syscall), выполняющихся уже в ядре.

Права доступа проверяются в функции `may_open`, а триплеты прав проверяются с помощью всё той же функции `acl_permission_check`. Цепочка вызовов функции `acl_permission_check` при открытии файла *может* выглядеть следующим образом:

```
open(), openat(), openat2()
              |
User space    |
==============|==========================
Kernel space  |
              ▯
sys_open, sys_openat, sys_openat2
              ▯
do_sys_open() / do_sys_openat2()
              ▯
       do_filp_open()
              ▯
         path_openat()
              ▯
link_path_walk() → do_open()
              ▯
          may_open()
              ▯
       inode_permission()
```

linux/fs/open.c:1456

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;
    return do_sys_open(AT_FDCWD, filename, flags, mode);  ①
}

SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,
        umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;
    return do_sys_open(dfd, filename, flags, mode);  ①
}
```

```
SYSCALL_DEFINE4(openat2, int, dfd, const char __user *, filename,
        struct open_how __user *, how, size_t, usize)
{
    int err;
    struct open_how tmp;
    ...
    return do_sys_openat2(dfd, filename, &tmp);  ②
}
```

① Системные вызовы `open()` и `openat()` являются обёртками функции `do_sys_open()`

② Системный вызов `openat2()` выполняет ряд проверок и подготовительных операций (пропущены в листинге) и вызывает функцию `do_sys_openat2()`

linux/fs/open.c:1449

```
int do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    struct open_how how = build_open_how(flags, mode);  ①
    return do_sys_openat2(dfd, filename, &how);  ②
}
```

① Режим открытия файла, флаги `flags` (`O_RDONLY`, `O_WRONLY` и прочие `O_*`) и права доступа `mode` (`S_IRUSR`, `S_IWUSR` и иные `S_I*`), собираются в структуре `struct open_how`

② Вызывается функция `do_sys_openat2()` - `do_sys_open()` в свою очередь является обёрткой

linux/fs/open.c:1420

```
static int do_sys_openat2(int dfd, const char __user *filename,
              struct open_how *how)
{
    struct open_flags op;
    ...
    err = build_open_flags(how, &op);
    ...
        struct file *f = do_filp_open(dfd, tmp, &op);  ①
    ...
}
```

① Формируется структура флагов `struct open_flags` и вызывается функция `do_filp_open()`

linux/fs/namei.c:4153

```
struct file *do_filp_open(int dfd, struct filename *pathname,
        const struct open_flags *op)
{
    struct nameidata nd;
    int flags = op->lookup_flags;
    struct file *filp;

    set_nameidata(&nd, dfd, pathname, NULL);
```

```
    filp = path_openat(&nd, op, flags | LOOKUP_RCU);  ①
    if (unlikely(filp == ERR_PTR(-ECHILD)))
        filp = path_openat(&nd, op, flags);  ①
    if (unlikely(filp == ERR_PTR(-ESTALE)))
        filp = path_openat(&nd, op, flags | LOOKUP_REVAL);  ①
    restore_nameidata();
    return filp;
}
```

① Функция `do_filp_open()` вызывает `path_openat()`

linux/fs/namei.c:4114

```
static struct file *path_openat(struct nameidata *nd,
            const struct open_flags *op, unsigned flags)
{
    struct file *file;
    int error;

    file = alloc_empty_file(op->open_flag, current_cred());
    if (IS_ERR(file))
        return file;

    if (unlikely(file->f_flags & __O_TMPFILE)) {
        error = do_tmpfile(nd, flags, op, file);  ①
    } else if (unlikely(file->f_flags & O_PATH)) {
        error = do_o_path(nd, flags, file);  ②
    } else {
        const char *s = path_init(nd, flags);
        while (!(error = link_path_walk(s, nd)) &&
               (s = open_last_lookups(nd, file, op)) != NULL)
            ;
        if (!error)
            error = do_open(nd, file, op);  ③
        terminate_walk(nd);
    }
    ...
```

① Установлен флаг `O_TMPFILE` - требуется создать именованный временный файл, для этого вызывается `do_tmpfile()`

② Установлен флаг `O_PATH` - требуется получить файловый дескриптор файла без его открытия, для этого вызывается функция `do_o_path()`

③ Выполняется поиск файла по заданному пути и вызов функции `do_open()` для его открытия

linux/fs/namei.c:3931

```
static int do_open(struct nameidata *nd,
           struct file *file, const struct open_flags *op)
{
```

```
    ...
    do_truncate = false;
    acc_mode = op->acc_mode;
    if (file->f_mode & FMODE_CREATED) {
        /* Don't check for write permission, don't truncate */
        open_flag &= ~O_TRUNC;
        acc_mode = 0;  ①
    } else if (d_is_reg(nd->path.dentry) && open_flag & O_TRUNC) {
        error = mnt_want_write(nd->path.mnt);
        if (error)
            return error;
        do_truncate = true;
    }
    error = may_open(idmap, &nd->path, acc_mode, open_flag);  ②
    if (!error && !(file->f_mode & FMODE_OPENED))
        error = vfs_open(&nd->path, file);  ③
    if (!error)
        error = security_file_post_open(file, op->acc_mode);
    if (!error && do_truncate)
        error = handle_truncate(idmap, file);
    ...
```

① Открываемый файл ранее не существовал (т.е. был создан в ходе открытия), поэтому проверка прав доступа не выполняется и все биты `acc_mode` сбрасываются
N.B. Т.к. файл был создан, ему назначены права `0666 & umask`, а владелец и группа установлены на основе `FSUID`/`FSGID` процесса

② Права доступа для требуемого режима открытия файла проверяются в функции `may_open()`

③ В случае успешной проверки прав открытие файла выполняется посредством VFS-функции `vfs_open()`

linux/fs/namei.c:3527

```
3527 static int may_open(struct mnt_idmap *idmap, const struct path *path,
3528               int acc_mode, int flag)
3529 {
3530     struct dentry *dentry = path->dentry;
3531     struct inode *inode = dentry->d_inode;
3532     int error;
3533
3534     if (!inode)
3535         return -ENOENT;
3536
3537     switch (inode->i_mode & S_IFMT) {  ①
3538     case S_IFLNK:
3539         return -ELOOP;
3540     case S_IFDIR:
3541         if (acc_mode & MAY_WRITE)
3542             return -EISDIR;
3543         if (acc_mode & MAY_EXEC)
```

```
3544            return -EACCES;
3545        break;
3546    case S_IFBLK:
3547    case S_IFCHR:
3548        if (!may_open_dev(path))
3549            return -EACCES;
3550        fallthrough;
3551    case S_IFIFO:
3552    case S_IFSOCK:
3553        if (acc_mode & MAY_EXEC)
3554            return -EACCES;
3555        flag &= ~O_TRUNC;
3556        break;
3557    case S_IFREG:
3558        if ((acc_mode & MAY_EXEC) && path_noexec(path))  ②
3559            return -EACCES;
3560        break;
3561    default:
3562        VFS_BUG_ON_INODE(!IS_ANON_FILE(inode), inode);
3563    }
3564
3565    error = inode_permission(idmap, inode, MAY_OPEN | acc_mode);  ③
3566    if (error)
3567        return error;
3568
3569    /*
3570     * An append-only file must be opened in append mode for writing.
3571     */
3572    if (IS_APPEND(inode)) {
3573        if  ((flag & O_ACCMODE) != O_RDONLY && !(flag & O_APPEND))
3574            return -EPERM;
3575        if (flag & O_TRUNC)
3576            return -EPERM;
3577    }
3578
3579    /* O_NOATIME can only be set by the owner or superuser */
3580    if (flag & O_NOATIME && !inode_owner_or_capable(idmap, inode))
3581        return -EPERM;
3582
3583    return 0;
3584 }
```

① В зависимости от типа открываемого файла до проверок прав доступа выполняются специальные проверки
N.B. `inode→i_mode & S_IFMT` оставляет только биты, задающие тип файла

② В случае необходимости открытия файла с правом на исполнение с помощью функции `path_noexec()` выполняется проверка флагов монтирования файловой системы, в которой хранится этот файл
N.B. В ОС Linux имеется возможность смонтировать файловую систему в особом режиме - например, без права исполнения файлов или только для чтения

③ Права доступа дополняются правом на чтение (если бит `MAY_OPEN` не был установлен до этого) и вызывается функция проверки прав доступа `inode_permission()`

Дальнейшие проверки прав согласно классической Unix-модели выполняются аналогично рассмотренным ранее примерам удаления и создания файлов.

Стоит отдельно отметить проверку флагов монтирования файловой системы в случае, если режим открытия файла предполагает исполнение файла:

linux/fs/exec.c:115

```
115 bool path_noexec(const struct path *path)
116 {
117     /* If it's an anonymous inode make sure that we catch any shenanigans. */
118     VFS_WARN_ON_ONCE(IS_ANON_FILE(d_inode(path->dentry)) &&
119              !(path->mnt->mnt_sb->s_iflags & SB_I_NOEXEC));
120     return (path->mnt->mnt_flags & MNT_NOEXEC) ||
121            (path->mnt->mnt_sb->s_iflags & SB_I_NOEXEC);  ①
122 }
```

① Если флаги `SB_I_NOEXEC` или `MNT_NOEXEC` установлены, функция возвращает `true`

Внимательный читатель (кто ты, герой? дожил ли кто-нибудь до этого места?..) может заметить, что помимо наличия требуемых прав на собственно файл у процесса должны быть права на исполнение `x` на все директории, указанные в пути к файлу, заданному при вызове функции `open*` (см. Комментарии по доступу к директориям)!

Наличие у процесса прав на исполнение на все фрагменты пути выполняется в разрешения пути в функции `path_openat()`:

linux/fs/namei.c

```
static struct file *path_openat(struct nameidata *nd,
            const struct open_flags *op, unsigned flags)
{
    ...
        const char *s = path_init(nd, flags);
        while (!(error = link_path_walk(s, nd)) &&  ①
               (s = open_last_lookups(nd, file, op)) != NULL)
            ;
        if (!error)
            error = do_open(nd, file, op);
        terminate_walk(nd);
    ...
}

static int link_path_walk(const char *name, struct nameidata *nd)
{
    ....
    /* At this point we know we have a real path component. */
    for(;;) {  ②
```

```
        struct mnt_idmap *idmap;
        const char *link;
        unsigned long lastword;

        idmap = mnt_idmap(nd->path.mnt);
        err = may_lookup(idmap, nd);  ②
        if (unlikely(err))
            return err;
    ...
}

static inline int may_lookup(struct mnt_idmap *idmap,
                 struct nameidata *restrict nd)
{
    int err, mask;

    mask = nd->flags & LOOKUP_RCU ? MAY_NOT_BLOCK : 0;
    err = inode_permission(idmap, nd->inode, mask | MAY_EXEC);  ③
    if (likely(!err))
        return 0;

    // If we failed, and we weren't in LOOKUP_RCU, it's final
    if (!(nd->flags & LOOKUP_RCU))
        return err;

