Национальный Открытый Университет "ИНТУИТ":
www.intuit.ru
Георгий Курячий, Кирилл Маслинский
Лекция 6. Права доступа

Права доступа в файловой системе

Работая с Linux, Мефодий заметил, что некоторые файлы и каталоги недоступны ему ни для чтения, ни для записи, ни для использования. Зачем такие нужны? Оказывается, другие пользователи могут обращаться к этим файлам, а у Мефодия просто **недостаточно прав**.

Идентификатор пользователя

Говоря о *правах доступа* пользователя к файлам, стоит заметить, что в действительности манипулирует файлами не сам пользователь, а запущенный им *процесс* (например, утилита `rm` или `cat`). Поскольку и файл, и *процесс* создаются и управляются системой, ей нетрудно организовать какую угодно политику доступа одних к другим, основываясь на любых свойствах *процессов* как `субъектов` и файлов как `объектов` системы.

В Linux, однако, используются не какие угодно свойства, а результат *идентификации* пользователя – его *UID*. Каждый *процесс* системы обязательно **принадлежит** какому-нибудь пользователю, и *идентификатор пользователя* (*UID*) – обязательное свойство любого *процесса* Linux. Когда программа `login` запускает *стартовый командный интерпретатор*, она приписывает ему *UID*, полученный в результате диалога. Обычный запуск программы (`exec()`) или порождение нового *процесса* (`fork()`) не изменяют *UID процесса*, поэтому **все** *процессы*, запущенные пользователем во время терминальной сессии, будут иметь его *идентификатор*.

Поскольку *UID* однозначно определяется *входным именем*, оно нередко используется **вместо** *идентификатора* – для наглядности. Например, вместо выражения " *идентификатор пользователя*, соответствующий *входному имени* `methody` ", говорят " *UID* `methody` " (в приведенном ниже примере этот *идентификатор* равен **503**):

```
[methody@localhost methody]$ id
 uid=503(methody) gid=503(methody) группы=503(methody)
[methody@localhost methody]$ id shogun
 uid=400(shogun) gid=400(shogun) г
руппы=400(shogun),4(adm),10(wheel),19(proc)
```

`6.1. Как узнать идентификаторы пользователя и членство в группах`

Утилита `id` , которой воспользовался Мефодий, выводит *входное имя* пользователя и соответствующий ему *UID*, а также *группу по умолчанию* и полный список *групп*, членом которых он является.

Идентификатор группы

Как было рассказано в лекции 1, пользователь может быть членом нескольких *групп*, равно как и несколько пользователей могут быть членами одной и той же *группы*. Исторически сложилось так, что одна из *групп* – *группа по умолчанию* – является для пользователя основной: когда (не вполне точно) говорят о " *GID* пользователя", имеют в виду именно идентификатор *группы по умолчанию*. В лекции 12 будет рассказано о том, что *GID* пользователя вписан в *учетную запись* и хранится в `/etc/passwd` , а информация о соответствии **имен** *групп* их идентификаторам, равно как и о том, в какие **еще** *группы* входит пользователь – в файле `/etc/group` . Из этого следует, что пользователь **не может не быть** членом как минимум одной *группы*, как снаряд не может не попасть в эпицентр взрыва. [1]

Часто процедуру **создания** пользователя проектируют так, что имя *группы по умолчанию* совпадает с *входным именем* пользователя, а *GID* пользователя – с его *UID*. Однако это совсем не обязательно: не всегда нужно заводить для пользователя отдельную *группу*, а если заводить, то не всегда **удается** сделать так,чтобы **желаемый** идентификатор *группы* совпадал с **желаемым** *идентификатором пользователя*.

