МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Национальный исследовательский ядерный университет «МИФИ»

Факультет кибернетики и информационной безопасности
Кафедра 36 «Информационные технологии»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к учебно-исследовательской работе на тему:

Добавление новых классов доступа для защищенной операционной системы

Оценка: - _____

Выполнил: _____ Студент группы К8-361
В.Б. Холявин

Руководитель работы: _____ Доцент, кандидат технических наук
С.К. Муравьев

Члены комисии:

_____ / _____

_____ / _____

_____ / _____

_____ / _____

Москва, 2012 г.

АННОТАЦИЯ

Пояснительная записка к учебно-исследовательской работе: 51 страница, 1 рисунок, 1 таблица, список литературы из наименований.

Ключевые слова: *Linux, SELinux, защищённая операционная система, классы доступа.*

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Модуль безопасности SELinux для операционных систем семейства Linux добавляет к классической дискреционной системе контроля доступа систему мандатного контроля доступа. Оставаясь в рамках дискреционной системы контроля доступа, ОС имеет фундаментальное ограничение в плане разделения доступа процессов к ресурсам — доступ к ресурсам основывается на правах доступа пользователя.

SELinux позволяет ограничивать доступ работающих приложений только к тем ресурсам, которые им необходимы. Таким образом, если в приложении найдется уязвимость, злоумышленник не сможет получить доступ ни к каким ресурсам, за исключением тех которые приложение использовало для своей работы.

Существующая политика SELinux включает в себя множество классов, представляющих системные ресурсы системы (такие как файл или сокет). Однако элементов графического интерфейса пользователя нет среди классов доступа SELinux. Добавление подобных элементов может позволить использовать систему контроля доступа для разграничения возможностей пользователя при использовании графических приложений. Например, пользователь с соответствующими правами, прописанными в модуле политики SELinux сможет воспользоваться кнопкой или текстовым полем ввода, а пользователь без соответствующих разрешений - нет.

В процессе работы был изучен механизм добавления новых классов доступа в существующую версию политики SELinux. А так же была написана политика, позволяющая разграничивать доступом к объектам данного класса в зависимости от прав процесса, получающего доступ.

1. СТРУКТУРА ПОЛИТИКИ SELINUX

1.1. Reference политика

Изначально разработкой SELinux занималось Агентство национальной безопасности США, затем его исходные коды представлены для свободного скачивания. Компания Tresys продолжила разработку политик SELinux. Результатом ее работы стала Reference политика. Проект reference политики является в настоящее время наиболее популярным, на его основе разрабатываются политики под наиболее популярные операционные системы семейства Linux.

Основной целью проекта reference политики является попытка перестроить оригинальную политику и превратить ее в более лёгкую для использования, понимания и поддержки. В основе проекта reference политики лежат строгие принципы разработки, основанные на понятных принципах проектирования программного обеспечения. При этом reference политика сохраняет все наработки оригинальной политики, которые были получены при использовании ее в работе. Иными словами ее цель сохранить хорошее и исправить плохое.

Главным недостатком предыдущей версии политики является отсутствие разбиения политики на модули и тесная связь исходного кода модуля с финальной политикой. Несмотря на то, что макросы добавили абстракции в оригинальную политику, все идентификаторы политики (типы, роли, атрибуты и т.д.) являлись в действительности глобальными. Изменение одного модуля политики могло потребовать знание о многих других модулях. Взаимосвязь между модулями была очень тесная и плохо документированная. Создание нового модуля политики требовало детального понимания других модулей политики.

Вот некоторые ключевые характеристики reference политики, которые сделали разработку политики более простой:

1) Одно дерево исходных кодов, которое поддерживает strict и targeted политики, и опционально многоуровневая или многокатегорная защита (MLS, MCS). Политика состоит из одного файла, называемого монолит-

ной политикой. Новая система загрузки модулей.

2) Применение строгих принципов проектирования, заключающихся в основном в слабой связи модулей, с четко определёнными интерфейсами и не глобальное использование типов и других идентификаторов. (Так, например, все изменения, сделанные с типом, остаются в рамках одного модуля.)
3) Встроенная поддержка генерирования документации, захватывающая описания интерфейсов модулей. Так что разработчик модуля политики может использовать интерфейс без понимания того, как интерфейс реализован в модуле.

Помимо упрощения разработки политики, reference политика облегчает проверку свойств безопасности, например, для обеспечения сертификации, а так же для расширения поддержки развития инструментов, таких как интегрированная среда разработки и сложные отладчики политики.

Больше информации о reference политике, а так же последние ее исходные коды можно получить на web сайте проекта http://oss.tresys.com/projects/refpolicy. В последних версиях Fedora core targeted политика основана на reference политике.

1.2. Описание структуры исходных кодов политики

Структура reference политики отличается от оригинальной политики. Перед тем, как описывать основные детали реализации ссылочной политики, давайте рассмотрим расположение основных файлов и исходных кодах политики, для того что бы ознакомиться с ее файловой структурой.

1.2.1. Файлы для компиляции и вспомогательные файлы

Следующие файлы и директории используются для сборки или являются вспомогательными для reference политики:

`build.conf` Данный файл определяет ряд параметров сборки которые мы можем изменять. Данный файл включается в Makefile в процессе make. Некоторые из опций данного файлу будут рассмотрены ниже.

`Rules.modular` Данный файл содержит правила для сборки политики, которая поддерживает загрузку модулей. Он поддерживает сборку обоих базовых модулей политики и загружаемых модулей политики. Какие именно модули собирать как часть базового модуля, а какие как загружаемые определено в файле policy/modules.conf. Параметр сборки MONOLITHIC в build.conf контролирует модульная или монолитная политика будет собираться.

`Rules.monolithic` Если собирается монолитная политики, данный файл (вместо Rules.modular) включается в Makefile для определения сборки монолитной политики.

`config/` Эта директория содержит поддиректории с конфигурационными файлами для каждого варианта политики, которые мы может построить с reference политикой. Эти конфигурационные файлы обычно те же самые что и файлы в

`appconfig/` Эти файлы устанавливаются в директорию с политикой (`/etc/selinux/r`) для поддержки сервисов и приложений.

`doc/` Эта директория содержит файлы, которые поддерживают встроенную генерацию документации, которая является частью reference политики. Конечную документацию можно посмотреть на web сайте `http://oss.tresys.co` Или выполнив make html и заглянув в директорию

`doc/html/support/` Эта директория содержит исходные коды и скрипты для вспомогательных утилит, использующихся в процессе сборки политики.

1.2.2. Файлы ядра политики

`policy/constraints` в этом файле определены не MLS ограничения.

`policy/flash/` Эта директория содержит определения классов и разрешений.

`policy/mls` и `policy/mcs` два файла, определяющие конфигурацию для не обязательных MLS особенностей SELinux.

`policy/global_booleans` и `policy/global_tunables` два файла, содержащие определения Boolean переменных и их значения по умолчанию. Они компилируются и устанавливаются в `/etc/selinux/refpolicy/Booleans` и позволяют администратору изменять значения по умолчанию. Они используются для Условных политик. Причиной двух файлов является то, что в `global_booleans` содержатся переменные направленные действительно на поддержку условной политики, администратор может включать и выключать их в рабочей системе. `global_tunables` содержат переменные, которые являются конфигурацией сборки. Они изменяются один раз при установке политики и не меняются после.

`policy/modules.conf` Этот файл содержит информацию о том, какие модули включаются в процесс сборки и в какой форме. Модули могут собираться в монолитную политику или в базовые загружаемые модули, или вообще не собираться. Файл modules.conf создается после выполнения команды make conf.

`policy/modules/` Директория, содержащая все модули политики, разделенные в разные директории по слоям.

`policy/support` эта директория содержит макросы, используемые модулями для помощи при написании политики. Например файл `policy/support/obj_perm`
определяет макрос, который определят список разрешений.

`policy/users` этот файл содержит все определения пользователей для политики. Он содержит только основные определения. Обычно это `system_u`, `user_u` и иногда root.

1.3. Принципы разработки

Reference политика сформирована по нескольким принципам проектирования. Эти принципы сфокусированы на достижении основной цели проекта. Сейчас большинство из этих принципов выполняются толь по соглашению.