    // Drop out of RCU mode to make sure it wasn't transient
    if (!try_to_unlazy(nd))
        return -ECHILD; // redo it all non-lazy

    if (err != -ECHILD) // hard error
        return err;

    return inode_permission(idmap, nd->inode, MAY_EXEC);  ③
}
```

① В процессе разрешения пути вызывается функция `link_path_walk()`

② Функция `link_path_walk()` вызывает `may_lookup(idmap, nd)` для каждого директории пути

③ Функция `may_lookup()` вызывает функцию проверки прав `inode_permission()` с запрашиваемым битом `MAY_EXEC` - выполняется проверка права на исполнение `x` на соответствующую директорию пути

Подытоживая, в ходе выполнения операции открытия файла ядро выполняет проверки прав доступа согласно модели Unix:

- в функции `may_open()` - выполняются проверки заданных режимом открытия прав доступа `acc_mode` и флагов монтирования файловой системы (возможность исполнения файлов этой FS при требуемом праве на исполнение `x` файла);
- в функции `link_path_walk()` - выполняется проверка наличия права на исполнение `x` на каждую директорию, явно или опосредованно указанную в заданном в системной

функции `open*()` пути к файлу.

1.8. Ограничения классической модели разграничения доступа Unix

Классическая модель разграничения доступа Unix позволяет решать базовые задачи, однако круг задач, возникающих на практике, существенно шире возможностей чистой DAC-модели.

Частично дополнительный функционал реализован посредством внедрения специальных битов setuid, setgid и sticky-бита, однако эти механизмы не позволяют решить множество других задач, некоторые из которых перечислены ниже.

1. Запрет редактирования файла всем пользователям

Запрет на изменение содержимого некоторого файла любому непривилегированному пользователю может быть реализован посредством удаления права на запись `w` в соответствующем триплете прав. Однако при выполнении действий от имени `root` результаты проверок триплетов прав доступа игнорируются.

Следовательно, запретить пользователю `root` редактирование файла средствами модели Unix невозможно.

2. Разрешение только добавления данных в файл

В некоторых случаях требуется запретить редактирование существующего содержимого файла, разрешая добавление данных в файл. Такой режим открытия файла называют *append-only*.

Очевидный пример: лог-файлы, в которые в штатном режиме работы системы постоянно дописываются новые записи, в том числе непривилегированными процессами, при этом требуется защитить содержимое от удаления злоумышленником.

Данная задача не может быть реализована штатными средствами классической модели Unix даже для обычных пользователей: право на запись `w` даёт возможность редактировать любую часть содержимого файла.

3. Запрет удаления файлов из директории

Иногда требуется *запретить удаление* файлов, оставив возможность полного редактирования их содержимого, а также право *создавать новые файлы* в директории.

В качестве примера потенциально требующих защиты от удаления файлов можно привести конфигурационные файлы, лог-файлы, исполняемые файлы и скрипты.

Классическая модель разграничения доступа Unix не позволяет решить обозначенную проблему, так как возможность создания новых файлов означает наличие права на запись `w` на директорию, однако право на запись также означает возможность редактирования списка файлов - то есть удаление любого файла директории.

Очевидно, для запрета удаления конкретного файла требуется вспомогательный механизм управления доступом.

4. Гибкая настройка прав доступа для нескольких пользователей или групп

Классическая модель Unix определяет три группы субъектов: владелец, группа владельцев и остальные пользователи. Каждый непривилегированный пользователь системы относится к одной из этих групп. Права доступа определяются только для этих групп с помощью соответствующего триплета.

В реальных сценариях работы в ОС такой подход существенно ограничивает реализуемые сценарии управления доступом. Рассмотрим несколько примеров.

i. Требуется выдать права на чтение и запись пользователю `user1`, права только на чтение пользователю `user2`, запретить доступ для группы и остальных.
Такой режим доступа практически невозможно организовать с помощью модели Unix (*подумайте, как можно решить эту задачу триплетами прав, в чем проблема такого решения?*).

ii. Файлы проекта `project/` доступны для чтения и записи всем членам команды разработки `dev`, но доступны только для чтения членам команды тестировщиков `qa`.
Данный сценарий невозможно реализовать в рамках классической модели (*почему нельзя использовать третий триплет* `others` *для задания прав доступа второй группы* `qa`?).

iii. Аналогичный предыдущему сценарий, но среди тестировщиков имеется один пользователь `power-tester`, наделённый правами на запись файлов некоторой поддиректории `project/unittests/`.
Очевидно, данный сценарий невозможно реализовать даже используя триплет `others` для задания прав доступа группе `qa` (что было бы неверной реализацией в любом случае).

Перечисленные в настоящем пункте сценарии управления доступом слишком сложны для реализации средствами базовой модели Unix - необходимы дополнительные механизмы.

5. Управление журналированием файловых операций

Выполнение файловых операций сопряжено с риском возникновения необработанных ошибок: например, во время длительной операции копирования может произойти перезагрузка системы. Подобные ситуации оставляют файловую систему в некорректном состоянии, нарушая целостность хранимых данных.

Основной способ борьбы с описанными проблемами - использование механизма *журналирования* - логирования файловых операций в специальной структуре, называемой журнал файловой системы.

Журналирование может быть включено для всей системы целиком (посредством опций монтирования FS `data=writeback`, `data=ordered` и `data=journal`), однако логирование всех операций может существенно снизить производительность системы. К тому же далеко не все файлы требуют повышенных гарантий целостности.

В случае необходимости выполнения журналирования операций точечно, для конкретных файлов, необходим специальный механизм.

6. Надёжное удаление

Важнейшей задачей обеспечения конфиденциальности данных является надёжное удаление данных - без возможности восстановления. Базовый режим функционирования ОС не выполняет физическое удаление данных с устройства хранения - изменяются / удаляются лишь структуры, хранящие метаданные.

Для надёжного удаления данных с устройства требуется специальный механизм.

7. Оптимизация производительности файловых операций

Почти любая файловая операция изменяет метаданные целевого файла: размер, время последнего доступа `atime` и др. Изменение метаданных, равно как и содержимого файла, очевидно, влечёт запись обновлённых данных на устройство хранения. Частые изменения атрибутов файлов порождают большое количество дисковых операций, что может негативно сказаться на производительности.

В зависимости от вида обрабатываемых данных (от типа приложения) запись на диск может требовать мгновенного исполнения, либо же допускать буферизацию - выполнение операций пакетно.

В некоторых случаях допускается отключение некоторых из перечисленных выше механизмов для повышения производительности, однако для этого требуется поддержка на уровне файловой системы и ОС.

Приведённые в пп. 5-7 примеры не относятся к задачам, решаемым непосредственно с помощью подсистемы разграничения доступа, однако решение об их внедрении обязано быть согласовано с действующим набором правил доступа, а реализация соответствующих механизмов тесно связана с реализацией модели управления доступом.

2. Linux File Attributes - дополнительные атрибуты файлов

2.1. Дополнительные атрибуты файлов

Ряд файловых систем реализуют специальные режимы выполнения некоторых файловых операций (открытие файла, обновление метаданных и журналирование, запись на устройство хранения и проч.). Управление спец. режимами выполняется посредством дополнительных атрибутов файлов, называемых ***Linux file attributes*** [2, p. 5.5] [4, p. 15.5][8, гл. 8][9].

Дополнительные атрибуты файла (*Linux file attributes*) – тип метаданных файловой системы ОС Linux, описывающих дополнительные свойства файлов и директорий, а также изменяющих поведение и список доступных операций над ними. *File attributes* рассматриваются отдельно от базовых атрибутов, таких как временные метки MAC (`Modification`, `Access`, `Change`), расширения имени файла или права доступа.

По всей видимости, вследствие долгого и нецентрализованного развития файловых систем и Unix-подобных ОС возникла некоторая неточность терминологии:

- основные метаданные называются атрибуты файлов `attributes`;
- дополнительные атрибуты могут называться атрибутами Linux (`Linux file attributes`, `attr`) или расширенными атрибутами (*extended attributes*);
- также существуют расширенные атрибуты (см. раздел Расширенные атрибуты файлов xattr), являющиеся самостоятельным механизмом.

Типичные файловые атрибуты задают:

- видимость файла;
- особый формат хранения данных - является ли файл сжатым и/или зашифрованным;
- возможные операции с файлом - чтение, редактирование, удаление и т.п.;
- действия после удаления файла;
- и др.

Прочие объекты файловой системы, такие как директории, разделы (тома), специальные файлы, также могут иметь определённые атрибуты.

Конкретный список поддерживаемых файловых атрибутов зависит от файловой системы.

Впервые поддержка дополнительных атрибутов файлов появилась в файловой системе ext2. Впоследствии поддержка Linux attributes была реализована и в других файловых системах [4, p. 15.5]: btrfs, ext3, ext4, ReiserFS (с версии Linux 2.4.19), XFS, JFS и ZFS.

Тем не менее механизм дополнительных атрибутов является нестандартным расширением ядра. Как следствие, количество доступных атрибутов несколько различается для разных файловых систем.

К базовому набору файловых атрибутов, поддерживаемых большинством файловых систем, можно отнести следующие:

Table 2. Основные Linux file attributes

Флаг	Атрибут	Константа	Описание
`a`	`append only`	`FS_APPEND_FL`	Файл доступен только для дополнения (запись в конец файла)
`i`	`immutable`	`FS_IMMUTABLE_FL`	Файл не может быть изменён, переименован или удалён
`u`	`nounlink`	—	Файл не может быть удалён, остальное редактирование возможно (только ZFS)

Флаг	Атрибут	Константа	Описание
s	secure deletion	FS_SECRM_FL	Безопасное удаление файла (перезапись секторов диска нулями)
c	compressed	FS_COMPR_FL	Включает сжатие при хранении файла (средствами файловой системы в случае поддержки)
t	no tail	FS_NOTAIL_FL	Выключает "упаковку хвоста" файла при хранении (только ReiserFS)
j	data journaling	FS_JOURNAL_DATA_FL	Включает журналирование операций записи в файл
A	no atime update	FS_NOATIME_FL	Время последнего обращения к файлу (atime) не изменяется
C	no copy on write	FS_NOCOW_FL	Отключает copy-on-write (в случае поддержки COW файловой системой)
d	no-dump	FS_NODUMP_FL	Исключает файл из dump-резервирования
S	sync file updates	FS_SYNC_FL	Синхронная запись файла (отключает буферизацию записи на диск, эквивалент mount sync)
D	sync dir updates	FS_DIRSYNC_FL	Синхронное обновление директории (эквивалент mount sync)

2.2. Управление дополнительными атрибутами: chattr и lsattr, ioctl()

Атрибут может находиться в двух состояниях: установленный (set) либо снятый (cleared).

Программно операции с атрибутами выполняются посредством системной функции ioctl(2) и указания флагов:

- FS_IOC_GETFLAGS - прочитать текущие флаги inode;
- FS_IOC_SETFLAGS - установить флаги inode.

Пример чтения и установления дополнительных атрибутов (см. файл 01_attrs.c)

```
static int get_flags(int fd, unsigned long *attr)) {
    if (ioctl(fd, FS_IOC_GETFLAGS, attr)) < 0) {
        return -1;
    }
    return 0;
}

static int set_flags(int fd, unsigned long attr)) {
    if (ioctl(fd, FS_IOC_SETFLAGS, &attr)) < 0) {
        return -1;
    }
    return 0;
```

```
}
```

Просмотр и изменение файловых атрибутов в терминале выполняется соответственно утилитами lsattr(1) и chattr(1) пакета `e2fsprogs`. Для установки или удаления атрибута с помощью `chattr` указывают маркер атрибута с префиксом `+`, `-` или `=`.

Пример чтения и установления дополнительных атрибутов из командной строки

```
$ lsattr /etc/passwd
$ sudo chattr +i /etc/passwd
$ sudo chattr -i /etc/passwd
$ chattr +A /var/log/my-service.log
$ sudo chattr +a /var/log/auth.log
$ sudo chattr =iae /etc/ssh/sshd_config
```

Необходимые права для изменения дополнительных атрибутов файла представлены в таблице ниже.

Флаг	Атрибут	Константа	Условия доступа
`i`	`immutable`	`FS_IMMUTABLE_FL`	1. `fsuid` процесса равен 0 (запущен от `root`) 2. процесс запущен с `FS_APPEND_FL`
`a`	`append-only`	`FS_APPEND_FL`	1. `fsuid` процесса равен 0 (запущен от `root`) 2. процесс запущен с `CAP_LINUX_IMMUTABLE`
остальные атрибуты			`fsuid` процесса равен идентификатору владельца файла

В основном, в случае установления дополнительных атрибутов на директорию они автоматически наследуются новыми файлами и поддиректориями, создаваемыми внутри данного каталога. Однако некоторые флаги не наследуются [4, p. 15.5]:

- флаг `D, FS_DIRSYNC_FL` можно применять только для директории, поэтому наследуется только поддиректориями, создаваемыми в данном каталоге, не файлами;
- если флаг `i, FS_IMMUTABLE_FL` устанавливается для директории, он не наследуется файлами и поддиректориями, созданными внутри этого каталога, поскольку данный флаг не допускает добавления новых записей в файле директории.

2.3. Примеры использования дополнительных атрибутов

2.3.1. Запрет редактирования файла

Для защиты от случайного или преднамеренного редактирования и удаления важных файлов можно воспользоваться дополнительным атрибутом `i` (`immutable`).

В качестве примера использования `i` защитим важнейшие конфигурационные файлы подсистемы PAM от изменения даже из-под пользователя `root` в Ubuntu 24.04:

```
$ cd /etc/pam.d/
$ sudo lsattr . | grep common  ①
lsattr: Operation not supported While reading flags on ./gdm-smartcard
--------------e------- ./common-session
--------------e------- ./common-password
--------------e------- ./common-auth
--------------e------- ./common-session-noninteractive
--------------e------- ./common-account
$ ls -ld common-password     ②
-rw-r--r-- 1 root root 1893 Nov 30 17:42 common-password
$ sudo nano common-password  ②
...
# password      requisite                       pam_pwquality.so retry=3
password        requisite    pam_pwquality.so retry=7 dcredit=-2 ucredit=-2 ocredit=-2
lcredit=-2 dictcheck=0 minlen=10
...