Ярлыки объектов файловой системы

При **создании** объектов файловой системы – файлов, каталогов и т. п. – каждому в обязательном порядке приписывается *ярлык*. ***Ярлык*** включает в себя *UID – идентификатор пользователя - хозяина файла*, *GID* – идентификатор *группы*, которой принадлежит файл, тип объекта и набор так называемых *атрибутов*, а также некоторую дополнительную информацию. *Атрибуты* определяют, кто и что имеет право делать с файлом, они описаны ниже:

```
[methody@arnor methody]$ ls -l
 итого 24
 drwx------     2       methody methody 4096    Сен 12  13:58   Documents
 drwxr-xr-x     2       methody methody 4096    Окт 31  15:21   examples
 -rw-r--r--     1       methody methody 26      Сен 22  15:21   loop
 -rwxr-xr-x     1       methody methody 23      Сен 27  13:27   script
 drwx------     2       methody methody 4096    Окт 1   15:07   tmp
 -rwxr-xr-x     1       methody methody 32      Сен 22  13:26   to.sort
```

6.2. Права доступа к файлам и каталогам, показанные командой ls -l

Ключ " -l " утилиты ls определяет "длинный" (**l** ong) формат выдачи (справа налево): имя файла, время последнего изменения файла, размер в байтах, *группа*, *хозяин*, количество *жестких ссылок* и строчка *атрибутов*. Первый символ в строчке *атрибутов* определяет тип файла. Тип " - " отвечает "обычному" файлу, а тип " d " – каталогу (**d** irectory). Имя пользователя и имя *группы*, которым принадлежит содержимое домашнего каталога Мефодия, – естественно, methody .

Быстрый разумом Мефодий немедленно заинтересовался вот чем: несмотря на то, что создание *жестких ссылок* на каталог невозможно, значение поля "количество *жестких ссылок* " для всех каталогов примера равно **двум**, а не одному. На самом деле этого и следовало ожидать, потому что **любой** каталог файловой системы Linux всегда имеет не менее двух имен: собственное (например, tmp) и имя " ." в самом этом каталоге (tmp/ .). Если же в каталоге создать подкаталог, количество *жестких ссылок* на этот каталог увеличится на 1 за счет имени " . ." в подкаталоге (например, tmp/subdir1/. .):

```
[methody@arnor methody]$ ls -ld tmp
 drwx------ 3 methody methody 4096 Окт 1 15:07 tmp
[methody@arnor methody]$ mkdir tmp/subdir2
[methody@arnor methody]$ ls -ld tmp
 drwx------ 4 methody methody 4096 Окт 1 15:07 tmp
[methody@arnor methody]$ rmdir tmp/subdir*
[methody@arnor methody]$ ls -ld tmp
 drwx------ 2 methody methody 4096 Окт 1 15:07 tmp
```

6.3. Несколько жестких ссылок на каталог все-таки бывает!

Здесь Мефодий использовал ключ " -d " (**d** irectory) для того, чтобы ls выводил информацию не о содержимом каталога tmp , а о самом этом каталоге.

Иерархия прав доступа

Теперь – более подробно о том, чему соответствуют девять символов в строке *атрибутов*, выдаваемой `ls`. Эти девять символов имеют вид "`rwxrwxrwx`", где некоторые "`r`","`w`" и "`x`" могут заменяться на "`-`". Очевидно, буквы отражают принятые в Linux три вида доступа – чтение, запись и использование – однако в *ярлыке* они присутствуют в трех экземплярах!

Дело в том, что любой пользователь (*процесс*) Linux по отношению к любому файлу может выступать в трех **ролях**: как *хозяин* (**u** ser), как член *группы*, которой принадлежит файл (**g** roup), и как *посторонний* (**o** ther), никаких отношений собственности на этот файл не имеющий. Строка *атрибутов* – это три тройки "`rwx`", описывающие *права доступа* к файлу *хозяина* этого файла (первая тройка, "`u`"), *группы*, которой принадлежит файл (вторая тройка, "`g`") и *посторонних* (третья тройка, "`o`"). Если в какой-либо тройке не хватает буквы, а вместо нее стоит "`-`", значит, пользователю в соответствующей роли будет в соответствующем виде доступа отказано.

При выяснении отношений между файлом и пользователем, запустившим *процесс*, роль определяется так:

1. Если *UID* файла совпадает с *UID процесса*, пользователь – *хозяин файла*
2. Если *GID* файла совпадает с *GID* **любой** *группы*, в которую входит пользователь, он – член *группы*, которой принадлежит файл.
3. Если ни *UID*, ни *GID* файла не пересекаются с *UID процесса* и списком *групп*, в которые входит запустивший его пользователь, этот пользователь – *посторонний*.