1.3.1. Слои

Одним из основных принципов проектирования политики является строгое разбиение ее на модули. Слабым, но не маловажным принципом reference политики является разбиение этих модулей на слои. Слои обеспечивают свободную организованную структуру для модулей, которые показывают общую архитектуру системы. В общем, в reference политике сделана попытка сохранить зависимость между модулями в пределах слоя и более низких слоев. Слои разнесены по каталогам в директории policy/modules/. В настоящее время в reference политике определены следующие слои:

Kernel – Ядро. Этот слой содержит модули политики, которые непосредственно относятся к ядру Linux. Это самый низкий уровень модулей. Модули в этой слое включают программные инструкции для ядра, устройств, файловой системы, сети. Большинство из этих модулей всегда включается в любой тип политики.

System – это модули, которые так же обычно включаются во все политики, но не обращаются к ядру напрямую. Модули в этом слое включаются основные библиотеки, процесс аутентификации и управление сетью.

Services – этот слой содержит модули политики для всех сервисов и демонов. В нем содержатся модули от cron, sshd до apache.

Admin – этот слой содержит модули политики для утилит администрирования и команд, у которых есть свой тип домена.

Apps – этот слой содержит модули политики для всех остальных программ, у которых есть свой собственный тип домена и модуль политики. Заметим, что разделение на слои не строгое, оно нужно в первую очередь для организации наборов модулей.

1.3.2. Модульность

Одним из принципов проектирования reference политики является модульность. В reference политике модули должны быть слабо связаны. Эта слабая связь

осуществляется через исполнение двух принципов проектирования: инкапсуляции и абстракции.

1.3.3. Инкапсуляция

Инкапсуляция является одним из принципов для достижения модульности. Она требует, что бы все типы и атрибуты использовались только в одном модуле. В результате, типы и атрибуты не могут быть использованы как глобальные. Только модули, которые определяют тип или атрибут могут на него ссылаться напрямую. Любой другой модуль должен запросить использование типа или атрибута через специальный интерфейс.

Например, в оригинальной политике, все типы, которые являются типами домена, получают атрибут domain. Поэтому в каждом модуле необходимо просто добавлять атрибут domain ко всем типам, которые являются типами домена. Если придется изменить концепцию, то придется изменять все модули.

В reference модуле, модуль domain в слое ядра определяет концепт домена. Эта концепция реализуется так же, как и в exemple политике, однако, если потребуется ее изменить, то изменять необходимо будет только в одном месте. Любой другой модуль, если хочет объявить один из своих типов доменом, вызывает интерфейс, определенный в модуле domain:

```
domain\_type(my\_type)
```

Этот интерфейс объявлен в `policy/modules/kernel/domain.if`.

Инкапсуляция позволяет делать реализацию модуля скрытой от остальных модулей с помощью интерфейсов.

1.3.4. Абстракция

Абстракция это цель разработки, когда интерфейс раскрывает то, что он делает, но не раскрывает как он это делает. Целью интерфейса является описание того, какой именно абстракций доступ предоставляется, или какая системная возможность включается и этим интерфейсом. Рассмотрим пример ранее показанного интерфейса domain_type(). Целью данного интерфейса является именно

превращение типа в тип домена. Добавление к типу атрибута domain является скрытой реализацией данного интерфейса, поэтому этот интерфейс не называется add_domain_attribute().

1.3.5. Файлы модулей

Как было сказано, когда обсуждалась файловая структура reference политики, все модули содержатся в

```
policy/modules/[layer]/
```

где layer является директорией с именем соответствующего слоя. Каждый модуль должен содержать три связанных файла, которые имеют одинаковое имя (без расширения), которое является именем модуля:

Внутренний файл политики (.te) Этот файл содержит внутренние объявления и правила модуля. Все объявления типов и атрибутов находятся в этом файле и правила, которые дают этим типам и атрибутам их права находятся здесь.

Внешний интерфейс (.if) Этот файл содержит интерфейс модуля. Этот интерфейс означает, какой доступ получат другие модули к типам и атрибутам данного модуля.

Маркировочный файл (.fc) Этот файл содержит инструкции маркирования файлов, относящиеся к данному модулю.

Только .te и .if файлы данного модуля могут использовать типы и атрибуты везде. Все другие модули должны использовать внутренние типы и атрибуты через интерфейсы модуля.

1.3.6. Интерфейсы

Как уже говорилось ранее, одним из наиболее значимых усовершенствований, добавленных в reference политику, является использование интерфейсов для получения доступа к типу из внешних модулей. Интерфейс предоставляет доступ к ресурсам модуля (например, приватных типов или атрибутов). Все остальные

модулю используют этот интерфейс для доступа к этим приватным типам или атрибутам. Таким образом, внутренние изменения модуля не вовлекают изменения других модулей, как это было в огиринальной политике.

Как отмечалось выше, интерфейсы являются частью модуля и находятся в .if файлах. Интерфейсы реализованы в виде макросов. В настоящее время reference политика поддерживает два вида интерфейсов: интерфейсы доступа и шаблонные интерфейсы.

2. СИНТАКСИС ЯЗЫКА ПОЛИТИКИ SELINUX

2.1. Добавление новых классов и разрешений

Классы и связанные с ними разрешения является основой контроля доступа в SELinux. Классы представляют собой категории ресурсов, такие как файлы или сокеты, а разрешения представляют собой виды доступа к этим ресурсам, такие как чтение ли отправка. Понимание классов и разрешений является трудным аспектом SELinux, так как требует знание как по SELinux так и по Linux.

Объекты классов представляют все ресурсы основных типов (например, файлы или сокеты). Экземпляры классов (например, конкретный файл или сокет) обычно называются просто объект. Иногда термины класс и объект класса используются взаимозаменяемо, но важно понимать различие между ними. Класса относятся к целой категории ресурсов, в то время как объекты указывают на конкретный экземпляр класса.

Рассмотрим правило allow:

```
allow user_t bin_t : file {read execute getattr};
```

В этом правиле процессу с типом `user_t` (который является источником или предметом) разрешается читать, выполнять и получать атрибуты от всех объектов класса файл, у которых указан тип `bin_t` и их защищённом контексте. Класса file указывается на категорию ресурсов, а `bin_t` указывает на экземпляр этой категории ресурсов, для которого это правило применимо (это те файлы, тип которых `file_t`). Это правило не применяется к тем объектам, у которых тип `bin_t` но класс отличен от file. Так же оно не применяется к файлам, у которых тип отличен он `bin_t`.

Разрешения в данном правиле read, execute и `getattr` определяют доступ, который разрешается к объекту от предмета, который имеет тип `user_t`. Каждое из этих разрешений должно быть действительно для указанного класса, и оно должно представлять некоторую форму доступа к этому объекту. (Например, разрешение read требует использования системных вызовов open(2) и read(2).) Набор

разрешений определенных для экземпляра класса (так же называемый вектором доступа) предоставляет все возможные разрешения, которые позволено выполнять над данным объектом.

Набор объектов классов зависят от версии SELinux и ядра Linux. Со временем добавляются новые классы, для того что бы обеспечить наиболее полное покрытие функционала ядра. Например, новые версии ядра Linux содержат новый вид сокета `netlink`, для логирования фрэймворка. Для таких ядер, которые поддерживают `netlink socket`, в SELinux добавлен соответствующий класс с необходимыми разрешениями.

Политика должна включать объявление всех классов и их разрешений, поддерживаемых ядром или другими мэнеджерами объектов. В действительности, для того что бы писать политики для конкретных программ нет необходимости добавлять новые классы, однако необходимо понимать структуру объявления классов, для эффективного написания модулей политики. Понимание синтаксиса объявления классов и разрешений, позволяет узнать какие классы и разрешения поддерживает данная версия политики.

2.1.1. Добавление новых классов и разрешений

Добавление новых классов и изменение их разрешений задача, требующая изменения системного кода. В отличие от других аспектов языка политик SELinux, классы и разрешения используются для отображения деталей Linux, в частности ядра. На деле получается, что классы и разрешения необходимы для того, что бы наиболее правильно передать структуру ресурсов системы. Поэтому соответствующие изменения в классах и разрешениях должны следовать за изменениями в системе.

Примером такого изменения может служить добавление новой формы межпроцессорного взаимодействия в ядро. В данном случае, добавляется новая категория ресурсов с новыми системными вызовами. И данные ресурсы требуют соответствующего представления в SELinux.

Простое добавление классов или разрешений, не приведет ни к какому эффекту. Требуется так же добавить поддержку данного класса в системный код, реализующий проверку доступа к объектам данного класса.

2.1.2. Объявление классов

Классы объявляется с помощью инструкции объявления классов, которая имеет следующий синтаксис:

```
class class_name
```

`class_name` – идентификатор класса. Идентификатор может быть любой длинны и содержать ASCII буквы и цифры.

