Московский Государственный Университет

Факультет Вычислительной Математики и Кибернетики
Лаборатория Вычислительных Комплексов

Курсовая Работа

Автоматизация разметки исполнимого кода программы контрольными точками для точного разделения пространства ее состояний со стороны ядра ОС.

Федор Сахаров
группа 322
научные руководители: Денис Гамаюнов, Стас Беззубцев

Москва, 2009

Аннотация

В работе рассматривается возможность расширения функциональности механизма контроля поведения программ, используемого в SELinux, при помощи повышения гранулярности контроля поведения приложений в указанной системе за счет отслеживания внутреннего состояния программы из ядра. Предлагается делать это при помощи разметки исполнимого кода приложения контрольными точками на уровне исходных текстов. В работе приводится сравнительный анализ систем безопасности уровня ядра, описание разработки инструментальной системы для проставления контрольных точек в программах и использование существующих средств для добавления состояний контролируемых приложений.

Содержание

1 Введение.

Стандартные системы безопасности ОС, основы которых были заложены несколько десятилетий назад, давно не являются удовлетворительными. Стандартная система безопасности Unix предоставляет одинаковые права всем пользователям в определенной группе, все процессы, запущенные от имени конкретного пользователя, обладают его привилегиями. Любая уязвимость становится потенциальной причиной компрометации учетной записи пользователя. В 1985 году был введен стандарт «Критерии оценки доверенных компьютерных систем» более известный под названием «Оранжевая книга». Данный стандарт получил международное признание и оказал сильное влияние на последующие разработки в области информационной безопасности. Появилось семейство так называемых «trusted» операционных систем — TrustedBSD, Trusted Solaris, Trusted UNICOS 8.0, HP-UX 10.26, PitBull for AIX 5L, XTS-400. На сегодняшний день результатами разработки более современных и продвинутых систем безопасности, работающие поверх стандартных, стали такие продукты, как SELinux (NSA, Red Hat), AppArmor (Immulinux, Novell), PaX(GRSecurity), Seatbelt(Apple). Кроме этого, разработчики некоторых систем пытаются расширить стандартные системы безопасности, улучшая их и добавляя новые способы защиты, как это происходит с Windows (Vista). В данной работе будут рассмотрены некоторые из перечисленных систем.

2 Постановка задачи

2.1 Расшифровка темы

В работе рассматриваются формализованный механизм контроля поведения программ, используемый в ОС Linux - SELinux. В задачу входит анализ теоретической базы этих механизмов, практический инструментарий в распространённых дистрибутивах (Debian, Ubuntu для AppArmor?), достоинства и недостатки, в том числе по научной литературе. Основная цель данной работы — повысить гранулярность контроля поведения приложений в указанных системах за счёт отслеживания внутреннего состояния программы из ядра. В рамках работы предполагается делать это специальной разметкой программы на уровне исходных текстов, а также бинарных патчей (хотя бы на уровне исследования). Конечная цель — автоматизация такой разметки по тестам.

2.2 Актуальность

Существующие механизмы защиты ядра и контроля поведения приложений в ОС Linux (SELinux, AppArmor) имеют ряд недостатков, в частности, используемые в них методы выявления аномального поведения приложений не учитывают внутреннее состояние защищаемого приложения. В ряде случаев это накладывает сильные ограничения на допустимое поведение, что ограничивает применимость этих механизмов. Учёт внутреннего состояния контролируемого приложения позволит избежать жёстких обобщённых ограничений на его поведение.

2.3 Цель работы

Расширение функциональности систем защиты уровня ядра Linux (SELinux) за счёт повышения гранулярности отслеживания поведения приложений и разделения их внутренних состояний.

2.4 Постановка задачи

В рамках работы должны быть решены следующие задачи:

- Сравнительный анализ систем безопасности уровня ядра ОС Linux.
- Исследование возможности автоматической простановки контрольных точек на уровне исходных текстов и бинарных патчей программ.
- Разработка инструментальной системы для автоматизации проставления контрольных точек в программах по тестам.
- Расширение профилей SELinux, добавление состояний контролируемых приложений.

3 Существующие системы безопасности уровня ядра ОС

Рассмотрим самые значимые и распространенные системы из перечисленных выше.

Рассмотренные системы безопасности уровня ОС Было рассмотрено 7 систем безопасности уровня ОС. В таблице 1.1 приведена краткая информация по каждой из них.