Именно в роли *хозяина* пользователь (*процесс*) может **изменять** *ярлык файла*. Это вполне соответствует обыденным понятиям о собственности (" **мой** файл: захочу – покажу, захочу – спрячу"). Единственное, чего не может делать *хозяин* со своим файлом – менять ему *хозяина*.

Далее следует протокол действий Гуревича, который, пользуясь возможностями *суперпользователя*, создал в каталоге `/tmp` несколько файлов с различными *правами доступа* к ним. Для того чтобы напомнить человеку, что работа ведется с *правами суперпользователя* (это требует гораздо большей ответственности), `bash` заменил привычный "доллар" в конце *приглашения командной строки* на "решетку". *Входное имя* он тоже заменил, но практика показывает, что решетка эффективнее:

```
[root@localhost root]# echo "All can read" > /tmp/read.all
[root@localhost root]# echo "Group wheel can read" > /tmp/read.wheel
[root@localhost root]# echo "Group methody can read" > /tmp/read.methody
[root@localhost root]# echo "Methody himself can read" > /tmp/read.Methody
[root@localhost root]# chgrp wheel /tmp/read.wheel; chmod o-r /tmp/read.wheel
[root@localhost root]# chgrp methody /tmp/read.methody; chmod o-r /tmp/read.methody
[root@localhost root]# chown methody /tmp/read.Methody; chmod og-r /tmp/read.Methody
```

6.4. Создание файлов с различными правами доступа

Права доступа изменяются с помощью трех команд: `chown` (**ch** ange **own** er, сменить владельца), `chgrp` (**ch** ange **gr** ou`p` , сменить *группу*) и `chmod` с расширенным форматом параметра: перед частью, определяющей **доступ** (перед знаком "`+`" или "`-`"), могут быть перечислены роли "`u`","`g`","`o`" и "`a`" (**a** ll, что соответствует "`ugo`"), доступ для которых изменяется. Кроме того, при задании доступа можно вместо "`+`" и "`-`" использовать "`=`", тогда для заданных ролей указанные способы доступа разрешаются, а **неуказанные** – запрещаются. Вместо пары команд `chown` *`хозяин файл`* ; `chgrp группа файл` можно применять одну: `chown хозяин:группа файл` , которая изменяет одновременно и

UID, и *GID* файла (каталога, ссылки и т. п.).

Мефодий хочет посмотреть, кто имеет доступ к файлам, созданным Гуревичем, а вдобавок – к файлу `/etc/shadow`. Для этого он использует команду `ls -l` с *шаблоном*, описывающим сразу **все** файлы, которые находятся в `/tmp` и имя которых **начинается** на "`read`.".

```
[methody@localhost methody]$ ls -l /tmp/read.* /etc/shadow
 -r-------- 1 root  root   0 Сен 10 02:08 /etc/shadow
 -rw-r--r-- 1 root  root  13 Сен 22 17:49 /tmp/read.all
 -rw-r----- 1 root  methody 23 Сен 22 17:49 /tmp/read.methody
 -rw------- 1 methody root  25 Сен 22 17:50 /tmp/read.Methody
 -rw-r----- 1 root   wheel  21 Сен 22 17:49 /tmp/read.wheel
[methody@localhost methody]$ cat /tmp/read.* /etc/shadow
 All can read
 Group methody can read
 Methody himself can read
 cat: /tmp/read.wheel: Permission denied
 cat: /etc/shadow: Permission denied
```

6.5. Чтение файлов с различными правами доступа

Что же получается? Из файла `/etc/shadow` можно только читать, причем только пользователю `root`. Изменять файлы `/tmp/read.all`, `/tmp/read.wheel` и `/tmp/read.methody` может только `root`, он же может и читать из них. Также читать из файла `/tmp/read.wheel` могут члены *группы* `wheel`, а из файла `/tmp/read.methody` – члены *группы* `methody` (это имя *группы*, а не имя пользователя!). Читать и писать в файл `/tmp/read.Methody` может только пользователь `methody`. Наконец, читать из файла `/tmp/read.all` могут к тому же и члены *группы* `root`, и вообще любые пользователи.