Объявление класса описывает только класс и ничего больше. Например, следующая инструкция описывает объект класса для директории:

```
class dir
```

Инструкция содержит ключевое слово class и следующее за ним имя класса. Обратите внимание, что инструкция не заканчивается точкой с запятой, как многие другие инструкции. Имена классов объявляются в отдельном пространстве имен. Поэтому возможно объявить класс, разрешение и тип с одинаковым именем, однако это строго не рекомендуется.

Классы могут объявляться только в монолитных политиках или базовых загружаемых модулях. Объявления классов в не базовых загружаемых модулях не действительны. Классы нельзя объявлять в условных операторах.

2.1.3. Объявление и связывание разрешений с классами

Существует два метода для объявления разрешений. Первый называется «общие разрешения» и позволяет нам создавать разрешения, которые в последствии можно связать с несколькими классами. Общие разрешения используются для похожих классов (например, файлы и символические ссылки). Второй метод называется классовые разрешения. Он позволяет указать уникальные разрешения для конкретного класса. Как вы увидите, некоторые классы содержат только классовые разрешения, а некоторые только общие, и некоторые содержат и те и

те.

2.1.4. Общие разрешения

Инструкция, объявляющая общие разрешения позволяет создавать список разрешений, который можно связать как группу с двумя или более классами. Она имеет следующий синтаксис:

```
common common_name { perm_set }
```

`common_name` - идентификатор общих разрешений. Идентификатор может быть любой длинны и содержать ASCII символы, буквы, тире или точку.

`perm_set` – один или несколько идентификаторов разрешений, разделенных пробелами или переводом строки. Идентификатор может быть любой длинны, содержать ASCII буквы, цифры, тире или точку.

Объявление общих разрешений возможно только в монолитной политике и базовых загружаемых модулях. Объявления общих разрешений в условных выражениях или не базовых загружаемых модулях считается не действительным.

Из UNIX философии «все есть файл» вытекает то, что многие объекты доступа имеют одинаковые наборы разрешений. В данном случае удобно применять общие разрешения. Следующий пример объявляет общие разрешения, связанные с файлами:

```
common file
{
      ioctl
      read
      write
      create
      getattr
      setattr
      lock
```

```
        relabelfrom
        relabelto
        append
        unlink
        link
        rename
        execute
        swapon
        quotaon
        mounton
}
```

Данная инструкция объявляет общие разрешения, названные file. Только объявление общих разрешений не имеет смысла. Их необходимо связать с классом которых их использует.

Как и с классами, общие разрешения объявляются в своем собственном пространстве имен. Это может привести к некоторым проблемам, если не быть осторожными. Например, как показано в предыдущем примере, у нас есть класс и общее разрешение, названные file. Несмотря на то, что имя одинаковое, на деле это два разных и сильно отличающихся компонента политики.

2.1.5. Связывание разрешений и классов

Связывание разрешений и классов происходит с помощью инструкции вектора доступа. Она имеет следующий синтаксис:

```
class class_name [ inherits common ] [{ perm_set } ]
```

`class_name` – объявленный ранее идентификатор класса

`common` – объявленный ранее идентификатор общих разрешений

`perm_set` – одно или несколько классовых разрешений. Идентификаторы классовых разрешений могут быть любой длины и содержать ASCII буквы, цифры, тире или точку.

Как минимум одно разрешение должно быть указано в инструкции, но могут быть указаны и общие разрешения и классовые вместе в одной инструкции. Результатом будет пересечение множества всех разрешений.

Объявление инструкций вектора доступа действительно в монолитных политиках и базовых загружаемых модулях. Их нельзя объявлять в условных выражениях и не базовых загружаемых модулях.

Следующие пример показывает инструкцию связи одного единственного классового разрешения с классом dir:

```
class dir { search }
```

Как показывает данный пример, инструкция вектора доступа выглядит похоже на инструкцию объявления классов (то же ключевое слово class). Объявление классов и инструкция вектора доступа являются различными инструкциями, начинающимися с одного ключевого слова. Для инструкции вектора доступа необходимо, что бы класс был объявлен заранее. Обратите внимание, что инструкция не заканчивается точкой с запятой.

В предыдущем примере было объявлено единственное разрешение, связанное с классов. В действительности их может быть несколько:

```
class dir { search add_name remove_name }
```

В данном примере связывается три классовых разрешения с классом dir. Мы может так же связать общие разрешения, используя ключевое слово inherits в инструкции вектора доступа. Например, класс dir является одним из так называемых «файловых» классов. И он включает общие разрешения, свойственные всем файловым классам. Следующая инструкция вектора доступа является полной инструкцией для класса dir, она связывает с классом объявленные ранее общие разрешения file и несколько классово уникальных разрешения для директорий:

```
class dir
inherits file
{
      add_name
  remove_name
      reparent
      search
      rmdir
}
```

Как показано в этом примере, мы использовали ключевое слово inherits, следующее за именем класса, объявленного ранее, далее идет идентификатор общих разрешений (file). В результате с классом dir будут связаны все разрешения из общих разрешений file и 5 классово уникальных разрешений.

Возможно так же использовать только общие разрешения, как в следующем примере:

```
class lnk_file inherits file
```

Данный пример связывает с классом lnk_file только общие разрешения файлов.

2.2. Правила вектора контроля доступа

Вектор доступа определяют свои значения в зависимости от разрешений для объекта класса. Язык политики SELinux сейчас поддерживает четыре типа правил вектора доступа:

`allow` – определяет доступ разрешения между двумя типами

`dontaudit` – определяет отказ в доступе

`auditallow` – определяет разрешение на запись события

`neverallow` – определяет те права доступа, которые не могут быть предоставлены в любом из модулей.

Мы рассмотрим каждое из этих правил, их общий и уникальный синтаксис и семантику и примеры их использования.

2.2.1. Общий синтаксис правил вектора доступа

Несмотря на то что каждое правило вектора доступа имеет различные цели, у них у всех общий синтаксис. Каждое правило состоит из пяти элементов:

Имя правила – `allow` , `dontaudit` , `auditallow`, `neverallow`

Тип источника – тип которому предоставляется доступ, обычно это тип домена процесса который пытается получить доступ.

Целевой тип – тип(ы) объекта, на который источник получает доступ.

Класс объекта(ов) – класс(ы) объекта(ов) , которому предоставляется доступ.

Разрешение – конкретные права доступа, указывающие что именно разрешается источнику.

Простое правило вектора доступа содержит один тип источника, один целевой тип, класс объекта и разрешение. Пример такого правила:

```
allow user_t bin_t : file execute;
```

Это правило `allow` содержит тип источника `user_t`, целевой тип `bin_t`, объект класса file, и разрешение execute. Это правило следует читать как «Разрешить `user_t` запускать файлы типа `bin_t`.

Все четыре правила вектора доступа обычно имеют похожий синтаксис с разными именами ключевых слов. Например, мы можем преобразовать предыдущий пример в auditallow правило, просто заменив ключевое слово:

```
auditallow user_t bin_t : file execute;
```

О том, что значит данное правило будет сказано ниже. Что важно в данный момент, так это то, что синтаксис действительно одинаковый.

2.2.2. Ключи вектора доступа

Внутри ядра все правила вектора доступа уникально идентифицируются триплетов из типа источника, целевого типа и объекта класса. Этот триплет называется ключом, используемым для хэш таблицы и содержат ключ внутри данных политики. Правила ищутся и хранятся по данному ключу. Когда процесс совершает запрос на доступ, модуль LSM SELinux запрашивает доступ основываясь на этом ключе.

Итак, что случится когда появляется несколько векторов доступа с одинаковым ключом (т.е. содержат тот же тип источника, целевой тип и объект класса). Например, рассмотрим политику со следующими правилами:

```
allow user_t bin_t : file execute;
allow user_t bin_t : file read;
```

Процессу типа `user_t` позволяется чтение или выполнение файлов типа `bin_t`? Позволяется и то и то. Все правила с одинаковым ключом комбинируются при выполнении checkpolicy. Скомпилированная политика содержит единственное правило с обоими и execute и read разрешениями. Все правила вектора доступа складываются таким образом.

Каждое последующее правило вектора доступа в политике, которое имеет те же ключи что и предыдущие, добавляет разрешения на конечное правило в собранную политику.

Не существует понятия удаления разрешения, предоставленного другим правилом. Так что будьте осторожны, вы можете написать одно разрешения в одной части политики, другое в другой, а в конечной политике будут содержаться оба разрешения.