Название системы	Производитель	Ссылки
AppArmor	Novell	http://www.novell.com/linux/security/apparmor/
Selinux	Red Hat	http://www.nsa.gov/selinux/
PaX	GRSecurity	http://pax.grsecurity.net/
Trusted BSD	Trusted BSD	http://www.trustedbsd.org/

Таблица 1.1

3.1 SELinux

SELinux является системой безопасности уровня ядра Linux, основанной на фреймворке LSM. LSM - интерфейс, позволяющий создвать модули безопасности, реализующие некоторую логику принятия решений относительно разрешения или запрещения тех или иных взаимодействий между объектами и субъектами системы. SElinux обеспечивает возможность комплексной защиты системы, ограничивая поведение приложений и пользователей в рамках политик безопасности. В первую очередь SELinux направлена на борьбу с успешными атаками, в частности, с «атаками нулевого дня», когда уязвимость уже известна злоумышленнику, но лекарства еще не было выпущено. В таких случаях уязвимость устраняется на уровне политики. Компания Tresys ведет подсчет конкретных случаев угроз безопасности, которые, в частности, могли быть предотвращены SELinux. В их числе: переполнение буфера в Samba (may 2007), Apache DoS (jun 2007) Mambo exploit (jul 2007), hplip Security flaw (oct 2007). Конфигурация политик является весьма сложной задачей, учитывая наличие специального языка описания политик, сложного в написании правил. Кроме этого, добавление новых профилей может повлечь за собой необходимость в модификации уже имеющихся профилей, отрицательно сказываясь на удобстве и простоте описания политик.

Основные понятия.

Рассмотрим основные понятия SELinux.

Принудительное присвоение типов (TE).

Основной идеей принудительного присвоеня типов является явная разметка всех объектов в системе специальными структурами данных (метками безопасности), хранящими в себе информацию об атрибутах объекта, используемую при принятии решений внутри логики политики. Для процессов и объектов используется один и тот же тип атрибутов. Поэтому достаточно одной матрицы для описания взаимодействий между разными типами, при этом объекты одного типа могут рассматриваться поразному, если их их ассоциированные классы безопасности различны. Пользователи не привязаны к типам безопасности напрямую, вместо этого используется RBAC.

Ролевой контроль доступа (RBAC)

Данный метод используется для определения множества ролей, которые могут быть назначены пользователям. SELinux расширяет модель RBAC до жесткой привязки пользовательских ролей к определенным доменам безопасности, роли могут быть организованы в виде иерархии приоритетов. Такая привязка ролей к доменам позволяет принимать большинство решений на основе конфигурации TE. Контекст безопасности, кроме всего прочего, включает в себя атрибут роли.

Многоуровневая система безопасности (MLS)

SELinux предоставляет MLS для случаев, когда есть необходимость в традиционной многоуровневой системе безопасности. У объектов и субъектов могут быть различные уровни и категории. Как правило, используется лишь один уровень.

Принципы работы

Главными элементами системы безопасности являются субъект, объект и действия. В классы объектов входят классы файлов (blk_file, chr_ file, dir, fd,...) , классы межпроцессного взаимодействия (ipc,msg,msgq,sem,shm), классы сетевого взаимодействия (key_socket,netif,node, packet_socket,tcp_socket), классы объектов (passwd), системные классы (capability, process, Secutity, System). Под субъектами понимаются процессы, демоны, ядро и.т.д.. Действия, которые субъекты SELinux могут производить над объектами меняются от класса к классу. Для классов файлов это, например, будут создание, исполнение, ссылки, чтение, запись, удаление. SELinux ассоциирует атрибуты безопасности с субъектами и объектами и основывает свои решения на этих атрибутах. Атрибутами являются: идентификатор пользователя, роль и тип. Идентификатор пользователя — пользовательская учетная запись, ассоциированная с субъектом или объектом. У каждого пользователя может быть несколько ролей, но в какой-то конкретный момент времени ему может быть предписана только одна из них. Пользователь может менять роли командой newrole. Типы (для проеццов —

Домены) делят субъекты и объекты на родственные группы. Это — главный атрибут безопасности, используемый SELinux для принятия решений. Типы позволяют помещать процессы в «песочницы» и предотвращать повышение привилегий. К примеру, роль суперпользователя - sysadm_r, его тип — sysadm_t. Политика безопасности SELinux загружается системой из бинарного файла политики, который, как правило, находится в /etc/selinux. Бинарная политика собирается при помощи make, исходные коды, как правило, находятся в /etc/selinux/$(POLNAME)/src/policy. Инструменты работы с SELinux могут быть разделены на три категории: специальные утилиты для настройки и использования SELinux, модифицированные версии стандартных команд и программ Linux, некоторые добавочные инструменты, к примеру, для настройки и анализа политик. Среди основных команд можно выделить следующие: chcon – помечает файл или группу файлов указанным контекстом безопасности, checkpolicy – позволяет выполнять множество действий, связанных с политиками, в том числе, компиляцию политики и ее загрузку в ядро; getenforce — позволяет узнать в каком режиме работает SELinux, newrole – позволяет пользователю перемещаться между ролями; run_init — позволяет запускать, останавливать или контролировать сервис; setenforce позволяет менять режим работы системы; setfiles присваивает метки указанной директории и ее поддиректориям. Некоторые из измененных программ: cron, login, logrotate, pam, ssh. Некоторые инструменты: Apol – инструмент для анализа файла policy.conf; SeAudit – инструмент для анализа логов, имеющий графический интерфейс; SeCmds; SePCuT — инструмент для просмотра и редактирования файлов политик; SeUser — модификация пользовательских учетных записей.