Попытка вывести содержимое этих файлов на экран (а значит, **прочитать** их) приводит к ожидаемому результату: *процессу* `cat` (*UID* `methody`) разрешено чтение из трех файлов. Из `/tmp/read.all` – так как по отношению к нему Мефодий играет роль *постороннего*, а ему открыт доступ на чтение ("`r`" в начале третьей тройки). Из `/tmp/read.methody` – так как пользователь `methody` входит в *группу* `methody` (см. пример 6.1), членам которой разрешено читать из этого файла ("`r`" в начале второй тройки). И, конечно, из файла `/tmp/read.Methody`, которому `methody` – *хозяин*, имеющий доступ на чтение ("`r`" в первой тройке).

Если бы Мефодию захотелось **записать** что-нибудь в эти файлы, он бы получил доступ только к одному – `/tmp/read.Methody`, потому что по отношению к остальным файлам Мефодий играет роль, которой закрыт доступ на запись (`/tmp/read.methody` – член *группы*, остальные три – *посторонние*).

Таким образом, определение *прав доступа процесса* к объекту файловой системы (например, файла) происходит так. Используя *UID процесса*, список *групп*, в которые входит пользователь, запустивший этот *процесс*, *UID* файла и *GID* файла, система определяет **роль** *процесса* по отношению к файлу, а затем обращается к соответствующей тройке *атрибутов* файла. Стоит заметить, что *процесс* не может выступать сразу в **нескольких ролях**, поэтому, например, файл с *ярлыком* "`---rw-rw- methody methody`" **сам** Мефодий просмотреть не сможет (тройка *хозяина* определяет полное отсутствие доступа) [2].

Использование прав доступа в Linux

Использование групп

В Linux определено несколько *системных групп*, задача которых – обеспечивать доступ членов этих *групп* к разнообразным ресурсам системы. Часто такие *группы* носят говорящие названия: " disk ", " audio ", " cdwriter " и т. п. Тогда обычным пользователям доступ к некоторому файлу, каталогу или *файлу-дырке* Linux закрыт, но открыт членам *группы*, которой этот объект принадлежит.

Например, в Linux почти всегда используется *виртуальная файловая система* /proc – каталог, в котором в виде подкаталогов и файлов представлена информация из *таблицы процессов*. Имя подкаталога /proc совпадает с *PID* соответствующего *процесса*, а содержимое этого подкаталога отражает свойства *процесса*. *Хозяином* такого подкаталога будет хозяин *процесса* (с правами на чтение и использование), поэтому **любой** пользователь сможет посмотреть информацию о **своих** *процессах*. Именно каталогом /proc пользуется утилита ps :

```
[methody@arnor methody]$ ls -l /proc
 . . .
 dr-xr-x---  3 methody proc  0 Сен 22 18:17 4529
 dr-xr-x---  3 shogun proc   0 Сен 22 18:17 4558
 dr-xr-x---  3 methody proc  0 Сен 22 18:17 4589
 . . .
[methody@localhost methody]$ ps -af
 UID       PID PPID C STIME TTY     TIME CMD
 methody  4529 4523 0 13:41 tty1 00:00:00 -bash
 methody  4590 4529 0 13:42 tty1 00:00:00 ps -af
```

6.6. Ограничение доступа к полной таблице процессов

Оказывается, запущено немало *процессов*, в том числе один – пользователем shogun (*PID* 4558). Однако, несмотря на ключ " -a " (**a** ll), ps выдала Мефодию только сведения о его *процессах*: 4529 – это входной shell, а 4590 – видимо, сам ps .

Другое дело – Гуревич. Он, как видно из примера 6.1, входит в *группу* proc , членам которой разрешено читать и использовать каждый подкаталог /proc :

```
shogun@localhost ~ $ ps -af
 UID       PID  PPID C STIME TTY     TIME CMD
 methody  4529  4523 0 13:41 tty1 00:00:00 -bash
 shogun   4558  1828 0 13:41 tty3 00:00:00 -zsh
 shogun   4598  4558 0 13:41 tty3 00:00:00 ps -af
```

6.7. Доступ к полной таблице процессов: группа proc

Гуревич, опытный пользователь Linux, предпочитает bash "The Z Shell", zsh . Отсюда и различие приглашения в командной строке. Во всех shell, кроме самых старых, символ " ~ " означает домашний каталог. Этим сокращением удобно пользоваться, если текущий каталог – не домашний, а оттуда (или туда) нужно скопировать файл. Получается команда наподобие " cp ~/нужный_файл ." или " cp нужный_файл ~ " соответственно.