2.2.3. Использование атрибутов в правилах вектора доступа

Несмотря на то, что правила вектора доступа, которые мы видели до сих пор были простые, синтаксис поддерживает множество способов перечисления ти-

пов, объектов класса и разрешений. Это позволяет нам быть более гибкими при создании политик и делать инструкции более короткими.

В простой форме правила в предыдущем примере, правило упоминает непосредственно тип источника и целевой тип. Но часто бывает удобно, однако, указать несколько исходных или целевых типов. Одним из способов для обозначения различных типов является использование атрибутов. Атрибуты можно использовать везде, где можно использовать тип в правиле вектора доступа.

Например, предположим, что мы определили атрибут `exec_type`, который мы планируем связать со всеми типами программ обычного пользователя (тип домена `user_t`) для того, что бы разрешить ему запуск. Теперь мы можем поменять предыдущий пример и сослаться на атрибут `exec_type` вместо явного типа `bin_t`, как показано здесь:

```
allow user_t exec_type : file execute;
```

В отличие от предыдущего примера, это правило не прямо отражает то, что будет отражаться в ядре. Правила, которые включают атрибуты будут развернуты внутри ядра в отдельное правило для каждого типа, связанного с атрибутом. Если типов было 20, то в конечной политике появится 20 ключей, предоставляющих доступ каждому из типов.

Мы так же можем использовать атрибуты в качестве типа источника, или вместо обоих типов. Например, предположим, мы также создали атрибут (domain) который мы связали со всеми нужными типами доменов (включая `user_t`), и которым мы предоставляем доступ на выполнение файлов с атрибутом `file_type`, Мы можем достичь этой цели одним правилом:

```
allow domain exec_type : file execute;
```

Для того что бы лучше проиллюстрировать концепцию расширения правил, представим, что наша политика, связывает атрибут типов `user_t` и тип `staff_t` с типами `exec_type`, `bin_t`, `local_bin_t` и `sbint_t`. Таким образом, одно правило эквивалентно следующими правилам:

```
allow user_t bin_t : file execute;
allow user_t local_bin_t : file execute;
allow user_t sbin_t : file execute;
allow staff_t bin_t : file execute;
allow staff_t local_bin_t : file execute;
allow staff_t sbin_t : file execute;
```

2.2.4. Множественные типы и атрибуты в правиле вектора доступа

Мы не ограничиваем одним типом или атрибутом тип источника или целевой тип. Мы можем указывать список типов или атрибутов как для типа источника так и для целевого типа. Список разделяется пробелами и заключен в фигурные скобки, как показано в следующем примере:

```
allow user_t { bin_t sbin_t } : file execute;
```

В этом правиле, целевым типом являются являются оба и `bin_t` и `sbint_t` типа. Правило с несколькими типами или атрибутами преобразуются в несколько правил на этапе компиляции. В предыдущем примере, скомплированная политика будется содержать два ключа, для каждого из целевых типов.

Мы можем использовать списки типов или атрибутов как для целевого типа так и для типа источника, или для обоих сразу. Например, следующее правило совершенно законно:

```
allow {user_t domain} {bin_t file_type sbin_t} : file execute ;
```

Политика ядра будет содержать по одному ключу для каждой комбинации исходного типа и целевого типа.

2.2.5. Специальный тип self

Язык политики содержит ключевое слово self, которое используется в качестве целевого типа. Например, следующие два правила эквивалентны:

```
allow user_t user_t : process signal;
allow user_t self : process signal;
```

Ключевое слово self указывает политике предоставить доступ для каждого исходного типа самому себе. В предыдущем примере второе правило всего лишь создает ключ, в котором целевой и исходный типы `user_t`.

Рассмотрим более сложный пример:

```
allow {user_t staff_t} self : process signal;
```

В этом примере правило создает два ключа, по одному на каждый из исходных типов. Это правило эквивалентно двум следующим:

```
allow user_t user_t : process signal;
allow staff_t staff_t : process signal;
```

Обратите внимание, что когда используете ключевое слово self, эквивалентные правила создаются только для каждого исходного типа. Т.е. `user_t` не получает доступ к `staff_t` и наоборот.

Так же стоит обратить внимание, что ключевое слово self может использоваться только на месте целевого поля в правиле вектора доступа. В частности, вы не можете использовать self как исходный тип. Кроме того вы не можете объявить тип или атрибут с идентификатором self.

Использование self особенно ценно, при использовании атрибутов или больших списков типов в качестве источника правила. Например, предположим, мы хотим, что бы каждый домен имел возможность посылать самому себе сигнал. Мы могли бы написать правило подобное этому:

```
allow domain domain : process signal;
```

Но это не то, что мы хотим в действительности. В данном примере кроме того, что каждый домен может посылать сигнал себе, он так же может посылать сигнал и каждому процессу из домена domain. Используя ключевое слово self можно исправить данную проблему:

```
allow domain self : process signal;
```

2.2.6. Специальный оператор отрицания

Последним элементом синтаксиса для типов в правиле вектора доступа является оператор отрицания. Этот синтаксис позволяет удалить из списка тип и наиболее часто используется для удаления типа из атрибута в данном правиле. Это осуществляется, путем указывания перед типом знака минус: -. Например, мы можем позволить всем типам домена выполнять файл атрибута `exec_type` за исключением `sbin_t` в следующем правиле:

```
allow domain { exec_type -sbin_t } : file execute;
```

Это правило будет выполняться, даже если атрибут `exec_type` не содержит тип `sbin_t`. Порядок указания типа и атрибута неважен. Следующее правило эквивалентно предыдущему:

```
allow domain { -sbin_t exec_type } : file execute;
```

2.2.7. Указание класса объекта и разрешений в правиле вектора доступа

Правило вектора доступа может так же содержать список классов и разрешений. Синтаксис аналогичен описанию типов, список разделенный пробелами и заключенный в фигурные скобки.

```
allow user_t bin_t : { file dir } { read getattr };
```

Это правило в результате преобразуется в два ключа, по одному на каждый класс, так же как с типами источника или целевыми типами.

```
allow user_t bin_t : file { read getattr };
allow user_t bin_t : dir { read getattr };
```

Обратите внимание, что список классов разширяется, но каждое правило содержит обинаковый список разрешений. Это значит, что все указанные разрешений

должны быть действительны для всех классов, указанных в правиле. Иногда приходится создавать два одиночных правила с одинаковыми исходными и целевыми типами, потому что классы указанные в правиле, имеют несовместимый набор разрешений. Например, если мы посмотрим на разрешений классов file и dir мы заметим, что большинство из них одинаковые, но некоторые нет. (Разрешения действительные для обоих являются результатом использования общих разрешений).

Предположим, например, мы хотим написать правило предоставляющее доступ на чтение обоим классам. Следующее правило не является правильным:

```
allow user_t bin_t : { file dir } { read getattr search };
```

Несмотря на то, что read и getattr являются общими разрешениями для обоих классов, разрешение search есть только у класса dir. Из-за этого checkpolicy не может создать ключ, который предоставит классу file разрешение search, и мы получим ошибку, когда попытаемся скомпилировать это правило. Нам придется создать два правила, такие как следующие:

```
allow user_t bin_t : file { read getattr };
allow user_t bin_t : dir { read getattr search } ;
```

2.2.8. Специальные операторы разрешений

Мы можем использовать два специальных оператора для указания разрешений в правиле вектора доступа. Первый оператор звездочка *. Означает все разрешения, доступные данному классу:

```
allow user_t bin_t : { file dir } *;
```

Это правило разворачивается в все разрешения для классов file и dir.

Как видно из примера, данный оператор отличается от простого перечисления тем, что может указывать разные разрешения для разных классов. Это позволяет использовать оператор * в правилах с несколькими классами, даже если классы имеют различные разрешения. Таким образом, правило выше предоставляет все разрешения для класса dir и все разрешения для класса file.

Второй специальный оператор позволяет указать все разрешения на исключением какого-либо списка, благодаря оператор тильда ( ):

```
allow user_t bin_t : file ~{ write setattr ioctl };
```

При компиляции данное правило предоставлет все разрешения класса file за исключением write, setattr, ioctl. Так же как и оператор *, данный оператор предоставляет доступ индивидуально для каждого из указанных классов.

Имейте в виду, что разрешено использовать данные специальные операторы и в других случаях. Данная возможность появилась в последних версиях компилятора. Многие последние версии checkpolicy позволяют, например, использовать оператор * для типов.

2.2.9. Общий синтаксис правил вектора доступа

Полный общий синтаксис следующий:

```
rule_name  type_set  type_set : class_set  perm_set ;
```

`rule_name` – Имя правила. Позволяется использовать allow, auditallow, auditdeny, dontaudit, neverallow ключевые слова.