$ sudo chattr +i common-*      ③
$ sudo lsattr . | grep common  ③
lsattr: Operation not supported While reading flags on ./gdm-smartcard
----i---------e------- ./common-session
----i---------e------- ./common-password
----i---------e------- ./common-auth
----i---------e------- ./common-session-noninteractive
----i---------e------- ./common-account

$ sudo cp common-password /tmp/common-password  ④
$ sudo nano common-password  ⑤
...
[ File 'common-password' is unwritable ]

$ sudo rm common-password  ⑥
$ sudo mv common-password common-password.moved  ⑥
mv: cannot move 'common-password' to 'common-password.moved': Operation not permitted

$ sudo chattr -i common-*  ⑦
$ sudo lsattr . | grep common
$ sudo nano common-password
...
password      requisite                       pam_pwquality.so retry=3
...
```

① В файлах "чистой" системы установлен только атрибут `e` (из-за использования файловой системы ext4)

② Операция редактирования выполняется успешно от `root`

③ Устанавливаем атрибут `i`

④ Операция копирования файла с атрибутом `i` от `root` выполнена успешно

⑤ Попытка редактирования файла даже от `root` завершается с ошибкой - файл защищён от

изменений

⑥ Операции удаления и переименования также недоступны

⑦ После сброса атрибута `i` файл вновь доступен для редактирования от `root`

2.3.2. Разрешения только добавления данных в файл

Для защиты содержимого файла от редактирования можно воспользоваться дополнительным атрибутом `a` (`append-only`).

В качестве примера защитим лог-файл `/var/log/auth.log`, в котором в Debian-based системах сохраняются записи о событиях, связанных с аутентификацией:

```
$ sudo ls -ld /var/log/auth.log
-rw-r----- 1 syslog adm 94949 Nov 30 19:45 /var/log/auth.log
$ sudo lsattr /var/log/auth.log  ①
--------------e------- /var/log/auth.log
$ sudo tail -n 1 /var/log/auth.log
2025-11-30T19:45:24.071040+04:00 OSSec-L2 sudo: pam_unix(sudo:session): session opened
for user root(uid=0) by x(uid=1001)

$ sudo chattr +a /var/log/auth.log  ②
$ sudo lsattr /var/log/auth.log
-----a--------e------- /var/log/auth.log

$ exit
$ ssh x@l4-machine  ③

$ sudo tail -n 23 /var/log/auth.log  ④
2025-11-30T19:45:24.071040+04:00 OSSec-L2 sudo: pam_unix(sudo:session): session opened
for user root(uid=0) by x(uid=1001)
...
2025-11-30T19:47:45.545162+04:00 OSSec-L2 systemd-logind[4185]: New session 18 of user
x.
2025-11-30T19:47:53.194230+04:00 OSSec-L2 sudo:        x : TTY=pts/0 ; PWD=/home/x ;
USER=root ; COMMAND=/usr/bin/tail -n 3 /var/log/auth.log
2025-11-30T19:47:53.194803+04:00 OSSec-L2 sudo: pam_unix(sudo:session): session opened
for user root(uid=0) by x(uid=1001)

$ sudo cp /var/log/auth.log /tmp/auth.log  ⑤
$ sudo rm /var/log/auth.log  ⑥
rm: cannot remove '/var/log/auth.log': Operation not permitted
$ sudo mv /var/log/auth.log /var/log/auth-moved.log  ⑥
mv: cannot move '/var/log/auth.log' to '/var/log/auth-moved.log': Operation not
permitted
$ echo "" | sudo tee /var/log/auth.log  ⑥
tee: /var/log/auth.log: Operation not permitted

$ echo "hehehe" | sudo tee -a /var/log/auth.log  ⑦
hehehe
```

```
$ sudo tail -n 23 /var/log/auth.log
2025-11-30T20:04:31.141481+04:00 OSSec-L2 sudo:        x : TTY=pts/0 ; PWD=/home/x ;
USER=root ; COMMAND=/usr/bin/tee -a /var/log/auth.log
2025-11-30T20:04:31.141971+04:00 OSSec-L2 sudo: pam_unix(sudo:session): session opened
for user root(uid=0) by x(uid=1001)
hehehe  ⑦
2025-11-30T20:04:31.143958+04:00 OSSec-L2 sudo: pam_unix(sudo:session): session closed
for user root
2025-11-30T20:04:38.597247+04:00 OSSec-L2 sudo:        x : TTY=pts/0 ; PWD=/home/x ;
USER=root ; COMMAND=/usr/bin/tail -n100 /var/log/auth.log
2025-11-30T20:04:38.597775+04:00 OSSec-L2 sudo: pam_unix(sudo:session): session opened
for user root(uid=0) by x(uid=1001)
```

① В "чистой" системе у файла `/var/log/auth.log` атрибут `a` по умолчанию не установлен

② Устанавливаем атрибут `a`

③ Выполняем новый логин в систему - в логе `auth.log` должны появиться новые записи

④ В логе `auth.log` действительно появились записи о новом логине пользователя `x`

⑤ Операция копирования файла с атрибутом `a` от `root` выполнена успешно

⑥ Операции удаления, переименования и редактирования (содержимого файла) даже от `root` завершается с ошибкой

⑦ Операция дописывания в конец файла также выполняется успешно от имен `root` вручную

2.3.3. Запрет удаления и создания новых файлов в директории

Для защиты хранящихся в некоторой директории файлов и папок от удаления можно воспользоваться дополнительным атрибутом `a` (`append-only`), установленным на целевую директорию. Более того, в случае необходимости зафиксировать структуру директории (запретить также создание новых файлов) поставленную задачу можно решить с помощью дополнительного атрибута `i` (`immutable`).

```
$ mkdir -p /tmp/experiments
$ sudo chgrp user /tmp/experiments  ①
$ cd /tmp/experiments
$ ls -la
total 8
drwxrwxr-x  2 x    user 4096 Nov 30 20:34 .
drwxrwxrwt 20 root root 4096 Nov 30 20:34

$ touch 1.txt                           ②
$ mkdir 1
$ echo "Hello" | sudo -u user tee 2.txt
Hello
$ sudo -u user mkdir 2
$ ls -la                                ②
total 20
drwxrwxr-x  4 x    user 4096 Nov 30 20:35 .
```

```
drwxrwxrwt 20 root root 4096 Nov 30 20:34 ..
drwxrwxr-x  2 x    x    4096 Nov 30 20:35 1
-rw-rw-r--  1 x    x       0 Nov 30 20:35 1.txt
drwxrwxr-x  2 user user 4096 Nov 30 20:35 2
-rw-rw-r--  1 user user    6 Nov 30 20:35 2.txt

$ sudo chattr +a /tmp/experiments  ③
$ sudo lsattr /tmp | grep experiments
-----a--------e------- /tmp/experiments

$ rm -f 1.txt  ④
rm: cannot remove '1.txt': Operation not permitted
$ sudo -u user rm -rf 2
rm: cannot remove '2': Operation not permitted
$ sudo rm -rf *
rm: cannot remove '1': Operation not permitted
rm: cannot remove '1.txt': Operation not permitted
rm: cannot remove '2': Operation not permitted
rm: cannot remove '2.txt': Operation not permitted

$ touch new.txt  ⑤
$ sudo -u user mkdir new
$ ls -la | grep new
drwxrwxr-x  2 user user 4096 Nov 30 20:37 new
-rw-rw-r--  1 x    x       0 Nov 30 20:37 new.txt

$ cd /
$ sudo rm -rf /tmp/experiments  ⑥
rm: cannot remove '/tmp/experiments/new': Operation not permitted
rm: cannot remove '/tmp/experiments/new.txt': Operation not permitted
rm: cannot remove '/tmp/experiments/2.txt': Operation not permitted
rm: cannot remove '/tmp/experiments/1': Operation not permitted
rm: cannot remove '/tmp/experiments/1.txt': Operation not permitted
rm: cannot remove '/tmp/experiments/2': Operation not permitted

$ sudo chattr -a+i /tmp/experiments  ⑦
$ sudo lsattr /tmp | grep experiments
----i---------e------- /tmp/experiments
$ touch /tmp/experiments/newrer.txt  ⑧
touch: cannot touch '/tmp/experiments/newrer.txt': Operation not permitted
$ sudo -u user mkdir /tmp/experiments/newer
mkdir: cannot create directory □/tmp/experiments/newer□: Operation not permitted
$ sudo mkdir /tmp/experiments/newer
mkdir: cannot create directory □/tmp/experiments/newer□: Operation not permitted

$ sudo chattr =e /tmp/experiments  ⑨
$ sudo lsattr /tmp | grep experiments
--------------e------- /tmp/experiments
$ sudo rm -rf /tmp/experiments  ⑨
$ sudo ls -la /tmp/experiments
```

```
ls: cannot access '/tmp/experiments': No such file or directory
```

① Добавляем права на создание файлов в директории пользователю `user`

② Создаём файлы и папки в директории от разных пользователей

③ Устанавливаем атрибут `a` на директорию `/tmp/experimets`

④ Удаление файлов и директорий владельцем директории `/tmp/experimets` и даже от `root` завершается с ошибкой

⑤ Создание новых файлов завершается успешно для всех пользователей, имеющих соответствующие права

⑥ Удаление директории с установленным атрибутом `a` невозможно

⑦ Устанавливаем атрибут `i` на директорию `/tmp/experimets`

⑧ Теперь и создание новых файлов и директорий запрещено

⑨ Сбросив атрибуты `a` и `i`, директорию `/tmp/experimets` вновь можно удалять
N.B. Обратите внимание на формат команды `chattr =e`

2.3.4. Надёжное удаление содержимого файла

Согласно документации утилит `lsattr()` и `chattr()`, множеству мануалов и статей в Интернете и книгах, для включения опции безопасного удаления файла требуется установить дополнительный атрибут `s` (`secure`). Тогда в случае удаления файла выделенные под хранение его содержимого блоки дискового устройства будут перезаписаны нулями.

Также отмечается, что для большинства современных файловых систем это поведение не реализовано (в частности, для ext4, btrfs, xfs, zfs и др.). Более того, единственная поддерживающая функцию `secure delete` файловая система - ext2.

Проверим заявленное поведение.

```
$ dd if=/dev/zero of=/tmp/ext2.img bs=1M count=64  ①
64+0 records in
64+0 records out
67108864 bytes (67 MB, 64 MiB) copied, 0.0501665 s, 1.3 GB/s

$ LOOP=$(sudo losetup --show -f /tmp/ext2.img)  ①
$ echo $LOOP
/dev/loop15
$ sudo mkfs.ext2 $LOOP  ①
mke2fs 1.47.0 (5-Feb-2023)
Discarding device blocks: done
Creating filesystem with 16384 4k blocks and 16384 inodes

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

$ sudo file -sL $LOOP
```

```
/dev/loop15: Linux rev 1.0 ext2 filesystem data, UUID=e2a4bebb-59b1-4b9f-b845-
447bfa0a00b2 (large files)

$ sudo mount "$LOOP" /mnt  ①
$ echo "Hello, secure deletion via attr s!" | sudo tee /mnt/file.txt  ②
Hello, secure deletion via attr s!
$ sync

$ OFFSET=$(sudo debugfs -R "stat file.txt" "$LOOP" 2>/dev/null | grep 'BLOCKS:' -A1 |
tail -n1 | cut -d ':' -f 2)  ③
$ BS=$(sudo dumpe2fs "$LOOP" 2>/dev/null | grep "Block size" | awk '{print $3}')
$ echo "Blocks offset: $OFFSET"
Blocks offset: 1536
$ echo "Block size: $BS"
Block size: 4096

$ sudo dd if="$LOOP" bs=$BS skip=$OFFSET count=1 | xxd -l 32         ③
00000000: 4865 6c6c 6f2c 2073 6563 7572 6520 6465  Hello, secure de
00000010: 6c65 7469 6f6e 2076 6961 2061 7474 7220  letion via attr
00000020: 7321 0a00 0000 0000 0000 0000 0000 0000  s!..............
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 3.8439e-05 s, 107 MB/s

$ sudo chattr +s /mnt/file.txt  ④
$ sudo rm -f /mnt/file.txt
$ sudo sync
$ sudo dd if="$LOOP" bs=$BS skip=$OFFSET count=1 | xxd -l 32         ⑤
00000000: 4865 6c6c 6f2c 2073 6563 7572 6520 6465  Hello, secure de
00000010: 6c65 7469 6f6e 2076 6961 2061 7474 7220  letion via attr
00000020: 7321 0a00 0000 0000 0000 0000 0000 0000  s!..............
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 3.8439e-05 s, 107 MB/s