Команда ps -a выводит информацию обо **всех** *процессах*, запущенных "живыми" (а не *системными) пользователями* [3]. Для просмотра всех *процессов* Гуревич пользуется командой ps -efH .

Разделяемые каталоги

Проанализировав систему *прав доступа* к каталогам в Linux, Мефодий пришел к выводу, что в

ней имеется существенный недочет. Тот, кто имеет право **изменять** каталог, может **удалить любой файл** оттуда, даже такой, к которому не имеет доступа. Формально все правильно: удаление файла из каталога – всего лишь изменение содержимого каталога. Если у файла было больше одного имени (**существовало несколько** *жестких ссылок* **на этот файл**), **никакого удаления данных не произойдет, а если ссылка была последней – файл в самом деле удалится**. Вот это-то, по мнению Мефодия, и плохо.

Чтобы доказать новичку, что право на удаление любых файлов **полезно**, кто-то создал прямо в домашнем каталоге Мефодия файл, недоступный ему, да к тому же с подозрительным именем, содержащим пробел:

```
[methody@localhost methody]$ ls
 4TO-TO Mep3koe Documents examples loop script tmp to.sort
[methody@localhost methody]$ ls -l 4*
 -rw------- 1 root root 0 Сен 22 22:20 4TO-TO Mep3koe
[methody@localhost methody]$ rm -i 4*
 rm: удалить защищенный от записи пустой обычный файл `4TO-TO Mep3koe'? y
```

6.8. Удаление чужого файла с неудобным именем

Подозревая, что от хулиганов всего можно ожидать, Мефодий не решился **набрать** имя удаляемого файла с клавиатуры, а воспользовался *шаблоном* и ключом " `-i` " (**i** nteractive) команды `rm` , чтобы та ожидала **подтверждения** перед тем, как удалять очередной файл. Несмотря на отсутствие доступа к самому файлу, удалить его оказалось возможно:

```
[methody@localhost methody]$ ls -dl /tmp
 drwxrwxrwt 4 root root 1024 Сен 22 22:30 /tmp
[methody@localhost methody]$ ls -l /tmp
 итого 4
 -rw-r--r-- 1 root    root    13 Сен 22 17:49 read.all
 -rw-r----- 1 root    methody 23 Сен 22 17:49 read.methody
 -rw------- 1 methody root    25 Сен 22 22:30 read.Methody
 -rw-r----- 1 root    wheel   21 Сен 22 17:49 read.wheel
[methody@localhost methody]$ rm -f /tmp/read.*
 rm: невозможно удалить '/tmp/read.all': Operation not permitted
 rm: невозможно удалить '/tmp/read.methody': Operation not permitted
 rm: невозможно удалить '/tmp/read.wheel': Operation not permitted
[methody@localhost methody]$ ls /tmp
 read.all read.methody read.wheel
```

6.9. Работа с файлами в разделяемом каталоге

Убедившись, что **любой** доступ в каталог `/tmp` открыт **всем**, Мефодий решил удалить оттуда все файлы. Затея гораздо более хулиганская, чем **заведение** файла: а вдруг они кому-нибудь нужны? Удивительно, но удален оказался только файл, принадлежащий самому Мефодию...

Дело в том, что Мефодий проглядел особенность *атрибутов* каталога `/tmp` : вместо " `x` " в тройке "для *посторонних* " `ls` выдал " `t` ". Это **еще** один *атрибут* каталога, наличие которого как раз и запрещает пользователю удалять оттуда файлы, которым он не *хозяин*. Таким образом, права записи в каталог с *ярлыком* " `drwxrwxrwt группа хозяин` " и для членов *группы*, и для *посторонних* ограничены их собственными файлами, и только *хозяин* имеет право изменять список файлов в каталоге, как ему вздумается. Такие каталоги

называются *разделяемыми*, потому что предназначены они, как правило, для **совместной** работы всех пользователей в системе, обмена информацией и т. п.