`type_set` – Один или несколько типов или атрибутов. Задаются отдельные списки для исходных типов и целевых типов. Несколько типов и атрибутов разделяются пробелами и заключаются в фигурные скобки, например `{bin_t,sbin_t}` Типы могут быть исключены из списка, путем использования оператора -. Например,`{exec_type-sbin_t}`. Ключевое слово self может использовать в качестве целевого типа и не может в качестве исходного. Правило neverallow так же позволяет использовать оператор * для включения всех типов и оператор для включения всех типов за исключением списка.

`class_set` – Один или несколько классов. Несколько классов разделяются пробелами и заключаются в фигурные скобки, например `{file, lnk_file}`.

`perms_set` – Одно или несколько разрешений. Все разрешения должны быть действительными для всех объектов класса, указанных в `class_set`. Несколько разрешений должны быть заключены в фигруные скобки, например `{read create}`

Оператор * означает список из всех разрешений, доступных данному классу. Оператор означает список из всех разрешений, за исключением указанных.

Все правила вектора доступа можно указывать в монолитной политике, базовых загружаемых модулях и не базовых загружаемых модулях. Все правила, за исключением auditdeny и neverallow могут быть использованы в условных операторах.

2.2.10. Правило allow

Вы могли уже не раз видеть примеры данного правила в этой работе. Правило allow наиболее популярное правило в политиках.

Как уже рассказывалось, правило allow указывает все разрешения, которые доступны домену. Запомните, никакое разрешение не дается по умолчанию. Мы явно указать разрешения между двумя списками типов – исходных и целевых, для списка заданных классов, например:

```
allow user_t bin_t : file { read execute };
```

Данное правило позволяет любому процессу, чей контекст безопасности содержит тип `user_t`, разрешить чтение и выполнение любого обычного файла, у которого тип `bin_t`. Правило allow поддерживает весь общий синтаксис правил вектора доступа и не имеет ни какого дополнительного синтаксиса.

Как и все правила вектора доступа, allow правило в загруженной политике является пересечением всех правил с одинаковым ключом (объект, цель, класс). Например, следующие две инструкции:

```
allow user_t bin_t : file read;
allow user_t bin_t : file write;
```

Эквиваленты одной:

```
allow user_t bin_t : file { read write );
```

2.2.11. Правило audit

SELinux содержит широкие возможности для регистрации и проверки попыток доступа, которые разрешены или запрещены политикой. Сообщения проверки, так же называемые «AVC сообщения» дают детальную информацию о попытке доступа, разрешен был доступ или нет, контекст домена и контекст цели, и другую детальную информацию о ресурсах, вовлеченных в попытку доступа. Сообщения, которые похожи на другие сообщения в ядре, обычно сохраняются в лог файл в директории /var/log. Они являются незаменимым инструментом для разработки, системного администрирования и мониторинга. Правило audit указывает для каких попыток доступа следует генерировать сообщения.

По умолчанию SELinux не записывает никакие попытки доступа, для которых доступ был разрешен, однако записывает все разрешенные попытки доступа. Данное поведение оправдано, так как на многих система обрабатываются тысячи обращений проверки доступа в секунду, и только нескольким из них доступ запрещается. Язык политик позволяет частично переписать данное поведение по умолчанию. Благодаря этому можно генерировать сообщения о некоторых разрешенных попытках доступа, и, наоборот, не генерировать сообщения об определенных запретах доступа.

В SELinux существует два правила, которые позволяют нам контролировать логирование попыток доступа: dontaudit и auditallow. Эти два правила позволяют нам изменять настройки по умолчанию логирования попыток доступа. Наиболее часто используется правило dontaudit. Оно указывает те события, которые не стоит записывать в лог, несмотря на то, что доступ был запрещен.

Сообщения о запрете доступа генерируются только тогда, когда доступ запрещен модулем SELinux, который является LSM модулем. Проверка в LSM модуле осуществляется только тогда, когда стандартная проверка доступа Linux доступ

разрешила. Это означает, что если доступ заблокирован стандартной проверкой доступа Linux, то сообщение о запрете доступа не будет сгенерировано. Однако, если необходимо отлавливать и подобные случаи, то можно использовать напрямую систему логирования ядра, включенную в 2.6.x версии ядра. Для большей информации смотрите мануал по auditd(8) и auditctl(8).

Рассмотрим пример отмены логирования сообщения о запрете доступа:

```
dontaudit httpd_t etc_t : dir search;
```

Данное правило указывает, что когда процессу типа httpd_t запрещено производить поиск в директории типа etc_t, сообщение о запрете не будет сохраняться, несмотря на то, что по умолчанию данное сообщение должно быть сохранено. Правило dontaudit обычно используется в тех случаях, когда запрет доступа приложению является ожидаемым. Оно позволяет избежать большого количества сообщения в системных журналах.

Другое правило логирование, auditallow, позволяет нам записывать разрешенные попытки доступа, которые не логируются по умолчанию. Например, давайте посмотрим на следующее правило:

```
auditallow domain shadow_t : file write;
```

Данное правило показывает, что когда процесс с типом domain получает доступ на запись в файл типа shadow_t , сообщение о данном сообщении будет записано в лог. Правило auditallow используется для логирования важных событий, например доступа на запись к файлам с паролями, или загрузку новой политики в ядро.

Запомните, что правила логирования перезаписывают поведение по умолчанию. Правило allow указывает, какой доступ разрешен, а какой нет. Auditallow не выполняет данных действий, оно только включает запись событий.

Обратите внимание, что логирование отличается в permissive и enforcing режимах. Когда запущен enforcing режим, сообщения о запрете доступа генерируются каждый раз, когда возникает событие, требующее логирования. В permissive

режиме сообщения генерируются только один раз для каждого события, до следующей загрузки политики или до включения режима enforcing. Permissive режим наиболее часто используется для разработки модулей политики и данное поведение позволяет уменьшить размер логов.

2.2.12. Правило neverallow

Последним правилом вектора доступа является neverallow. Данное правило используется для указания тех правил доступа, которые никогда нельзя будет указать с помощью правила allow. По умолчанию доступ запрещен для всех тех случаев, которые не разрешены правилом allow. Правило neverallow существует не для запрета доступа, а для запреда создания соответствующих правил allow, для того, что бы явно заблокировать нежелательные разрешения. Напомним, что политика может содержать десятки тысяч правил доступа. Поэтому можно случайно указать тот доступ, который указывать не следуют. Правило neverallow помогает предотвратить подобные случаи.

Рассмотрим пример правила:

```
neverallow user_t shadow_t : file write;
```

Данное правило предотвращает добавления правила доступа, которое позволяет домену user_t писать в файлы типа shadow_t, генерируя ошибку компиляции. Данное правило не удаляет доступ, оно только генерирует ошибку компиляции.

Правило neverallow поддерживает некоторый дополнительный синтаксис, который расширяет общий синтаксис правил вектора доступа. В частности можно использовать операторы * и в поле указания целевого и исходного типов. Данные операторы работают так же как и со списком разрешений в других правилах вектора доступа.

Например, следующее правило:

```
neverallow * domain : dir ~{ read getattr };
```

Данное правило указывает, что не может быть ни каких правил allow которые разрешают любому типу домена любой доступ к директории типа, включенного

в атрибут domain, за исключением доступа на запись и получения атрибутов. Звездочка в данном случае означает все типы.

Правило neverallow, похожее на это используется в политике для предотвращения нежелательного доступа к директориям /proc/ которые содержат информацию о процессах.

Другой вариант правила следующий:

```
neverallow domain ~domain : process transition;
```

Данное правило указывает, что процесс типа с атрибутом domain, не может переходить в тип, который не имеет атрибута domain.

2.3. Правила типов

Правила типов определяют правила создания или изменения типов объектов. Существует два правила, определенных в языке политик:

`type_transition` – определяет поведение по умолчанию для перехода процесса из домена в домен и создания объектов.

`type_change` – задает правила изменения типа объектам из SELinux приложений.

В отличие от правил вектора доступа, данные правила вместо списка разрешений определяются тип.

2.3.1. Общий синтаксис правил типов.

Как и правила вектора доступа, правила типов имеют некоторый общий синтаксис. Каждое правило содержит пять элементов:

Тип правила – `type_transition` или `type_change`

Исходный тип – исходный тип процесса

Целевой тип – тип объекта, для которого создается новый тип или меняется старый Класс объекта – класс объекта, тип которого создается или меняется

Тип по умолчанию – единственный тип, для нового объекта или измененного.