$ sudo umount /mnt  ⑥
$ sudo losetup -d $LOOP
```

① Создаём файл размером 64MiB, с помощью `losetup` превращаем файл в дисковое устройство `$LOOP`, создаём файловую систему ext2 на этом устройстве и монтируем в `/mnt/`

② Создаём файл `/mnt/file.txt`

③ Ищем расположение содержимого файла `/mnt/file.txt` на диске `$LOOP` и проверяем найденный офсет, читая данные непосредственно с устройства

④ Устанавливаем атрибут `s` и удаляем файл

⑤ Проверяем хранимые на диске данные - видим, что данные не перезаписаны нулями!

⑥ Зачистка: выполняем размонтирование файловой системы и удаление дискового

устройства

Очевидно, заявленное поведение не реализовано и для файловой системы ext2!

На самом деле опция `secure remove` не была реализована ни в одной стабильной версии ядра Linux!

Краткий исторический экскурс:

1. Концепция безопасного удаления содержимого файла появилась во времена широкого распространения ext2, в конце 1990-х - начале 2000-х. В то же время в исходном коде ext2 появились определения флагов `EXT2_SECRM_FL` (`secure remove`) и `EXT2_UNRM_FL` (`undelete`).
2. Однако логика так и не была **реализована** - ни в одной стабильной версии. Флаги были определены в коде, утилиты `chattr` и `lsattr` умели (и умеют до сих пор) устанавливать флаг, но это не оказывает никакого эффекта.
   Существовали патчи и модули ядра (Bauer, Priyantha, van Hauser, Openwall Project, Joukov et al., Corbet), реализовывавшие этот механизм.
3. Тем не менее, ни в одной стабильной версии основных дистрибутивов Linux (Debian-based, RH-based, openSUSE, Arch Linux, Gentoo, etc.) эти патчи не использовались.
4. Примерно во время перехода ядра от версии 2.4 к 2.6 функционал окончательно признан де-факто устаревшим.

Среди причин, обусловивших такое решение, можно выделить:

1. журналирование файловых операций и дисковое кэширование / буферизация не позволяют гарантировать удаление данных на диске;
2. надёжное удаление данных с магнитных дисков (HDD) требует многократной перезаписи (см. алгоритм Гутмана]);
3. эффекты переназначения (remapping) секторов диска и SSD wear-levelling также не позволяют гарантировать уничтожение данных.

Для надёжного удаления данных требуется использовать специальные инструменты (`shred`, `wipe`, `srm`), а также применять полнодисковое шифрование средствами LUKS.

Однако, скорее всего, единственно надёжным методом является физическое уничтожение носителя.

2.4. Реализация механизма дополнительных атрибутов файла

TODO

2.5. Ограничения механизма дополнительных атрибутов файла

TODO

3. Расширенные атрибуты файлов xattr

3.1. О расширенных атрибутах файлов

Расширенные атрибуты (Extended Attributes, **xattr**) были впервые реализованы в файловых системах BSD и IRIX в 1990-х годах как механизм хранения дополнительных метаданных, не вписывающихся в классическую Unix-модель прав доступа `rwx`.

В Linux механизм `xattr` появился в конце 1990-х вместе с поддержкой ACL (Access Control Lists). Позже `xattr` стал основой для таких систем безопасности, как **SELinux**, **Smack**, **AppArmor**, а также для хранения **цифровых подписей** и **метаданных целостности файлов**.

Классическая модель разграничения доступа Unix (`rwx` для user/group/others) оказалась недостаточной, чтобы реализовать:

- многоуровневые политики доступа (DAC + RBAC + ABAC + MAC);
- разграничение полномочий `root`;
- хранение контекстной информации о безопасности.

Механизм `xattr` предоставляет возможность ассоциировать с файлом **произвольные пары "ключ-значение"**, которые могут интерпретироваться ядром, модулями LSM (Linux Security Modules) или пользовательскими программами.

3.2. Общие сведения о расширенных атрибутах

Расширенные атрибуты — это ассоциированные с файлом метаданные, хранящиеся отдельно от основных метаданных из `inode`.

`xarrt` не отображаются стандартными утилитами (`ls`, `stat`), операции с ними осуществляются посредством специальных системных вызовов.

Каждый атрибут имеет:

- **имя (name)** — строка, включающая пространство имён;
- **значение (value)** — произвольный двоичный блок данных (до нескольких килобайт);
- **пространство имён (namespace)** — определяет политику доступа и предназначение.

Пространства имён:

Пространство имён	Пример	Доступ	Назначение
`user.`	`user.comment`	Пользователь, владелец файла	Хранение пользовательских метаданных
`trusted.`	`trusted.md5`	Только `root` (`CAP_SYS_ADMIN`)	Для системных демонов и служб
`security.`	`security.selinux`, `security.capability`	LSM и `root` (`CAP_SYS_ADMIN`)	Метки SELinux, Smack, Capability процессов
`system.`	`system.posix_acl_access` `system.posix_acl_default`	Только ядро и системные вызовы	ACL, POSIX атрибуты

3.3. Управление расширенными атрибутами

```
$ sudo apt install attr
```

```
$ touch 1.txt
$ getfattr -d 1.txt

$ setfattr -n user.mylabel -v "User" 1.txt
$ sudo setfattr -n trusted.mylabel -v "Trusted" 1.txt
$ sudo setfattr -n security.mylabel -v "Security" 1.txt
$ getfattr -d 1.txt
# file: 1.txt
user.mylabel="User"

$ sudo getfattr -d 1.txt
# file: 1.txt
user.mylabel="User"

$ sudo getfattr -d -n trusted.mylabel  1.txt
# file: 1.txt
trusted.mylabel="Trusted"

$ sudo getfattr -d -n security.mylabel 1.txt
# file: 1.txt
security.mylabel="Security"


$ sudo setfattr -x security.mylabel 1.txt
$ sudo getfattr -d -n security.mylabel 1.txt
1.txt: security.mylabel: No such attribute
```

3.4. Применение в подсистемах безопасности

3.4.1. SELinux

SELinux использует `security.selinux` для хранения контекста безопасности (например, `system_u:object_r:etc_t:s0`).

При каждом обращении к файлу ядро проверяет контекст против политики.

3.4.2. Capabilities

Расширенные атрибуты `security.capability` хранят привилегии (Capabilities) процессов, применяемые к исполняемым файлам.

```
$ cp /bin/ls ls-copy
$ sudo setcap cap_net_bind_service=+ep ./ls-copy
$ getfattr -n security.capability ./ls-copy
```

3.5. Контроль целостности посредством подписей

Подсистемы IMA и EVM используют `xattr` `security.ima` и `security.evm` для хранения хешей и подписей файлов.

4. Списки контроля доступа

4.1. О списках контроля доступа

Трудная история ACL в *nix...

4.2. Организация списков контроля доступа

POSIX ACL являются в основном прямым расширением стандартной 9-битной модели разрешений UNIX. ACL-список представляет собой ряд записей, каждая из которых задает права доступа к файлу для отдельного пользователя или группы.

4.2.1. Формат записи ACL-списка

Список контроля доступа образован записями, состоящими из трёх частей [4, p. 17.1][10]:

1. `тип тега` (`tag-type`) - определяет (прямо или косвенно) тип субъекта и область применимости правила: пользователь, группа, маска, остальные пользователи;
2. `тег-квалификатор` (`tag-qualifier`) - идентифицирует пользователя или группу по имени или числовому ID (*необязательная* часть, присутствует только для записей с явно заданным пользователем или группой);

3. `права доступа` (`permissions`) - триплет прав `rwx`, назначаемых субъекту(-ам) доступа, обозначенным тегом.

Table 3. Пример списка контроля доступа

Тип тега	Тег-квалификатор	Права доступа	Текстовая форма (`setfacl`/`getfacl`)
`ACL_USER_OBJ`	—	`rwx`	`user::rwx`
`ACL_USER`	`1000`	`r-x`	`user:alice:r-x`
`ACL_USER`	`1001`	`--x`	`user:1001:--x`
`ACL_GROUP_OBJ`	—	`r-x`	`group::r-x`
`ACL_GROUP`	`1003`	`r--`	`group:1003:r--`
`ACL_GROUP`	`1002`	`rwx`	`group:devs:rwx`
`ACL_MASK`	—	`r-x`	`mask::r-x`
`ACL_OTHER`	—	`r--`	`other::r--`

Тип тега имеет одно из следующих значений [4, p. 17.1][10].

- `ACL_USER_OBJ` - запись определяет права доступа владельца файла. Каждый ACL-список содержит ровно одну такую запись. Она соответствует традиционным правам доступа владельца (пользователя).
- `ACL_USER` - запись определяет права доступа, предоставляемые пользователю, который идентифицируется тегом-квалификатором. ACL-список может содержать от нуля до нескольких записей `ACL_USER`, однако для конкретного пользователя можно определить не более одной такой записи.
- `ACL_GROUP_OBJ` - запись определяет права доступа группы владельцев файла. Каждый ACL-список содержит ровно одну такую запись. Она соответствует традиционным правам доступа группы (если только список не содержит также запись `ACL_MASK`).
- `ACL_GROUP` - запись определяет права доступа, предоставляемые группе, которая идентифицируется тегом-квалификатором. ACL-список может содержать от нуля до нескольких записей `ACL_GROUP`, однако для конкретной группы можно определить не более одной такой записи.
- `ACL_MASK` - запись определяет максимальные права доступа, которые могут быть предоставлены записями `ACL_USER`, `ACL_GROUP_OBJ` и `ACL_GROUP`. ACL-список содержит не более одной записи `ACL_MASK`. Если он содержит запись `ACL_USER` или `ACL_GROUP`, то наличие записи `ACL_MASK` является обязательным.
- `ACL_OTHER` - запись определяет права доступа, предоставляемые пользователям, не соответствующим другим записям в списке контроля доступа. Каждый ACL-список содержит ровно одну такую запись. Она соответствует традиционным правам доступа для остальных.

Тег-квалификатор используется только для записей `ACL_USER` и `ACL_GROUP`, т.к. только для этих записей необходимо задавать идентификатор - пользователя или группы, соответственно.

4.2.2. Текстовый формат правил списка контроля доступа

Текстовый формат представления правил ACL-списка применяется в первую очередь при работе с утилитами getfacl(1) и setfacl(1), а также с некоторыми библиотечными функциями.

Определены два текстовых формата:

1. *длинная текстовая форма* содержит по одной записи ACL на строку и может также включать комментарии, которые начинаются с символа # и продолжаются до конца строки;
2. *краткая текстовая форма* образована последовательностью правил, разделённых запятыми.

В обеих формах каждая запись образована тремя частями, разделёнными двоеточиями:

```
tag-type:[tag-qualifier]:permissions
```

где

- `tag-type` - определяет тип тега с помощью одной из строк: `user`/`u`, `group`/`g`, `mask`/`m` или `other`/`o`;
- `tag-qualifier` - идентифицирует пользователя или группу по имени или числовому ID;
- `permissions` - строка, образованная символами `r`, `w`, `x` и `-`.

В блоке `permissions` можно указывать только те права, которые выдаются соответствующему субъекту или маске.

В таблице Пример списка контроля доступа строковый формат представления правил показан в последнем столбце. Заданный в таблице набор правил в длинной текстовой форме может выглядеть примерно так:

```
user::rwx
user:alice:r-x
user:bob:--x
group::r-x
group:qa:r--
group:dev:rwx
mask::r-x
other::r--
```

4.2.3. Минимальный и расширенный списки контроля доступа

Всевозможные списки контроля доступа выделяют два вида [2, p. 5.6][4, p. 17.1]:

1. *Минимальный* - семантически эквивалентен обычному набору триплетов прав; такой список содержит только три записи `ACL_USER_OBJ`, `ACL_GROUP_OBJ` и `ACL_OTHER`.
2. *Расширенный* - минимальный список, дополненный правилами `ACL_USER`, `ACL_GROUP` и

`ACL_MASK`.

Различие минимального и расширенного ACL-списков обусловлено тем, что последний обеспечивает семантическое расширение классической модели прав доступа Unix.

Минимальный ACL хранится в виде триплетов прав - в поле `i_mode` структуры `inode` (см. Реализация модели Unix в ядре Linux). В расширенном списке задаются дополнительные правила, хранящиеся как расширенные атрибуты (см. Расширенные атрибуты файлов xattr), называющиеся `system.posix_acl_access`.

4.3. Проверка прав доступа

4.3.1. Алгоритм проверки прав доступа

Проверка прав доступа к файлу с ассоциированным списком контроля доступа аналогичен алгоритму проверки классической модели (см. Алгоритм проверки прав доступа).

Принципиально, алгоритм проверки прав доступа выглядит следующим образом [4, p. 17.2][10].

1. Если процесс является привилегированным (запущен от `root` или имеет необходимые Capabilities), предоставляется полный доступ.

Если привилегированный процесс запускает исполняемый файл, требуется наличие права на исполнение хотя бы в одном из правил ACL-списка!

2. Выполняется сравнение `FSUID` процесса с идентификатором владельца файла:
   - если идентификаторы совпадают, предоставляются права доступа, указанные в записи `ACL_USER_OBJ`, **проверка завершается**;
   - иначе переход к шагу 3.
3. Выполняется последовательный перебор записей `ACL_USER` и сравнение `FSUID` процесса с тегом-квалификатором записи:
   - если идентификаторы совпадают, предоставляются *маскированные* права `permissions`, указанные в этой записи - `permissions & ACL_MASK`, **проверка завершается**;
   - при несовпадении переход к следующей записи `ACL_USER`.

   Если ни одна запись `ACL_USER` не подошла, переход к шагу 4.
4. Выполняется сравнение идентификатора группы владельцев файла с одним из идентификаторов группы процесса (`FSGID` или дополнительной группы):
   - если в списке групп процесса содержится идентификатор группы файла, предоставляются *маскированные* права `permissions`, указанные в записи `ACL_GROUP_OBJ` - `permissions & ACL_MASK`, **проверка завершается**; * иначе переход к шагу 5.
5. Выполняется последовательный перебор записей `ACL_GROUP` и поиск тега-квалификатора записи в списке групп процесса:

- если тег-квалификатор записи содержится в списке групп процесса, предоставляются *маскированные* права `permissions`, указанные в этой записи `ACL_GROUP` - `permissions & ACL_MASK`, **проверка завершается**; * иначе переход к следующей записи `ACL_GROUP`.

Если ни одна запись `ACL_GROUP` не подошла, переход к шагу 6.

6. В остальных случаях процессу предоставляются права доступа, указанные в записи `ACL_OTHER`.

4.3.2. Примеры использования списков контроля доступа

В представленных в настоящем разделе примерах для работы со списками контроля доступа используются команды:

- `getfacl file` для чтения ACL-списка, ассоциированного с файлом `file`;
- `setfacl -m <ace> file` для добавления или изменения правила, где `ace` - правило в текстовой форме.

1. Минимальный список контроля доступа

```
$ touch 1.txt
$ ls -l 1.txt  ①
-rw-rw-r-- 1 x x 0 Dec  1 18:33 1.txt
$ getfacl 1.txt  ②
# file: 1.txt
# owner: x
# group: x
user::rw-
group::rw-
other::r--

$ getfattr -d -m "system.posix_acl_*" 1.txt  ③
$
```

① Созданный файл `1.txt` принадлежит `x`:`x` с правами `0664`

② `getfacl` выводит ACL-список в длинной текстовой форме; в списке три базовых правила, соответствующих триплетам прав

③ У файла `1.txt` не установлен расширенный атрибут `system.posix_acl_access` - минимальный ACL хранится в `inode`

2. Расширенный список контроля доступа

```
$ touch 2.txt
$ ls -l 2.txt  ①
-rw-rw-r-- 1 x x 0 Dec  1 18:49 2.txt
$ getfacl --omit-header 2.txt  ②
user::rw-
group::rw-
other::r--
```

```
$ setfacl -m user:user:rx 2.txt  ③
$ getfacl --omit-header 2.txt  ④
user::rw-
user:user:r-x  ④
group::rw-
mask::rwx ④
other::r--

$ getfattr -d -m "system.posix_acl_*" 2.txt  ⑤
# file: 2.txt
system.posix_acl_access=0sAgAAAAEABgD/////AgAFAOgDAAAEAAYA/////xAABwD/////IAAEAP////8=
```

① Созданный файл `2.txt` принадлежит `x:x` с правами `0664`

② `getfacl` выводит ACL-список - в данный момент это минимальный список, соответствующий триплетам прав

③ С помощью утилиты `setfacl` добавляем правило для пользователя `user` с правами `r-x`
N.B. При задании прав можно указывать только те, которые выдаются субъекту доступа

④ `getfacl` выводит ACL-список - теперь это расширенный список; появилось правило для `user` и маска (т.к. добавлено правило `ACL_USER`)!
N.B. Аргумент `--omit-header` сокращает вывод ACL-списка, убирая заголовок в начале

⑤ У файла `2.txt` установлен расширенный атрибут `system.posix_acl_access`, т.к. теперь ACL-список является расширенным

3. Привилегированный процесс не может запустить исполняемый файл без `x`

```
$ printf '#include <stdio.h>\n\nint main() {\n    printf("Hello!\\n");\n    return
0;\n}\n' > 3.c
$ gcc -o 3 3.c
$ ./3  ①
Hello!

$ chmod ugo-x 3  ②
$ ls -l 3
-rw-rw-r-- 1 x x 15960 Dec  1 19:12 3
$ sudo ./3
sudo: ./3: command not found

$ setfacl -m g:users:--x 3  ③
$ getfacl --omit-header 3
user::rw-
group::rw-
group:users:--x
mask::rwx
other::r--

$ ./3  ④
-bash: ./3: Permission denied
$ sudo ./3  ⑤
```

```
Hello!
```

① Создаём C-программу, компилируем и запускаем - процесс исполняется

② Без права на исполнение `x` программа не запускается даже от `root`

③ Добавляем правило для группы `users` с правами `--x`

④ Хотя `x` состоит в группе `users`, запуск от пользователя `x` недоступен (*почему?*)

⑤ Запуск от `root` возможен - в ACL-списке есть хотя бы одно правило, где выставлен бит `x`

4. Права доступа владельцу файла задаются в правиле `ACL_USER_OBJ`

Права доступа владельца определяются правилом `ACL_USER_OBJ` (`user::***`):

```
$ echo "Hello" > 4.txt  ①
$ setfacl -m u:user:r 4.txt
$ chmod u-r 4.txt
$ ls -l 4.txt
--w-rw-r--+ 1 x x 6 Dec  1 19:35 4.txt  ②
$ getfacl --omit-header 4.txt
user::-w-
user:user:r--
group::rw-
mask::rw-
other::r--

$ cat 4.txt  ③
cat: 4.txt: Permission denied

$ chmod u+r 4.txt
$ getfacl --omit-header 4.txt
user::rw-
user:user:r--
group::rw-
mask::rw-
other::r--

$ cat 4.txt  ④
Hello
```

① Создаём файл `4.txt` с правилом для `user` и правами на чтение, запрещаем чтение владельцу `x`

② Факт наличия установленного на файл ACL отражается в выводе `ls` с помощью символа `+` в конце блока прав

③ Владелец `x` не может прочитать содержимое собственного файла

④ После возвращения права на чтение пользователь `x` может читать содержимое файла `4.txt`

Обратите внимание: проверка прав доступа для владельца файла

> завершается на правиле ACL_USER_OBJ. В противном случае первая операция чтения завершилась бы успехом, т.к. права на чтение выставлены для всех остальных пользователей системы (см. первый ACL-список).

Причём при проверке прав доступа для владельца биты маски (ACL_MASK) не учитываются:

```
$ setfacl -m mask::--x 4.txt  ①
$ getfacl --omit-header 4.txt
user::rw-
user:user:r--                   #effective:---  ③
group::rw-                      #effective:---  ③
mask::--x  ②
other::r--

$ cat 4.txt  ④
Hello
$ sudo -u user cat 4.txt  ⑤
cat: 4.txt: Permission denied
```

① Устанавливаем маску --x - без права на чтение

② Новое значение маски установлено

③ getfacl заботливо вычисляет т.н. *эффективные права* для различных пользователей и групп (permissions & ACL_MASK) - в данном примере у пользователя user и группы владельцев прав не осталось

④ Владелец файла может читать содержимое файла - маска не учитывается для правила ACL_USER_OBJ

⑤ Чтение от пользователя user не разрешено - маска учитывается для правил ACL_USER

5. Права доступа именованного пользователя задаются в правиле ACL_USER

Права доступа конкретного пользователя user в случае наличия правила ACL_USER (user:user:***) с тегом-квалификатором, соответствующим идентификатору пользователя user, определяются заданными в правиле правами:

```
$ echo "Hello" > 5.txt  ①
$ chmod ugo-r 5.txt
$ setfacl -m u:user:r 5.txt
$ ls -l 5.txt
--w-rw----+ 1 x x 6 Dec  1 20:04 5.txt
$ getfacl --omit-header 5.txt
user::-w-
user:user:r--
group::-w-
mask::rw-
other::---

$ sudo -u x cat 5.txt  ②
cat: 5.txt: Permission denied
```

```
$ sudo -u user cat 5.txt  ③
Hello
$ sudo -u user2 cat 5.txt  ④
cat: 5.txt: Permission denied
```

① Создаём файл `5.txt` с правилом для `user` и правами на чтение, запрещаем чтение всем остальным пользователям (в том числе владельцу `x`)

② Чтение от пользователя `x` запрещено правилом `ACL_USER_OBJ` (`-w-`)

③ Чтение от пользователя `user` разрешено правилом `ACL_USER` (`r-- & rwx`)

④ Чтение от пользователя `user2` запрещено правилом `ACL_OTHER` (`--- & rwx`)

При этом при проверке прав доступа для `user` биты маски (`ACL_MASK`) **учитываются**:

```
$ setfacl -m mask::--x 5.txt  ①
$ getfacl --omit-header 5.txt
user::-w-
user:user:r--                   #effective:---  ②
group::-w-                      #effective:---
mask::--x
other::---

$ sudo -u user cat 5.txt  ③
cat: 5.txt: Permission denied
```

① Устанавливаем маску `--x` - без права на чтение

② *Эффективные права* для пользователя `user` - маска делает файл недоступным для чтения (`permissions & ACL_MASK` = `r-- & --x`)

③ Чтение от пользователя `user` действительно не разрешено

6. Права доступа группе владельцев файла задаются в правиле `ACL_GROUP_OBJ`

Права доступа для группы владельцев определяются правилом `ACL_GROUP_OBJ` (`group::***`):

```
$ echo "Hello" > 6.txt  ①
$ sudo chown user2 6.txt  ②
$ sudo setfacl -m group:nogroup:-wx 6.txt  ③
$ getfacl 6.txt  ④
# file: 6.txt
# owner: user2
# group: x
user::rw-
group::rw-
group:nogroup:-wx
mask::rwx
other::r--

$ id -gn
x  ⑤
```

```
$ cat 6.txt  ⑥
Hello

$ sudo -g nogroup groups
nogroup users x  ⑦
$ sudo -g nogroup cat 6.txt  ⑧
Hello
```

① Создаём файл `6.txt`

② Изменяем владельца файла на `user2` (чтобы правила `ACL_USER*` не участвовали в дальнейших проверках прав)

③ Создаём правило, запрещающее чтение файла для группы `nogroup`

④ Группа владельцев файла `6.txt` - `x`, правила для группы `x` дают права на чтение, для `nogroup` - не дают права на чтение, в маске установлены все права (т.е. маска не влияет на права)

⑤ Первичная группа процесса, запускаемого от пользователя `x` - `x` (т. е. её идентификатор записывается в `FSGID` процесса)

⑥ Т.к. владелец файла `6.txt` не `x`, а в ACL-списке отсутствуют правило `ACL_USER` для `x`, проверка прав доступа выполняется по группе владельцев - по правилу `ACL_GROUP_OBJ` (`rw- & rwx`), поэтому доступ разрешён

⑦ Первичная группа запускаемого с помощью `sudo -g nogroup` процесса - `nogroup`, при этом список остальных групп процесса содержит все группы пользователя `x`, в частности группу `x`

⑧ В списке групп запущенного процесса имеется группа `x`, являющаяся группой владельцев файла `6.txt`, поэтому проверка прав выполняется по правилу `ACL_GROUP_OBJ` (`rw- & rwx`)

Обратите внимание на последнюю команду предыдущего примера.

Несмотря на то, что в ACL-списке содержится правило `ACL_GROUP` для группы `nogroup`, Алгоритм проверки прав предписывает выполнить поиск группы владельцев файла `6.txt` (группы `x`) в списке **всех** групп запущенного процесса.

Группа `x` действительно содержится в списке групп процесса (пусть и не является его `FSGID`), поэтому проверка прав доступа выполняется по правилу для группы владельцев файла `ACL_GROUP_OBJ`.

При этом при проверке прав доступа для группы владельцев биты маски (`ACL_MASK`) **учитываются**:

```
$ sudo setfacl -m mask::--- 6.txt  ①
$ getfacl 6.txt
# file: 6.txt
# owner: user2
# group: x
user::rw-
```

```
group::rw-                      #effective:---
group:nogroup:-wx               #effective:---  ②
mask::---
other::r--

$ id -gn
x
$ cat 6.txt  ③
cat: 6.txt: Permission denied
```

① Устанавливаем маску `---` - без права на чтение

② *Эффективные права* для группы владельцев `x` - маска делает файл недоступным для чтения (`permissions & ACL_MASK` = `-wx & rwx`)

③ Чтение от группы пользователей `x` действительно не разрешено

7. Права доступа именованной группе файла задаются в правиле `ACL_GROUP`

Права доступа конкретной группе пользователей `group` в случае наличия правила `ACL_GROUP` (`group:group:***`) с тегом-квалификатором, соответствующим идентификатору группы `group`, определяются заданными в правиле правами:

```
$ echo "Hello" > 7.txt  ①
$ sudo chown user2:user2 7.txt  ②
$ sudo chmod 0000 7.txt  ③
$ sudo setfacl -m group:x:rwx 7.txt  ④
$ getfacl 7.txt  ⑤
# file: 7.txt
# owner: user2
# group: user2
user::---
group::---
group:x:rwx
mask::rwx
other::---

$ groups  ⑥
x users
$ cat 7.txt
Hello

$ sudo -g nogroup groups  ⑦
nogroup users x
$ sudo -g nogroup cat 7.txt
Hello

$ sudo -u nobody -g nogroup groups  ⑧
nogroup
$ sudo -u nobody -g nogroup cat 7.txt
cat: 7.txt: Permission denied
```

① Создаём файл `7.txt`

② Изменяем владельца и группу файла на `user2` (чтобы правила `ACL_USER*` и `ACL_GROUP_OBJ` не участвовали в дальнейших проверках прав)

③ Запрещаем доступ все пользователям системы

④ Создаём правило, предоставляющее полные права доступа для группы `x`

⑤ Права доступа к `7.txt` имеются только у группы пользователей `x`

⑥ Т.к. владелец и группа владельцев файла `7.txt` не `x`, а в ACL-списке содержится правило `ACL_GROUP` с тегом-квалификатором, равным первичной группе `x`, проверки выполняются по правилу `ACL_GROUP` (`rwx & rwx`), поэтому доступ разрешён

⑦ Аналогично предыдущему, в списке групп процесса содержится группа `x`, описанная в правиле `ACL_GROUP`, поэтому доступ разрешён

⑧ В списке групп запущенного с помощью `sudo -u nobody -g nogroup` процесса содержится всего одна группа `nogroup`, для которой в ACL-списке нет правил `ACL_GROUP` - проверки выполняются по правилу `ACL_OTHER`, поэтому в доступе отказано

При этом при проверке прав доступа для группы `x` биты маски (`ACL_MASK`) **учитываются**:

```
$ sudo setfacl -m mask::--- 7.txt  ①
$ getfacl 7.txt
# file: 7.txt
# owner: user2
# group: user2
user::---
group::---
group:x:rwx                     #effective:---  ②
mask::---
other::---

$ groups
x users
$ cat 7.txt  ③
cat: 7.txt: Permission denied
```

① Устанавливаем маску `---` - без права на чтение

② *Эффективные права* для группы `x` - маска делает файл недоступным для чтения (`permissions & ACL_MASK` = `rwx & ---`)

③ Чтение от группы пользователей `x` действительно не разрешено

8. Права доступа могут предоставляться разными правилами ACL-списка

Права могут быть выданы разными правилами:

```
$ sudo adduser johnny-three-groups
$ sudo usermod -aG user johnny-three-groups
$ sudo usermod -aG user2 johnny-three-groups
$ sudo usermod -aG user3 johnny-three-groups
```

```
$ sudo -u johnny-three-groups groups
johnny-three-groups users user user2 user3  ①

$ cp /bin/ls 8  ②
$ printf '#!/bin/bash\n\necho "Hello"\n' > 8.sh  ②
$ chmod 0000 8 8.sh  ③
$ setfacl -m g:user:r 8 8.sh
$ setfacl -m g:user2:w 8 8.sh
$ setfacl -m g:user3:x 8 8.sh
$ ls -la 8*
----rwx---+ 1 x x 40 Dec  2 00:47 8
----rwx---+ 1 x x 40 Dec  2 00:47 8.sh
$ getfacl 8*  ④
# file: 8
# owner: x
# group: x
user::---
group::---
group:user:r--
group:user2:-w-
group:user3:--x
mask::rwx
other::---

# file: 8.sh
# owner: x
# group: x
user::---
group::---
group:user:r--
group:user2:-w-
group:user3:--x
mask::rwx
other::---

$ sudo -u johnny-three-groups id
uid=1005(johnny-three-groups) gid=1005(johnny-three-groups) groups=1005(johnny-three-
groups),100(users),1000(user),1002(user2),1004(user3)

$ sudo -u johnny-three-groups cat ./8 | xxd -l 32  ⑤
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 306d 0000 0000 0000  ..>.....0m......
$ sudo -u johnny-three-groups ./8
8  8.sh

$ sudo -u johnny-three-groups cat 8.sh ⑥
#!/bin/bash

echo "Hello"

$ echo 'echo "World!"' | sudo -u johnny-three-groups tee -a 8.sh
```

```
echo "World!"

$ sudo -u johnny-three-groups ./8.sh
Hello
World!
```

① Создаём пользователя, состоящего в трёх группах `user`, `user2`, `user3`

② Создаём два исполняемых файла: ELF `8` и скрипт `8.sh`

③ Запрещаем любой доступ к файлам (`chmod 0000`) всем пользователям и разрешаем доступ на чтение, запись и исполнение группам `user`, `user2`, `user3`, соответственно, с помощью ACL-правил

④ Сформированные расширенные ACL-списки определяют заданный режим доступа

⑤ Файл `8` доступен группе `johnny-three-groups` и на чтение, и на исполнение

⑥ Аналогично, скрипт `8.sh` доступен на чтение (утилитой `cat`), запись (с помощью `tee -a`) и исполнение

Поиск подходящего правила в ACL-списке выполняется по именованным группам (`ACL_GROUP`), пока не будет найдено подходящее правило и подходящее право доступа!

Т.е. если первое подходящее правило - заданное для группы, которая содержится в списке групп запущенного процесса, не предоставило запрашиваемых прав, поиск продолжается по остальным правилам `ACL_GROUP` - пока не будет найдено правило для группы из списка групп процесса, предоставляющее права доступа.

Если ни одно подходящее правило не даёт запрашиваемые права, доступ запрещается.

Обратите внимание на пример запуска скрипта `./8.sh`. Заметили ли вы что-нибудь странное?

Однако права не собираются из разных правил для одной операции - ищется правило, задающее своими тремя битами требуемый режим доступа; в случае, если такое правило не найдено, в доступе отказывается:

```
$ head -n 23 02_file_reader.c | tail -n 3
    // Open file for reading and writing
    fd = open(argv[1], O_RDWR);  ①
    if (fd == -1) {

$ gcc -o file_reader 02_file_reader.c
$ sudo -u johnny-three-groups ./file_reader 8.sh  ②
Error opening file '8.sh': Permission denied

$ setfacl -m g:user:rw- 8.sh  ③
```

```
$ getfacl 8.sh
# file: 8.sh
# owner: x
# group: x
user::---
group::---
group:user:rw-  ③
group:user2:-w-
group:user3:--x
mask::rwx
other::---

$ sudo -u johnny-three-groups ./file_reader 8.sh  ④
File '8.sh' opened successfully. Contents:

#!/bin/bash

echo "Hello"
echo "World!"


File reading completed.
$ setfacl -m g:user:r 8.sh ⑤
```

① В коде программы `file_reader` запрашиваем права на чтение и запись (константа `O_RDWR`)

② Открытие файла `8.sh` без изменения ACL-списка недоступно: ни одно правило не предоставляет права `r` и `w` одновременно

③ Изменяем правило группы `user` на `rw`

④ Изменение правила для одной из групп на `rw` позволяет открытие файла утилитой `file_reader`

⑤ Восстанавливаем исходные права для дальнейших экспериментов

Однако запуск исполняемого скрипта `8.sh` (напомним, для этого требуются права на чтение и исполнение, причём в одном триплете) оказывается успешным:

```
$ ls -l 8.sh
----rwx---+ 1 x x 26 Dec  2 00:48 8.sh  ①
$ getfacl 8.sh  ②
# file: 8.sh
# owner: x
# group: x
user::---
group::---
group:user:r--
group:user2:-w-
group:user3:--x
mask::rwx
other::---
```

```
$ sudo -u johnny-three-groups ./8.sh  ③
Hello
World!
```

① По классической модели Unix группа владельцев имеет права `rwx` (второй триплет), несмотря на правило `group::---`

② ACL-список предоставляет по одному праву для трёх групп

③ Доступ на **чтение и исполнение** разрешён - правило для группы владельцев `group::---` игнорируется?!

ШОК!

Выглядит так, будто биты из правил для разных групп собрались-таки воедино для группы владельцев, хотя правило `ACL_GROUP_OBJ` явно **запрещает** доступ!

Подобное поведение мы также наблюдали в начале примера 8.

На самом деле противоречия (в данном случае) нет. Дело в том, что права `r` и `x` при непосредственном запуске скрипта проверяются **не одновременно**: сначала требуется право на исполнение, а затем на чтение.

Разберём подробнее на примере запуска bash-скрипта:

1. Интерпретатор `bash` может быть запущен в нескольких режимах:
   - интерактивая оболочка - режим выполнения команд в терминале;
   - `bash -c` - выполнение группы команд, заданных в строке-аргументе;
   - `bash script.sh` - выполнение скрипта с помощью явного запуска интерпретатора;
   - `./script.sh` - выполнение скрипта с указанным шебангом.
2. Запуская скрипт на исполнение напрямую:

   ```
   $ ./script.sh
   ```

   пользователь использует интерактивный режим `bash`. В процессе обработки интерактивно заданной команды управление передаётся функции `shell_execve()`:

   bash/execute_cmd.c:6126

   ```
   int
   shell_execve (char *command, char **args, char **env)
   {
     int i, fd, sample_len;
     char sample[HASH_BANG_BUFSIZ];
     size_t larray;

     SETOSTYPE (0);        /* Some systems use for USG/POSIX semantics */
   ```

```
  execve (command, args, env);  ①
  i = errno;            /* error from execve() */
  CHECK_TERMSIG;
  SETOSTYPE (1);
```

① Запуск системного вызова `sys_execve` с помощью системной функции `execve()`

из которой происходит вызов системной функции execve(2).

Вызов `execve("./script.sh", …)` требует наличия у запускающего процесса права на исполнение `x` на файл `script.sh`.

3. В процессе исполнения ядром системного вызова `sys_execve` выполняется одно из двух:
   - в случае наличия шебанга на первой строке файла `script.sh` путь к интерпретатору извлекается и выполняется:

     ```
     execve("/path/to/interpreter", ["/path/to/interpreter", "script.sh"], envp)
     ```

   - если в скрипте нет шебанга, ядро использует интерпретатор по умолчанию `/bin/sh`:

     ```
     execve("/bin/sh", ["/bin/sh", "script.sh"], envp)
     ```

     В большинстве дистрибутивов Linux `/bin/sh` является символической ссылкой на `/bin/bash`.

4. Ядро выполняет `sys_execve`, что приводит к запуску `bash` в режиме исполнения скрипта `script.sh`. В процессе исполнения команды `bash` управление передаётся функции `open_shell_script()`, которая среди прочего выполняет чтение файла `script.sh`:

   bash/shell.c:1572

   ```
   static int
   open_shell_script (char *script_name)
   {
     ...
     fd = open (filename, O_RDONLY);  ①
     if ((fd < 0) && (errno == ENOENT) && (absolute_program (filename) == 0))
     ...
   ```

   ① Открытие скрипта для чтения в режиме `O_RDONLY`

Очевидно, эта операция требует наличия у запускающего процесса права на чтение `r` файла `script.sh`.

Таким образом, запуск скрипта на исполнение подразумевает наличие у запускающего процесса прав на чтение и исполнение, однако эти права проверяются в различные моменты времени.

9. Последовательность проверки прав разных групп субъектов

Баааааааг!

4.4. Соответствие классической модели и списков контроля доступа: маска ACL

4.4.1. Механизм маски прав ACL_MASK

В случае наличия в ACL-списке хотя бы одного правила `ACL_USER` или `ACL_GROUP` список также обязан включать дополнительную запись `ACL_MASK` - правило-маску. Для минимального списка маска является необязательной (хотя, технически, может быть задана).

Маска `ACL_MASK` действует подобно верхней границе прав доступа, предоставляемых правилами `ACL_USER`, `ACL_GROUP` и `ACL_GROUP_OBJ`.

Правила `ACL_USER`, `ACL_GROUP` и `ACL_GROUP_OBJ` выделяют в отдельный класс, называемый *класс группы* [4, p. 17.4].

Объединение именованных пользователей и групп с группой владельцев файла можно рассматривать как своего рода *расширенную группу* владельцев: эти правила указываются для конкретных, идентифицируемых пользователей, остальные пользователи (кроме владельца файла) явно не указываются и попадают под правило `ACL_OTHER`.

Назначение правила `ACL_MASK` - обеспечить согласованное поведение при работе с файлом программ, не поддерживающих или не использующих списки контроля доступа. Покажем необходимость введения маски на примере [4, p. 17.1].

1. Предположим, файлу `file` установлен ACL-список:

```
user::r-x          # ACL_USER_OBJ
user:alice:r-x     # ACL_USER
group::r-x         # ACL_GROUP_OBJ
group:dev:r-x      # ACL_GROUP
other::--x         # ACL_OTHER
```

2. Пусть далее некоторая старая программа изменяет права доступа с помощью вызова `chmod("file", 0700)`. Не работающими с ACL-списками приложениями цель операции понимается однозначно: "Полностью запретить доступ к `file` всем, кроме владельца". Для согласованности управления доступом такая семантика должна обеспечиваться и ACL-списком.
3. Как необходимо изменить заданный ACL-список, чтобы добиться требуемого поведения?
   - во-первых, необходимо изменить записи `ACL_USER_OBJ`, `ACL_GROUP_OBJ` и `ACL_OTHER`, новый список выглядит так:

```
user::rwx          # ACL_USER_OBJ
user:alice:r-x     # ACL_USER
group::---         # ACL_GROUP_OBJ
group:dev:r-x      # ACL_GROUP
other::---         # ACL_OTHER
```

Очевидно, изменений правил минимального списка недостаточно, т.к. у пользователя `alice` и группы `dev` остались права доступа.

- в таком случае можно попробовать явно изменить правила `ACL_USER` и `ACL_GROUP` - в соответствии с триплетом прав для группы владельцев:

```
user::rwx          # ACL_USER_OBJ
user:alice:---     # ACL_USER
group::---         # ACL_GROUP_OBJ
group:dev:---      # ACL_GROUP
other::---         # ACL_OTHER
```

4. Такие изменения действительно решают поставленную задачу, однако приложение, не использующее ACL-списки, непреднамеренно вносит необратимые изменения в семантику изначального набора правил. Предположим, приложение далее пытается восстановить то, что оно считает изначальным набором прав, вызывая системную функцию `chmod("file", 0551)`. Список изменяется соответствующим образом:

```
user::r-x          # ACL_USER_OBJ
user:alice:---     # ACL_USER
group::r-x         # ACL_GROUP_OBJ
group:dev:---      # ACL_GROUP
other::--x         # ACL_OTHER
```

однако правила для пользователя `alice` и группы `dev` не восстанавливаются. Более того, этим правила оказываются утерянными безвозвратно!

Решением подобных проблем является введения специального правила `ACL_MASK`. Маска правил позволяет реализовать традиционное поведение `chmod()`, не нарушая семантику набора правил списка контроля доступа. Наличие записи `ACL_MASK` в списке влечёт следующие изменения:

- любые манипуляции с правами группы владельцев (второй триплет) в классической модели Unix посредством системной функции `chmod()` изменяют разрешения правила `ACL_MASK`, не `ACL_GROUP_OBJ`;
- системная функция `stat()` (а также утилиты наподобие `ls`) в качестве второго триплета прав возвращают разрешения правила `ACL_MASK`, а не `ACL_GROUP_OBJ`;
- *эффективные* права правил `ACL_USER`, `ACL_GROUP_OBJ` и `ACL_GROUP` **маскируются** - объединяются с помощью логического И с заданной маской.

С помощью правила `ACL_MASK` манипуляции в примере выше выглядят следующим образом:

1. После первого вызова `chmod("file", 0700)` в ACL-списке изменяется:

```
user::rwx                       # ACL_USER_OBJ
user:alice:r-x  #effective:---  # ACL_USER
group::r-x      #effective:---  # ACL_GROUP_OBJ
group:dev:r-x   #effective:---  # ACL_GROUP
mask::---                       # ACL_MASK
other::---                      # ACL_OTHER
```

   Правила `ACL_USER_OBJ` и `ACL_OTHER` изменились, остальные правила остались неизменными, а также добавилась маска `mask::---`. Появление маски изменяет эффективные права правил класса группы - теперь на самом деле только владелец `file` имеет права доступа. При этом разрешения, указанные в правилах класса группы, не изменились, поэтому возможно восстановление исходного состояния.

2. Второй вызов `chmod("file", 0551)` восстанавливает исходное состояние ACL-списка:

```
user::r-x         # ACL_USER_OBJ
user:alice:r-x    # ACL_USER
group::r-x        # ACL_GROUP_OBJ
group:dev:r-x     # ACL_GROUP
mask::r-x         # ACL_MASK
other::--x        # ACL_OTHER
```

Маска вычисляется автоматически при создании или изменении правил `ACL_USER`, `ACL_GROUP_OBJ` или `ACL_GROUP` так, чтобы предоставлялись все требуемые права каждого правила. Реализуется такое поведение просто - объединением с помощью логического ИЛИ всех триплетов правил класса группы.

Хотя правило `ACL_MASK` позволяет отразить изменения в списке контроля доступа для приложений, не использующих ACL-списки, обратное не верно.

Например [4, p. 17.4], пусть с файлом `file` ассоциирован ACL-список:

```
user::rw-         # ACL_USER_OBJ
group::---        # ACL_GROUP_OBJ
mask::---         # ACL_MASK
other::r--        # ACL_OTHER
```

после выполнения `chmod g+rw file` список изменяется:

```
user::rw-         # ACL_USER_OBJ
group::---        # ACL_GROUP_OBJ
mask::rw-         # ACL_MASK
```

```
other::r--        # ACL_OTHER
```

С точки зрения программ, не использующих ACL-списки, у группы владельцев должны быть права на файл:

```
$ ls -l file
-rw-rw-r--+ 1 user user 0 Dec 31 23:59 file
```

Однако на самом деле правило ACL_GROUP_OBJ запрещает доступ.

Таким образом, даже механизм ACL_MASK не позволяет реализовать полностью согласованное поведение двух механизмов разграничения доступа.

4.4.2. Примеры использования маски прав

1. chmod изменяет ACL_GROUP_OBJ минимального ACL-списка

```
$ touch file
$ ls -l file
-rw-rw-r-- 1 x x 0 Dec 31 23:59 file
$ getfacl --omit-header file
user::rw-
group::rw-
other::r--

$ chmod 0770 file
$ ls -l file
-rwxrwx--- 1 x x 0 Dec 31 23:59 file
$ getfacl --omit-header file
user::rwx
group::rwx  ①
other::---
```

① chmod изменил разрешения правила ACL_GROUP_OBJ

2. chmod изменяет ACL_MASK расширенного ACL-списка

Продолжая предыдущий пример:

```
$ setfacl -m user:user:rw file
$ ls -l file
-rwxrwx---+ 1 x x 0 Dec 31 23:59 file  ②
$ getfacl --omit-header file
user::rwx
user:user:rw-
group::rwx
mask::rwx
other::---
```

```
$ chmod 0333 file  ①
$ ls -l file
--wx-wx-wx+ 1 x x 0 Dec 31 23:59 file  ②
$ getfacl --omit-header file
user::-wx
user:user:rw-                   #effective:-w-
group::rwx                      #effective:-wx  ③
mask::-wx  ②
other::-wx
```

① Изменяем права доступа файла с ACL-списком с помощью `chmod`

② Второй триплет прав изменился, причём в выводе `ls` показываются права маски

③ Заданные в правиле `ACL_GROUP_OBJ` разрешения для группы владельцев не изменились - требуемые эффективные права задаются с помощью маскирования

3. Изменение `ACL_MASK` отражается на втором триплете прав

```
$ touch file2
$ setfacl -m g:nogroup:-w- file2
$ ls -l file2
-rw-rw-r--+ 1 x x 0 Dec 31 23:59 file2
$ getfacl --omit-header file2
user::rw-
group::rw-
group:nogroup:-w-
mask::rw-
other::r--

$ setfacl -m mask::--x file2  ①
$ ls -l file2
-rw---xr--+ 1 x x 0 Dec 31 23:59 file2  ②
```

① Изменяем маску на `--x`

② Вызов `stat()` возвращает в качестве прав группы владельцев разрешения правила `ACL_MASK`

4.5. Наследование списков контроля доступа

Помимо создания списка контроля доступа для конкретного файла существует возможность организовать наследование списка всеми файлами и поддиректориями, создаваемыми в некоторой директории. Наследование ACL-списков реализовано посредством т. н. *списков по умолчанию* (*default ACL*).

Список по умолчанию устанавливается на директорию и не используется для проверки прав доступа. Создание файлов и каталогов в директории с ACL-списком по умолчанию выполняется по следующим правилам:

1. директория наследует список по умолчанию в качестве своего ACL-списка по умолчанию - это позволяет реализовать иерархическое наследование списка;
2. файлу или директории устанавливается ACL-список (обычный), правила которого формируются на основе списка по умолчанию директории.

Предположим, в директории создаётся файл или директория и запрашиваются права доступа `mode`, представленные триплетом прав `abcdefghi`. ACL-список создаваемого объекта формируется по следующим правилам:

1. правила списка по умолчанию копируются в создаваемый список;
2. маска `umask` не участвует в процессе формирования триплетов правил;
3. триплет правила `ACL_USER_OBJ` объединяется с помощью логического И с первым триплетом режима создания файла `abc`;
4. в зависимости от наличия маски `ACL_MASK`:
   - правило `ACL_MASK` отсутствует - триплет правила `ACL_GROUP_OBJ` объединяется с помощью логического И с триплетом режима `def`;
   - правило `ACL_MASK` задано - триплет правила `ACL_MASK` объединяется с помощью логического И с триплетом режима `def`;
5. триплет правила `ACL_OTHER` объединяется с помощью логического И с третьим триплетом режима `ghi`.

На описанную процедуру можно смотреть как на маскирование наследуемого списка по умолчанию битами запрашиваемого набора прав.

Однако обратная интерпретация раскрывает назначение механизма наследования ACL-списков: список по умолчанию можно считать расширением механизма `umask`.
Правила списка по умолчанию выступают как маска для прав доступа создаваемых *обычным образом* файлов. Это позволяет настроить *максимальные* права доступа для всех файлы и поддиректорий.

Для создания и просмотра списка по умолчанию используются утилиты `setfacl` и `getfacl` с аргументом `-d`, удаление списка выполняется путём задания аргумента `-k` команды `setfacl`.

ACL-список по умолчанию хранится в виде расширенного атрибута с меткой `system.posix_acl_default`.

1. Создание ACL-списком по умолчанию влечёт создание обычного ACL-списка

```
$ mkdir -p /tmp/d
$ setfacl -d -m u::rwx,g::wx,o::x /tmp/d  ①
$ getfacl --omit-header /tmp/d
getfacl: Removing leading '/' from absolute path names
user::rwx  ②
group::rwx
other::r-x
```

```
default:user::rwx  ②
default:group::-wx
default:other::--x
```

① Создаём список по умолчанию

② В результате создаётся и ACL-список по умолчанию, и "обычный"

2. Создание файла и поддиректории в директории с минимальным ACL-списком по умолчанию

```
$ mkdir -p /tmp/no-acl
$ mkdir -p /tmp/acl
$ umask  ①
00002
$ touch /tmp/no-acl/file  ②
$ ls -l /tmp/no-acl/file
-rw-rw-r-- 1 x x 0 Dec  8 16:56 /tmp/no-acl/file  ③

$ setfacl -d -m u::rwx,g::wx,o::x /tmp/acl  ④
$ getfacl -d --omit-header /tmp/acl
getfacl: Removing leading '/' from absolute path names
user::rwx   ⑥
group::-wx  ⑦
other::--x  ⑧

$ touch /tmp/acl/file  ⑤
$ ls -l /tmp/acl/file
-rw--w---- 1 x x 0 Dec  8 16:40 file
$ getfacl --omit-header /tmp/acl/file
getfacl: Removing leading '/' from absolute path names
user::rw-   ⑥
group::-w-  ⑦
other::---  ⑧
```

① Установлена маска `0002`

② Создаём файл в директории без ACL-списка по умолчанию

③ Созданному файлу назначены права `0664` = `0666 & ~(0002)`

④ Создаём минимальный ACL-список по умолчанию

⑤ Создаём файл в директории с ACL-списком по умолчанию

⑥ Для правила `ACL_USER_OBJ` уставлены права `rw-` = `rwx & rw-`, где первый `rwx` задан в ACL по умолчанию, второй `rw-` указан при создании файла (первый триплет из `0666`)

⑦ Запись `ACL_MASK` отсутствует, поэтому для правила `ACL_GROUP_OBJ` уставлены права `-w-` = `-wx & rw-`, где первый `-wx` задан в ACL по умолчанию, второй `rw-` указан при создании файла (второй триплет из `0666`)

⑧ Для правила `ACL_OTHER` уставлены права `---` = `--x & rw-`, где первый `--x` задан в ACL по умолчанию, второй `rw-` указан при создании файла (третий триплет из `0666`)

Обратите внимание: при создании файла в директории без ACL-списка по умолчанию права отличаются от прав, назначенных файлу при создании в директории со списком по умолчанию - `umask` не используется во втором случае.