При установке *атрибута* " `t` " доступ на использование для *посторонних* (" `t` " в строчке *атрибутов* стоит на месте последнего " `x` ") не отменяется. Просто они так редко используются друг без друга, что `ls` выводит их в одном и том же месте. Если кому-нибудь придет в голову организовать *разделяемый каталог* без доступа *посторонним* на использование, `ls` выведет на месте девятого *атрибута* не " `t` ", а " `T` ":

```
[methody@localhost methody]$ ls -l loop
 -rw-r--r-- 1 root  root  26 Сен 22 22:10 loop
[methody@localhost methody]$ chown methody loop
 chown: изменение владельца `loop': Operation not permitted
[methody@localhost methody]$ cp loop loopt
[methody@localhost methody]$ ls -l loop*
 -rw-r--r-- 1 root    root    26 Сен 22 22:10 loop
 -rw-r--r-- 1 methody methody 26 Сен 22 22:15 loopt
[methody@localhost methody]$ mv -f loopt loop
[methody@localhost methody]$ ls -l loop*
 -rw-r--r-- 1 methody methody 26 Сен 22 22:15 loop
```

6.10. Что можно делать с чужим файлом в своем каталоге

Оказывается, мелкие пакости продолжаются. Кто-то сменил файлу `loop` *хозяина*, так что теперь Мефодий может только читать его, но не изменять. Удалить этот файл – проще простого, но хочется "вернуть все как было": чтобы получился файл с тем же именем и тем же содержанием, принадлежащий Мефодию, а не `root` . Это несложно: чужой файл можно переименовать (это действие над каталогом, а не над файлом), скопировать переименованный файл в файл с именем старого (доступ на чтение открыт) и, наконец, удалить чужой файл с глаз долой. Ключ " `-f` " (**f** orce, "силком") позволяет утилите `mv` делать свое дело, не спрашивая подтверждений. В частности, увидев, что файл с именем, в которое необходимо переименовывать, существует, даже чужой, даже недоступный на запись, `mv` преспокойно удалит его и выполнит операцию переименования.

Суперпользователь

Мефодий возмутился, узнав, что кто-то может проделывать **над ним** всякие штуки, которые сам Мефодий ни над кем проделывать не может. Обоснованное подозрение пало на Гуревича, единственного администратора этой системы, обладающего *правами суперпользователя*.

Суперпользователь - единственный пользователь в Linux, на которого не распространяются ограничения *прав доступа*. Имеет нулевой *идентификатор пользователя* .

Суперпользователь в Linux – это **выделенный** пользователь системы, на которого **не распространяются** ограничения *прав доступа*. *UID* суперпользовательских *процессов* равен `0` : так система отличает их от *процессов* других пользователей. Именно *суперпользователь* имеет возможность произвольно изменять владельца и *группу* файла. Ему открыт доступ на чтение и запись к **любому файлу** системы и доступ на чтение, запись и использование к **любому каталогу**. Наконец, суперпользовательский *процесс* может на время сменить **свой собственный** *UID* с нулевого на любой другой. Именно так и поступает программа `login` , когда, проведя процедуру *идентификации* пользователя, запускает *стартовый командный интерпретатор*.

Среди *учетных записей* Linux всегда есть запись по имени `root` ("корень"), соответствующая нулевому *идентификатору*, поэтому вместо " *суперпользователь* " часто говорят " `root` ". [4]

Множество системных файлов принадлежат root , множество файлов только ему доступны на чтение или запись. Пароль этой учетной записи – одна из самых больших драгоценностей системы. Именно с ее помощью системные администраторы выполняют самую ответственную работу. Свойство root иметь доступ ко **всем** ресурсам системы накладывает очень высокие требования на **человека**, знающего пароль root . *Суперпользователь* может все – в том числе и все поломать, поэтому **любую работу** стоит вести с *правами* обычного пользователя, а к *правам* root прибегать только по необходимости.