Большая часть синтаксиса похожа на синтаксис правил вектора доступа. Но существуют важные различия. Первое – нет списка разрешений. В отличие от

правил вектора доступа, правила типа не предоставляют доступ или логировние, поэтому им не нужны разрешения. Вторым важным отличием является то что класс не связан с целевым типов. Класс связан с объектом, который которому будет назначен тип по умолчанию.

Простым примером правила типа является следующий:

```
type_transition user\_t passwd\_exec\_t : process passwd\_t;
```

Данное правило показывает, что когда процесс типа `user_t` запускает файлы типа `passwd_exec_t`, тип процесса пытается измениться, по умолчанию, на `passwd_t`. Целевой тип связан с классом file, тогда как запускаемый класс process. Запускаемый класс (process) связан с исходным типом и типом по умолчанию. Эту ассоциацию легко не заметить, даже если вы уже опытный разработчик политик.

Как и с правилами вектора доступа мы можем указывать несколько классов, используя разделенный пробелами список, заключенный в фигурные скобки. Так же мы можем использовать атрибуты и списки типов в полях целевого и исходного типов:

```
type_transition { user_t sysadm_t } passwd_exec_t : process passwd_t;
```

Данное правило включает два типа, `user_t` и `sysadm_t`, в списке исходных типов. Как и правила вектора доступа, данное правило будет преобразовано в два правила. Предыдущий пример имеет тот же смысл что и следующие два правила:

```
type_transition user_t passwd_exec_t : process passwd_t;
type_transition sysadm_t passwd_exec_t : process passwd_t;
```

Использование атрибутов работает так же как в правилах вектора доступа.

В отличие от целевого и исходного типов, в качестве типа по умолчанию не может использовать атрибут или список типов. Если определить в качестве типа по умолчанию несколько типов, то ядро не сможет выбрать какой именно тип присвоить объекту после выполнения данного правила. Из-за этого не может быть

более одного правила с одинаковыми исходным типов, целевым типов и классом. Следующие два примера вызовут конфликт:

```
type_transition user_t passwd_exec_t : process passwd_t;
type_transition user_t passwd_exec_t : process user_passwd_t;
```

Компилятор политики сгенерирует ошибку для обоих из этих правил, присутствующих в политике.

2.3.2. Общий синтаксис:

```
rule_name  type_set  type_set : class_set  default_type;
```

`rule_name` – имя правила типа.

`type_set` – один или несколько типов. Это список исходных и список целевых типов. Несколько типов или атрибутов разделены пробелами и объединены в фигурные скобки, например `{ bin\_t sbin\_t }`. Типы могут быть исключены из списка с помощью оператора – перед именем типа (например `{ exec\_type -sbin\_t }`).

`class_set` – один или несколько классов. Несколько классов разделены пробелами и объединены в фигурные скобки, например `{ file lnk\_file}`.

`default_type` – один тип, который задается по умолчанию для создаваемого объекта или для изменяющегося объекта. Атрибут или список типов не может быть использован.

Все правила типов действительны в монолитных политиках, базовых загружаемых модулях и не базовых загружаемых модулях, а так же в условных выражениях.

2.3.3. Правила перехода типов

Правило type_transition используется для того, что бы указать тип по умолчанию для определенных событий. В настоящее время существует две формы правила type_transition. Первая обрабатывает события перехода из домена в домен.

Вторая форма этого правила обрабатывает переход объектов, которая позволяет нам указать явно тип по умолчанию для новых объектов.

Обе формы помогают быть расширенной безопасности SELinux быть более прозрачной для обычного пользователя. В SELinux по умолчанию, вновь созданные объекты в директории наследуют тип объекта, содержащего их (например, директории), а процесс наследует тип родительского процесса. Правила typte_transition позволяет нам изменить данное поведение по умолчанию. Это используется, к примеру, для того, что бы когда программа работы с паролями создает свои файлы в директории /tmp, эти файлы имели отличный тип от тех, которые создает обычный пользователь.

Правила type_transition не предоставляют доступ. Они обеспечивают только новое правило маркирования объектов по умолчанию. Успешное событие перехода типа всегда требует связанного с ним набора allow правил, которые позволяют создавать объекты и маркировать их. Добавим, что поведение по умолчанию, которое описывается в правиле type_transition производит какой-либо эффект, только если созданный процесс не переопределяет данное поведение.

2.3.4. Правила по умолчанию для перехода доменов

Рассмотрим подробнее одну из форм правила type_transition, а именно правила перехода доменов. Домен изменяет тип, когда запускает файл на выполнение. Например, взглянем на следующий пример:

```
type_transition init_t apache_exec_t : process apache_t;
```

В данном type_transition правиле говорится, что когда процессы типа init_t выполняют файл типа apache_exec_t, тип процесса должен измениться на apache_t. Класс в данном правиле указывает, что это правило перехода домена. Правила перехода домена на самом деле изменяют тип существующего процесса вместо маркирования вновь созданного. Это связано с тем, что в Linux новый процесс создается вначале точной копией родительского процесса с помощью вызова fork().

Последние версии SELinux добавляют к классу process разрешение dyntransition. Это разрешение, которое было добавлено в основном для совместимости с другими системами, позволяет процессу изменять тип домена еще в запросе, а не только после выполнения. Данный тип перехода процесса не является безопасным, поскольку он позволяет вызывающему домену выполнять произвольный код в новом домене, стирая различия между двумя доменами. Рекомендуется никогда не использовать данное разрешение в политике, за исключением случаев, когда вы точно уверены что без этого не обойтись.

Как было замечено ранее, переход типа может произойти только если политика позволяет соответствующий доступ. Для успешной смены домена политика должна включать три разрешения:

Execute – исходный тип должен иметь разрешение на выполнение файлов целевого типа.

Transition – исходный домен должен иметь разрешение на переход к типу по умолчанию

Entrypoint – новый домен (по умолчанию) должен иметь entrypoint разрешение на файлы с целевым типом apache_exec_t.

Таким образом, правило перехода домена, указанное выше, требует следующих allow правил, для успешного выполнения:

```
allow init_t apache_exec_t : file execute;
allow init_t apache_t : process transition;
allow apache_t apache_exec_t : file entrypoint;
```

На практике обычно данные правила стоит дополнить еще несколькими. Например, разрешение уведомлять исходный тип из домена типа по умолчанию при его завершении (это sigchld разрешение), наследование дескрипторов файлов, а также разрешение на общение процессов через pipe.

2.3.5. Правила перехода объектов

Правила переходов объектов указывают тип по умолчанию для вновь созданных объектов. На практике вы обычно используем данную форму правила time_transition в основном для объектов файловой системы (например, file, dir, lnk_file). Так же как и переход доменов, эти правила указывают только тип по умолчанию для перехода. Успешность перехода зависит от того, есть ли в политике необходимые правила allow.

Пример правила перехода объектов:

```
type_transition passwd_t tmp_t : file passwd_tmp_t;
```

Данное type_transition правило означает, что когда процесс типа passwd_t создает обыкновенный файл (file object class) в директории типа tmp_t, файлу, по умолчанию, присваивается тип passwd_tmp_t, если это разрешено политикой. Обратите внимание, что в данном случае класс связан только с типом по умолчанию (passwd_tmp_t). В данном правиле, tmp_t неявно связаны с объектом класса dir, потому что это единственный класс, который может содержать файлы. Также, как и прежде, политики должны разрешить маркирование файлов, для того что бы задание типа файлу произошло. Доступ, необходимый для того, что бы маркирование было выполнено включает разрешения add_name, write и search для директории типа tmp_t, и разрешения write и create для файлов типа passwd_tmp_t.

Данный пример показывает технику для решения проблем безопасности в директориях, которые предназначены для файлов многих приложений, таких как временные директории. Правила перехода типов используются для любых объектов, которые создаются при выполнении и которые должны иметь тип отличный от родительского.

При некоторых условиях правила перехода типов не могут быть исполнены. Когда процессу необходимо создать файл с различными типами в одном контейнере, type_transition правила не достаточно. Например, рассмотрим процесс,

который создает два сокета и директории /tmp, которая используется многими другими доменами для общения. Если мы хотим, что бы файлы сокета имели различный тип, правила type_transition будет не достаточно. Мы получим два правила с одинаковыми типами источника, назначения и классом, но разным типами по умолчанию, что приведет к ошибке компиляции. Решением данной проблемы может быть создание файлов сокета при установке и четкого обозначения типов для них, или создания сокетов в различных директориях, или непосредственный запрос на задание типа сокета.