```
$ mkdir /tmp/no-acl/dir  ①
$ ls -l /tmp/no-acl | grep dir
drwxrwxr-x 2 x x 4096 Dec  8 17:19 dir  ②

$ getfacl -d --omit-header /tmp/acl  ③
getfacl: Removing leading '/' from absolute path names
user::rwx   ⑤
group::-wx  ⑥
other::--x  ⑦

$ mkdir /tmp/acl/dir  ④
$ ls -l /tmp/acl | grep dir
drwx-wx--x+ 2 x x 4096 Dec  8 17:20 dir
$ getfacl --omit-header /tmp/acl/dir
getfacl: Removing leading '/' from absolute path names
user::rwx   ⑤
group::-wx  ⑥
other::--x  ⑦
default:user::rwx   ⑧
default:group::-wx
default:other::--x
```

① Создаём поддиректорию в директории без ACL-списка по умолчанию

② Созданному файлу назначены права `0775` = `0777 & ~(0002)`

③ Минимальный ACL-список по умолчанию директории `acl`

④ Создаём поддиректорию в директории с ACL-списком по умолчанию

⑤ Первый триплет указанного при создании режима `0777` максимален, поэтому правило `ACL_USER_OBJ` копируется из списка по умолчанию без изменений

⑥ Второй триплет указанного при создании режима `0777` максимален, поэтому правило `ACL_GROUP_OBJ` (запись `ACL_MASK` отсутствует) копируется из списка по умолчанию без изменений

⑦ Третий триплет указанного при создании режима `0777` максимален, поэтому правило `ACL_OTHER` копируется из списка по умолчанию без изменений

⑧ Помимо обычного ACL-списка копируется список по умолчанию родительской директории

3. Создание файла и поддиректории в директории с расширенным ACL-списком по умолчанию

Продолжая предыдущий пример, добавим правило в ACL-список по умолчанию:

```
$ mkdir /tmp/acl/sub
```

```
$ setfacl -d -m g:nogroup:x /tmp/acl/sub  ①
$ getfacl --omit-header /tmp/acl/sub
getfacl: Removing leading '/' from absolute path names
user::rwx
group::-wx
other::--x
default:user::rwx
default:group::-wx
default:group:nogroup:--x
default:mask::-wx  ②
default:other::--x

$ touch /tmp/acl/sub/file  ③
$ ls -l /tmp/acl/sub/file
-rw--w----+ 1 x x 0 Dec  8 19:16 /tmp/acl/sub/file
$ getfacl --omit-header /tmp/acl/sub/file
getfacl: Removing leading '/' from absolute path names
user::rw-   ④
group::-wx                      #effective:-w-  ⑤
group:nogroup:--x               #effective:---  ⑤
mask::-w-   ⑤
other::---  ④

$ mkdir /tmp/acl/sub/dir
$ ls -l /tmp/acl/sub | grep dir
drwx-wx--x+ 2 x x 4096 Dec  8 19:03 dir
$ getfacl --omit-header /tmp/acl/sub/dir  ⑥
getfacl: Removing leading '/' from absolute path names
user::rwx
group::-wx
group:nogroup:--x
mask::-wx
other::--x
default:user::rwx
default:group::-wx
default:group:nogroup:--x
default:mask::-wx
default:other::--x
```

① Добавляем правило для группы `nogroup` в список по умолчанию директории `/tmp/acl/sub`

② ACL-список по умолчанию - список стал расширенным, появилась запись `ACL_MASK`

③ Создаём файл в директории `/tmp/acl/suv/file`

④ Правила `ACL_USER` и `ACL_OTHER`, как и ранее, оказались маскированными: права из соответствующих правил списка по умолчанию объединились (логическим И) с триплетом `rw-` из режима создания файла `0666`

⑤ ACL-список по умолчанию расширенный, поэтому маскирование при создании списка для файла выполнено для правила `ACL_MASK` (`-wx` из списка по умолчанию И `rw-` из режима создания файла), именованные правила скопированы как есть (биты `x` остались)

⑥ Т.к. `mkdir` задаёт права `0777` для создаваемой директории, ACL-список директории `dir` создаётся путём копирования правил из списка по умолчанию; сам список по умолчанию также копируется

4. Создание файла и поддиректории с другими правами доступа

Продолжая пример, создадим файл и директорию с другими правами:

```
$ echo "import os
import stat
os.close(os.open('/tmp/acl/sub/file-111', os.O_CREAT | os.O_WRONLY, 0o111))
os.makedirs('/tmp/acl/sub/dir-555', mode=0o555, exist_ok=True)" > 1.py
$ python3 1.py  ①

$ getfacl -d --omit-header /tmp/acl/sub
getfacl: Removing leading '/' from absolute path names
user::rwx
group::-wx
group:nogroup:--x
mask::-wx
other::--x

$ getfacl --omit-header /tmp/acl/sub/file-111
getfacl: Removing leading '/' from absolute path names
user::--x   ②
group::-wx                      #effective:--x
group:nogroup:--x
mask::--x   ②
other::--x  ②

$ getfacl --omit-header /tmp/acl/sub/dir-555
getfacl: Removing leading '/' from absolute path names
user::r-x   ③
group::-wx                      #effective:--x
group:nogroup:--x
mask::--x   ③
other::--x  ③
default:user::rwx  ④
default:group::-wx
default:group:nogroup:--x
default:mask::-wx
default:other::--x
```

① Создаём файл и поддиректорию в директории `/tmp/acl/sub`

② При создании файла были заданы права `0111`, поэтому триплеты правил `ACL_USER_OBJ` и `ACL_MASK` оказались изменены (`ACL_OTHER` также маскировалось, но это не изменило триплет правила)

③ Директория создавалась с правами `0555`, что отразилось на триплетах правил `ACL_USER_OBJ`, `ACL_MASK` и `ACL_OTHER`

④ ACL-список по умолчанию наследуется без изменений

5. После наследования списки живут отдельной жизнью

Продолжаем эксперименты, попробуем изменить ACL-список (обычный) одного из созданных файлов:

```
$ getfacl --omit-header /tmp/acl  ①
getfacl: Removing leading '/' from absolute path names
user::rwx
group::rwx
other::r-x
default:user::rwx
default:group::-wx
default:other::--x

$ getfacl --omit-header /tmp/acl/file  ②
getfacl: Removing leading '/' from absolute path names
user::rw-
group::-w-
other::---

$ setfacl -m u:user:rwx /tmp/acl/file  ③
$ getfacl --omit-header /tmp/acl/file  ④
getfacl: Removing leading '/' from absolute path names
user::rw-
user:user:rwx
group::-w-
mask::rwx
other::---

$ getfacl --omit-header /tmp/acl  ⑤
getfacl: Removing leading '/' from absolute path names
user::rwx
group::rwx
other::r-x
default:user::rwx
default:group::-wx
default:other::--x
```

① Исходный ACL-список по умолчанию родительской директории `/tmp/acl`

② Исходный ACL-список файла `file`

③ Добавляем правило в ACL-список файла `file`

④ ACL-список файла `file` изменился

⑤ ACL-список родительской директории не изменился

Более того, изменение списков по умолчанию также не отражается на родительском/дочернем списке по умолчанию:

```
$ getfacl --omit-header /tmp/acl  ①
getfacl: Removing leading '/' from absolute path names
user::rwx
group::rwx
other::r-x
default:user::rwx
default:group::-wx
default:other::--x

$ getfacl --omit-header /tmp/acl/sub  ②
getfacl: Removing leading '/' from absolute path names
user::rwx
group::-wx
other::--x
default:user::rwx
default:group::-wx
default:group:nogroup:--x
default:mask::-wx
default:other::--x

$ getfacl --omit-header /tmp/acl/sub/dir  ③
getfacl: Removing leading '/' from absolute path names
user::rwx
group::-wx
group:nogroup:--x
mask::-wx
other::--x
default:user::rwx
default:group::-wx
default:group:nogroup:--x
default:mask::-wx
default:other::--x

$ setfacl -m u::-,g::-,o::- /tmp/acl/sub  ④
$ setfacl -d -m u::-,g::-,o::- /tmp/acl/sub

$ getfacl --omit-header /tmp/acl  ⑤
getfacl: Removing leading '/' from absolute path names
user::rwx
group::rwx
other::r-x
default:user::rwx
default:group::-wx
default:other::--x

$ sudo getfacl --omit-header /tmp/acl/sub/dir  ⑥
getfacl: Removing leading '/' from absolute path names
user::rwx
group::-wx
group:nogroup:--x
```

```
mask::-wx
other::--x
default:user::rwx
default:group::-wx
default:group:nogroup:--x
default:mask::-wx
default:other::--x
```

① Исходный ACL-список по умолчанию родительской директории `/tmp/acl`

② Исходный ACL-список по умолчанию директории `/tmp/acl/sub`

③ Исходный ACL-список по умолчанию дочерней директории `/tmp/acl/sub/dir`

④ Изменяем оба ACL-списка директории `/tmp/acl/sub`

⑤ ACL-список родительской директории не изменился

⑥ ACL-список дочерней директории не изменился

Вывод: после построения иерархии изменения ACL-списков не распространяются автоматически. Однако `setfacl` можно выполнять рекурсивно - с флагом `-R`.

6. Списки по умолчанию хранятся с помощью расширенного атрибута `system.posix_acl_default`

Для хранения списков по умолчанию используется расширенный атрибут с меткой `system.posix_acl_default`:

```
$ sudo getfattr --dump --match=- -- /tmp/acl/sub/dir
getfattr: Removing leading '/' from absolute path names
# file: tmp/acl/sub/dir
system.posix_acl_access=0sAgAAAAEABwD/////BAADAP////8IAAEA/v8AABAAAwD/////IAABAP////8=
①
system.posix_acl_default=0sAgAAAAEABwD/////BAADAP////8IAAEA/v8AABAAAwD/////IAABAP////8
=  ②
```

① `xattr system.posix_acl_default` хранит ACL-список по умолчанию

② `xattr system.posix_acl_access` хранит обычный ACL-список

Источники

- [1] [Wikipedia] Discretionary access control
- [2] Nemeth, Evi, et al. UNIX and Linux system administration handbook. Addison-Wesley Professional, 2018.
- [3] Shotts, William. The Linux command line: a complete introduction. No Starch Press, 2019.
- [4] Майкл, Керриск. Linux API. Исчерпывающее руководство."Питер", 2017.
- [5] [Red Hat Blog] Linux file permissions explained
- [6] [Red Hat Blog] Linux permissions: SUID,SGID,and sticky bit
- [7] [Unix StackExchange] Allow setuid on shell scripts

- [8] Tevault, Donald A. Mastering Linux Security and Hardening: A practical guide to protecting your Linux system from cyber attacks. Packt Publishing Ltd, 2023.
- [9] [Wikipedia] File attribute
- [10] [Linux manual page] acl - Access Control Lists