Существует два различных способа получить *права суперпользователя*. Первый – это **зарегистрироваться в системе** под этим именем, ввести пароль и получить *стартовую оболочку*, имеющую нулевой *UID*. Это – самый неправильный способ, пользоваться которым стоит, только если нельзя применить другие. Что в этом случае выдаст команда last ? Что тогда-то с такой-то консоли в систему вошел **неизвестно кто** с *правами суперпользователя* и что-то там такое **делал**. С точки зрения системного администратора, это очень подозрительное событие, особенно если сам он в это время к указанной консоли не подходил... Сами администраторы такого способа избегают.

Второй способ —воспользоваться специальной утилитой su (**s** hell of **u** ser), которая позволяет выполнить одну или несколько команд от лица другого пользователя. По умолчанию эта утилита выполняет команду sh от лица пользователя root , то есть запускает командный интерпретатор с нулевым *UID*. Отличие от предыдущего способа – в том, что всегда известно, **кто именно** запускал su , а значит, ясно, с кого спрашивать за последствия. В некоторых случаях удобнее использовать не su , а утилиту sudo , которая позволяет выполнять **только заранее заданные** команды.

Подмена идентификатора

Утилиты su и sudo имеют некоторую странность, объяснить которую Мефодий пока не в состоянии. Эта же странность распространяется и на давно известную программу passwd , которая позволяет редактировать собственную *учетную запись*. Запускаемый *процесс* **наследует** *UID* от родительского, поэтому если этот *UID* – не нулевой, он не в состоянии **поменять** его. Тогда как же su запускает для обычного пользователя **суперпользовательский** shell? Как passwd получает доступ к хранилищу всех *учетных записей*? Должен существовать механизм, позволяющий пользователю запускать *процессы* с *идентификаторами* **другого** пользователя, причем механизм строго контролируемый, иначе с его помощью можно натворить немало бед.

В Linux этот механизм называется *подменой идентификатора* и устроен очень просто. *Процесс* может сменить свой *UID*, если запустит вместо себя при помощи exec() **другую** программу из файла, имеющего специальный *атрибут SetUID* [5]. В этом случае *UID процесса* становится равным *UID* **файла**, из которого программа была запущена:

```
[foreigner@somewhere foreigner]$ ls -l /usr/bin/passwd /bin/su
 -rws--x--x 1 root root 19400 Фев  9 2004 /bin/su
 -rws--x--x 1 root root 5704  Янв 18 2004 /usr/bin/passwd
[foreigner@somewhere foreigner]$ ls -l /etc/shadow
 -r-------- 1 root root 5665  Сен 10 02:08 /etc/shadow
```

6.11. Обычная программа passwd, использующая SetUID

Как и в случае с t- *атрибутом*, ls выводит букву " s " вместо буквы " x " в тройке "для *хозяина* ". Точно так же, если соответствующего x- *атрибута* нет (что бывает редко), ls выведет " S " вместо " s ". Во многих дистрибутивах Linux и /bin/su , и /usr/bin/passwd имеют установленный *SetUID* и принадлежат пользователю root , что и позволяет su запускать *процессы* с правами этого пользователя (а значит, и любого другого), а passwd – модифицировать файл /etc/shadow , содержащий в таких системах сведения обо всех

учетных записях. Как правило, файлы с *атрибутом SetUID* доступны обычным пользователям только на выполнение, чтобы не провоцировать пользователей рассматривать содержимое этих файлов и исследовать их **недокументированные возможности**. Ведь если обнаружится способ заставить, допустим, программу `passwd` **выполнить** любую другую программу, то все проблемы с защитой системы от взлома будут разом решены – нет защиты, нет и проблемы.

Однако Мефодий работает с такой системой, где `/usr/bin/passwd` вообще не имеет *атрибута SetUID*. Зато эта программа принадлежит *группе* `shadow` и имеет другой *атрибут*, *SetGID*, так что при ее запуске *процесс* получает идентификатор *группы* `shadow`. Утилита `ls` выводит *SetGID* в виде "`s`" вместо "`x`" во второй тройке *атрибутов* ("для *группы*"). Замечания касательно "`s`", "`S`" и "`x`" действительны для *SetGID* так же, как и для *SetUID*:

```
[root@localhost root]# ls -l /usr/bin/passwd
 -rwx--s--x 1 root shadow 5704 Jan 18 2004 /usr/bin/passwd
[root@localhost root]# ls -al /etc/tcb/methody
 total 3
 drwx--s---  2 methody auth  1024 Sep 22 12:58 .
 drwx--x--- 55 root  shadow 1024 Sep 22 18:41 ..
 -rw-r-----  1 methody auth   81 Sep 22 12:58 shadow
 -rw-------  1 methody auth    0 Sep 12 13:58 shadow-
 -rw-------  1 methody auth    0 Sep 12 13:58 shadow.lock
```