2.3.6. Правила изменения типа

Правила type_change используется для того, что бы указать типы по умолчанию для переобозначения в SELinux-совместимых приложениях. Как и type_transition правила, правила type_change определяют только маркировку по умолчанию, но не разрешают доступ. В отличие от type_transition правил, результат type_change не отражается в ядре, он необходим для пользовательских приложений, каких как login или sshd, которые меняют метки на основе политики. Например, рассмотрим следующее правило:

```
type_change sysadm_t tty_device_t : chr_file sysadm_tty_device_t;
```

Это type_change правило показывает, что когда происходит процедура перемаркирования файлов от имени sysadm_t файлы класса chr_file должны получить тип sysadm_tty_device_t. Данное правило является наиболее распространенным правилом type_change в политике. Оно необходимо для перемаркирования терминала при входе пользователя в систему. При входе в программу она делает запрос политики через интерфейс модуля ядра SELinux, передавая типы sysamd_t и tty_device_t и получая sysadm_tty_device_t как результирующий тип. Этот механизм позволяет при входе в систему для обозначения устройства терминала от имени пользователя использовать политики заключенные в ядро, а не жестко задавать тип в программе.

2.3.7. Правило type_member

Компилятор политики так же поддерживает третье правило типов, type_member. В настоящее время это правило не имеет смыслового значения, и его использование не произведет никакого эффекта. Работа по этому правилу еще не завершена. Одна в будущем данное правило будет предназначено для указания типов объектам polyinstantiated. Синтаксис правила будет аналогичен двум другим правилам типов.

3. ПРОЦЕСС ДОБАВЛЕНИЕЯ НОВОГО КЛАССА ДОСТУПА

3.1. Настройка окружения и получение исходных кодов политики

Реализации SELinux модуля существуют для большинства популярных дистрибутивов Linux. Для Red Hat Enterprise Linux SELinux доступен с коммерческой поддержкой, для остальных дистрибутивов поддержка производится сообществом. В работе использовался дистрибутив Fedora 16, как один из самых популярных дистрибутивов Linux. Второй причиной использования именно этого дистрибутива, является то, что по умолчанию в дистрибутив уже включен модуль SELinux, а так же targeted политика. Так же для данного дистрибутива существует множество утилит для администрирования, настройки и написания политик и их модулей.

В дистрибутиве Fedora 16 по умолчанию присутствует политика SELinux, однако исходные коды ее необходимо скачивать отдельно. Как уже говорилось в первой части, для работы выбрана версия политики от компании Tresys technology, которая называется Reference Policy. Причиной данного выбора стало то, что политика используемая в выбранном дистрибутиве основана на reference политике, а так же то, что разработка данной политики продолжается и в нее добавляется новый функционал и правки. Исходные коды политики можно скачать с git репозитория, выполнив последовательность команд:

```
$ git clone http://oss.tresys.com/git/refpolicy.git
$ cd refpolicy
$ git submodule init
$ git submodule update
```

Несмотря на то, что по умолчанию с Fedora поставляется и политика SELinux, их можно поставить и отдельно, либо обновить существующие версии:

```
yum install selinux-policy selinux-policy-targeted
```

Ряд утилит, использующиеся при работе с SELinux включены в пакет coreutils. Часть команд используется и при работе с обычной версией Linux, но в них добавлен функционал для поддержки SELinux. Так, например, команды ls и ps позволяют просматривать соответственно контексты безопасности файлов в директории или процессов.

Для компиляции своих модулей, администрирования и просмотра информации о установленных политиках необходимо установить ряд пакетов, которые можно найти в репозиториях Fedora:

```
yum install policycoreutils setools libselinux checkpolicy
```

Описание наиболее часто употребляющихся утилит и пакеты, в которых их можно найти представлены в таблице

‘

Таблица 3.11 – Утилиты SELinux

Название	**Описание**	**Пакет**
apol	Предоставляет графический интерфейс для просмотра существующих правил политики.	setools
audit2allow	необходима для чтения лога и отображения правил	policycoreutils
audit2why	отображает почему доступ запрещен	policycoreutils
checkmodule	создает модуль политики из исходных кодов	checkpolicy
checkpolicy	создает базовую политики из исходных кодов	checkpolicy
getfilecon	позволяет просмотривать контекст файла по пути	libselinux
getseuser	отображает информацию о пользователях SELinux	libselinux
seinfo	Отображает такую информацию о загруженной политике, как список типов, список пользователей, список классов и т.д.	libselinux

semanage	Администрирует такие аспекты SELinux, как порты, интерфейсы, контексты файлов, логические параметры, роли и уровни для пользователей, MLS переходы и существующие типы	policycoreutils
semodule	устанавливает, удаляет и отображает список модулей в операционной системе	policycoreutils
sestatus	отображает информацию о запущенном SELinux	policycoreutils

Для проверки доступа потребуется библиотека libselinux. Она доступна для нескольких языков программирования, я использовал ее версию для СИ. Устанавливается она вместе с пакетом libselinux.

3.2. Добавление класса и его разрешений

Добавление новых классов возможно только в основную политику. Поэтому для того, что бы добавить новый класс доступа приходится перекомпилировать и перезагружать reference политику. Добавление новых классов необходимо перейти в папку со скаченными исходниками политики. Все классы описаны в файле policy/flask/security_classes. Добавим в нее следующие строчки, для описания объекта, связанного с элементом графического интерфейса - кнопкой:

```
class mybutton
```

После добавления класса необходимо так же добавить список разрешений, связанных с данным классом. Для этого необходимо отредактировать файл policy/flask/access
Этот файл содержит объявления общих расрешених и их связи с классами, а также объявления классовых разрешений. Добавим следующие строчки:

```
common gui
{
    set_enabled
    set_disabled
```

```
}
class mybutton
inherits gui
{
    click
}
```

Таким образом для класса кнопки объявлены три разрешения, позволяющие устанавливать состояние кнопки, а так же проверять разрешение на нажатие кнопки.

В процессе работы было выяснено, что reference политика не содержит класса service, в отличие от политики для Fedora, поэтому его понадобилось так же объявить в этих файлах. Без данного класса политика не работала.

Параметры компиляции можно отредактировать в файле build.xml. Я изменял два параметра, остальные параметры оставил по умолчанию:

```
#Тип политики
TYPE=mls
# Тип дистрибутива. Для Fedora необходимо выбирать redhat
DISTRO=redhat
```

Перед компиляцией необходимо произвести конфигурацию политики. Выполняется это с помощью команды:

```
make conf
```

Это команда создаст файл с описанием модулей, включаемых в политику, при желании его можно отредактировать.

Все готово к компиляции, выполняем команду:

```
make
```

Компиляция может занять некоторые время, по завершению будут созданы бинарные файлы для политики и отдельных ее модулей. Далее необходимо установить

скомпилированную политику и загрузить ее (выполнять данные действия необходимо от суперпользователя):

```
make install
make load
```

По умолчанию, скомпилированная политика будет загружаться в директорию /etc/selinux/refpolicy. При необходимости директорию refpolicy можно сменить на другую, это делается в файле build.xml перед компиляцией.

Политика установлена, однако для того, что бы при включении загружалась именно эта политика, необходимо ее прописать в конфигурационном файле /etc/selinux/config:

```
SELINUXTYPE=refpolicy
```

После перезагрузки Linux должен загрузить скомпилированную политику. Проверить это можно отобразив список всех классов текущей политики, среди него должен быть добавленный класс mybutton:

```
[vitalan69@UIR8 ~]$ seinfo -cmybutton -x
    mybutton
    set_disabled
    set_enabled
    click
[vitalan69@UIR8 ~]$
```

После этого данный класс и его разрешения можно использовать в модулях политики.

3.3. Написание модуля полити безопасности

Подгружаемый модуль политики безопасности состоит из трех файлов с расширениями .if, .te, .fc. Названия файлов должны совпадать с названием модуля. Назовем модуль для описания правил работы с классом кнопки mybutton. Файл

mybutton.te содержит объявления типов и основные правила вектора доступа и перехода типов. Файл имеет следующее содержание:

```
#Первая строка объявляет имя и версию модуля
policy_module(mybutton,1.0)

#Макрос, позволяющий использовать в тексте модуля соответствующие классы,
#разрешение, типы и роли
gen_require(‘
    class mybutton click;
    type user_t;
    type sysadm_t;
    role user_r;
    role sysadm_r;
    type chkpwd_t;
’)

# Объявления типов трех кнопок
type mybutton_red_t;
type mybutton_yellow_t;
type mybutton_green_t;


# Объявление типов домена для администратора и пользователя,
#в которых будет рабоать программа с кнопками
type mybutton_adm_domain_t;
type mybutton_user_domain_t;

# Объявление типа выполняемого файла программы
```

```
type mybutton_exec_t;

# Данный макрос указывает, что добавленные типы являются именно доменами
domain_type(mybutton_user_domain_t)
domain_type(mybutton_adm_domain_t)

# Связывание ролей пользователя и администратора
# с добавленными нами доменами
role user_r types {mybutton_user_domain_t};
role sysadm_r types {mybutton_adm_domain_t};

# Правила перехода типов домена.
type_transition user_t mybutton_exec_t:process mybutton_user_domain_t;
type_transition sysadm_t mybutton_exec_t:process mybutton_adm_domain_t;
type_transition chkpwd_t mybutton_exec_t:process mybutton_adm_domain_t;

#Разрешения доступа для пользователя к двум типам кнопок
allow mybutton_user_domain_t mybutton_yellow_t:mybutton click;
allow mybutton_user_domain_t mybutton_green_t:mybutton click;

#Разрешения доступа для администратора к двум типам кнопок
allow mybutton_adm_domain_t mybutton_red_t:mybutton  click;
allow mybutton_adm_domain_t mybutton_green_t:mybutton click;
```

Данный модуль политики указывает, что бы исполняемый файл, имеющий тип mybutton_exec_t запускался в разных доменах для пользователя и администратора. Так же в данной политике для администратора разрешен доступ к красной и зеленой кнопке, а для пользователя к желтой и зеленой кнопке.

Также в модуле политики указывается, что роль пользователя имеет право выполнять домен mybutton_user_domain_t, а роль администратора имеет право выполнять домен mybutton_adm_domain_t. Без этого переход типов будет осуществлять некорректно.

Файл mybutton.if содержит интерфейсы модуля. В рамках поставленной задачи не требуется предоставлять интерфейс к модулю. Однако в этом файле можно добавить документацию к модулю.

```
## <summary>Mybutton exemple policy</summary>
## <desc>
## <p>
## Provide different access for users and sysadm to
##      objects class mybutton.
## </p>
## </desc>

################################
```

Файл mybutton.fc содержит правила для маркирования файлов. Нам необходимо указать контекст для исполняемого файла программы, демонстрирующей возможности добавленного класса. Содержащие файла:

```
/home/vitalan69/workspace/uir/uir  -- gen_context(user_u:user_r:mybutton_ex
```

Можно приступать к компиляции модуля. Для компиляции используется Makefile, расположенный в директории /usr/share/selinux/devel/. Выполняем команду:

```
make -f /usr/share/selinux/devel/Makefile
```

По завершению компиляции должен появиться бинарный файл модуля политики с расширением .pp

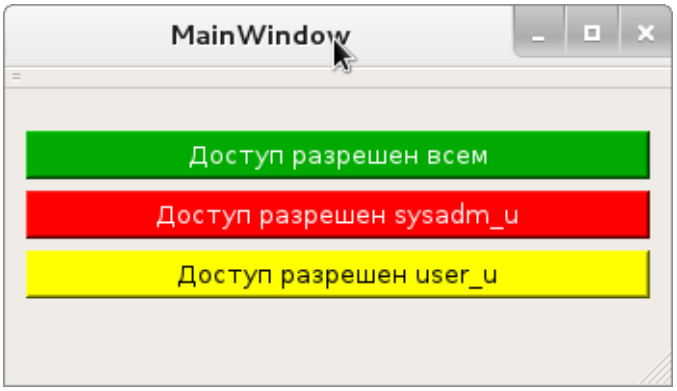
Установка модуля политики производится с помощью утилиты semodule:

```
semodule -i mybutton.pp
```

Проверить, загружен ли правила из модуля в систему можно с помощью утилиты apol, которая позволяет отображать список всех правил, а так же производить поиск по ним.

3.4. Программная проверка доступа с помощью библиотеки libselinux

Для демонстрации результатов написанной политики была написана программа с пользовательским интерфейсом. Для разработки пользвотельского интерфейса использовался QT фрэймворк. Программа имеет следующий вид:



Интерфейс программы
Рисунок 3.1 – Интерфейс программы

Если доступ разрешен пользователю показывается сообщение с текстом Access allowed, иначе показывается сообщение Access deny.

Программа используется библиотеку libselinux для проверки доступа, поэтому ее надо подключить в профайле проекта:

```
LIBS += -L/usr/lib/ -lselinux
```

Перед проверкой доступа надо получить контекст текущего процесса, а так же активировать работу с вектором доступа:

```
void MainWindow::initSelinux()
{
    getcon(&program_context);
```

```
    qDebug()<<"Program context = " << program_context;
    selinux_opt *opt = new selinux_opt();
    opt->type = 1;
    opt->value = "avc";
    avc_open(opt, AVC_OPT_SETENFORCE);
}
```

В программе для каждой из кнопок заданы контексты с соответствующим типом. При клике на кнопку проверяется доступ с помощью следующей функции:

```
int MainWindow::checkAccess(security_context_t buttoncon){
    security_id_t myid = NULL;
    struct av_decision avd = {0, 0, 0, 0, 0,0};
    security_id_t buttonid = NULL;
    avc_context_to_sid(program_context, &myid);
    avc_context_to_sid(buttoncon, &buttonid);
    access_vector_t av=0;
    int length=security_compute_av(program_context,buttoncon,SECCLASS_MYBU
    int result = avc_has_perm(myid, buttonid, SECCLASS_MYBUTTON, MYBUTTON_
    QMessageBox msgBox;
    if (result!=0&&(errno==EACCES)){
        qDebug()<<"Errno="<<errno;
        msgBox.setText(tr("Access deny"));
    } else if (result==0){
        msgBox.setText(tr("Access allowed"));
    } else {
        msgBox.setText(tr("Error "));
        qDebug()<<"errno="<<errno;
    }
    msgBox.exec();
```

```
    qDebug()<<"Length="<<length;
    qDebug()<<avd.allowed<<avd.auditallow<<avd.auditdeny<<avd.decided<<avd
    return result;
}
```

Наиболее интересна в данной функции строчка:

```
int result = avc_has_perm(myid, buttonid,
           SECCLASS_MYBUTTON, MYBUTTON__CLICK ,NULL, &avd);
```

В ней проверяется доступ процесса с SID myid к объекту с SIC buttonid, указывается что объект buttonid принадлежит к классу SECCLASS_MYBUTTON, а разрешение, которое запрашивается для него - MYBUTTON__CLICK. В результате, если возвращается 0, то доступ разрешен. Если возвращается -1, то это означает, что либо доступ запрещен, либо произошла другая ошибка. Если доступ запрещен, то то значение errno устанавливается в значение EACCES.

Для демонстрации работы программы необходимы два пользователя, первый работает с ролью user_r, второй с ролью sysadm_r. Пользователю user_r разрешен доступ к желтой и зеленой кнопкам, а пользователю sysadm_r разрешен доступ к красной и зеленой кнопкам.

ЗАКЛЮЧЕНИЕ

В процессе выполнения задания были рассмотрены ключевые особенности мандатного разграничения доступа, использующегося в SELinux. Был изучен и применен на практике язык написания политик для модуля безопасности SELinux. Результатом работы можно считать описание алгоритма добавления новых классов доступа в политику SELinux, а так же проверки доступа с помощью библиотеку libselinux.

Результат работы можно использовать для написания приложений с пользовательским интерфейсом, в котором необходимо разграничить доступ пользователя к различным элементам управления в зависимости от уровня доступа пользователя.

Кроме того, был получен опыт написания и редактирования существующих политик SELinux, который можно использовать при настройки и администрирование компьютеров с ОС Linux, на которых включен модуль безопасности SELinux.

ОТЗЫВ РУКОВОДИТЕЛЯ

« ___ » _____ 2012 г.

Доцент, кандидат технических наук — С.К. Муравьев

ЗАДАНИЕ
НА УЧЕБНО-ИССЛЕДОВАТЕЛЬСКУЮ РАБОТУ

В рамках выполнения работы необходимо:

1) Изучить основы работы с политиками SELinux
2) Добавить новый класс доступа в политику SELinux
3) Написать модуль политики безопасности для добавленного класса
4) Реализовать проверку доступа, используя библиотеку libselinux для добавленного класса

« ___ » _____ 2012 г.

Студент группы К8-361
_____ В.Б. Холявин

Доцент, кандидат технических наук
_____ С.К. Муравьев