6.12. Не подверженная взлому программа passwd, использующая SetGID

Каталог `/etc/tcb` в этой системе содержит подкаталоги, соответствующие *входным именам* всех ее пользователей. В каждом подкаталоге хранится, в числе прочего, `собственный` файл `shadow` соответствующего пользователя. Доступ к каталогу `/etc/tcb` на **использование** (а, следовательно, и ко всем его подкаталогам) имеют, кроме `root`, только члены *группы* `shadow`. Доступ на **запись** к каталогу `/etc/tcb/methody` и к файлу `/etc/tcb/methody/shadow` имеет только пользователь `methody`. Значит, чтобы изменить что-либо в этом файле, *процесс* **обязан** иметь *UID* `methody` и *GID* `shadow` (или нулевой *UID*, конечно). Именно такой *процесс* запускается из `/usr/bin/passwd`: он наследует *UID* у командного интерпретатора и получает *GID* `shadow` из-за *атрибута SetGID*. Выходит, что даже найдя в программе `passwd` ошибки и заставив ее делать что угодно, злоумышленник всего только и сможет, что... отредактировать **собственную** *учетную запись*!

Оказывается, *атрибут SetGID* можно присваивать каталогам. В последнем примере каталог `/etc/tcb/methody` имел *SetGID*, поэтому все создаваемые в нем файлы получают *GID*, равный *GID* самого каталога – `auth`. Используется это разнообразными *процессами идентификации*, которые, не будучи суперпользовательскими, но входя в *группу* `auth` и `shadow`, могут прочесть информацию из файлов `/etc/tcb/входное_имя/shadow`.

Вполне очевидно, что *подмена идентификатора* распространяется на **программы**, но не работает для **сценариев**. В самом деле, при исполнении сценария *процесс* запускается не из него, а из программы-интерпретатора. Файл со сценарием передается всего лишь как параметр командной строки, и подавляющее большинство интерпретаторов просто не обращают внимания на *атрибуты* этого файла. Интерпретатор наследует *UID* у родительского *процесса*, так что если он и обнаружит *SetUID* у сценария, поделать он все равно ничего не сможет.

Восьмеричное представление атрибутов

Все двенадцать *атрибутов* можно представить в виде битов двоичного числа, равных 1, если *атрибут* установлен, и 0, если нет. Порядок битов в числе следующий: `sU|sG|t|rU|wU|xU|rG|wG||xG|r0|w0|x0`, где `sU` – это *SetUID*, `sG` – это *SetGID*, t – это t-*атрибут*, после чего следуют три тройки *атрибутов* доступа. Этим форматом можно

пользоваться в команде `chmod` вместо конструкции "`роли=виды_доступа`", причем число надо записывать в **восьмеричной системе счисления**. Так, *атрибуты* каталога `/tmp` будут равны `1777`, *атрибуты* `/bin/su` – `4711`, *атрибуты* `/bin/ls` – `755` и т. д.

Тем же побитовым представлением *атрибутов* регулируются и *права доступа* по умолчанию при **создании** файлов и каталогов. Делается это с помощью команды `umask`. Единственный параметр `umask` – восьмеричное число, задающее *атрибуты*, которые **не надо** устанавливать новому файлу или каталогу. Так, `umask 0` приведет к тому, что файлы будут создаваться с *атрибутами* "`rw-rw-rw-`", а каталоги – "`rwxrwxrwx`". Команда `umask 022` убирает из *атрибутов* по умолчанию *права доступа* на запись для всех, кроме *хозяина* (получается "`rw-r--r--`" и "`rwxr-xr-x`" соответственно), а с `umask 077` новые файлы и каталоги становятся полностью недоступны ("`rw-------`" и "`rwx------`") всем, кроме их *хозяев*. [6]