Методы контроля за внутренним состоянием приложения

SELinux предоставляет разработчикам приложений инструментарий, позволяющий создавать более безопасные приложения путем изменения текущих привилегий приложения во время его исполнения. Это реализуется путем изменения домена приложения. Приложение должно запросить у ядра смену своего текущего домена на указанный. При этом возможность такой смены доменов должна быть явно описана в политике безопасности. Далее данный метод будет рассмотрен более подробно.

3.2 AppArmor

Во многом схож с SELinux, использует LSM. В отличие от SELinux, AppArmor не использует явную разметку всех объектов в системе. AppArmor контролирует поведение приложений, опираясь на политики, которые являются текстовыми файлами,

удобными к восприятию и редактированию. В данных файлах хранится информация, основанная на путях к объектам в файловой системе, о том, к каким объектам и с какими правами имеет доступ приложение. В отличие от SELinux, в которой настройки глобальны для всей системы, профили AppArmor разрабатываются индивидуально для каждого приложения. Таким образом, гораздо меньше вероятность необходимости изменения существующих профилей при генерации новых профилей. Кроме этого, AppArmor предоставляет инструменты автоматической генерации профилей на основе поведения приложения и возможность производить контроль в двух режимах: режиме обучения и режиме принуждения. AppArmor является системой безопасности, поддерживаемой компанией Novell, включена в дистрибутивы openSUSE и SUSE Enterprise. Изначально в AppArmor включен набор стандартных профилей, запускаемых после установки. Отдельно доступны профили для разных популярных программ и серверов. Кроме этого, существуют инструменты для генерации профилей (genprof и logprof). Основная идея — верный выбор приложений, нуждающихся в ограничении привилегий и создание/редактирование профилей безопасности. Таким образом, в случае эксплуатации злоумышленником уязвимости, нанесенный ущерб сводится к минимуму. Система может работать в двух режимах: режиме обучения (complain) и в принудительном режиме (enforce). В первом из них все нарушения правил профиля разрешены, но немедленно регистрируются. Загрузка профиля в принудительном режиме предписывает системе отправлять сообщения о нарушениях в syslogd. Запуск и остановку AppArmor можно осуществлять при помощи команды rcapparmor с одним из следующих параметров: start (загрузка модуля ядра, анализ профиля, монтирование своей фс); stop (фс размонтируется, профили становятся недействительными); reload (перезагрузка профилей), status (информация о количестве запущенных профилей, в каком режиме они работают). Инструменты командной строки AppArmor: autodep (создает приблизительный профиль для программы или рассматриваемого приложения); complain (устанавливает профиль AppArmor в обучающий режим); enforce (переводит профиль в принудительный режим); genprof (генерирует профиль, программа указывается при запуске); logprof (управляет профилями AppArmor); unconfined (выводит список процессов с портами tcp и udp, которые не имеют загруженных профилей AppArmor). Система AppArmor построена на системе полных путей к файлам, проще говоря, типичное описание профиля выглядит примерно так:

```
#include <tunables/global>
/usr/bin/man {
    #include <abstractions/base>
```

```
    #include <abstractions/nameservice>
    capability setgid,
    capability setuid,
    /usr/lib/man-db/man Px,}
}
```

Профиль состоит из файлов, каталогов с указанием полных путей к ним и прав доступа к этим объектам. При этом r — разрешение на чтение, w — запись(за исключением создания и удаления файлов), ix — исполнение и наследование текущего профиля, px — исполнение под специфическим профилем, Px — защищенное выполнение, ux — неограниченное исполнение, Ux — защищенное неограниченное исполнение, m — присвоение участку памяти атрибута «исполняемый», I — жесткая ссылка. Чтобы подключить готовый профиль к AppArmor, достаточно его скопировать в каталог /etc/apparmor.d.

Методы контроля за внутренним состоянием приложения

Система предоставляет возможность смены текущих привилегий для веб-сервера Apache (Change Hat). Тем не менее, из-за фактического прекращения разработки AppArmor данная система так и не стала доступной для использования с произвольным приложением.

3.3 PaX

PaX - набор патчей ядра от GRsecurity. Существует три класса угроз, предотвращением которых занимается PaX. Это внедрение и исполнение кода с повышенными привилегиями, исполнение кода самого процесса путем изменения нормального течения исполнения процесса, нормальное исполнение программы, но над данными, для которых предусмотрены повышенные привилегии. Non-executable pages (NOEXEC) и mmap/mprotect (MPROTECT) предотвращают атаки первого класса. За одним исключением: если злоумышленник имеет право на создание/запись в файл на атакуемой машине и mmap() его в атакуемый процесс, у него появляется возможность внедрения кода. Address Layout Randomisation (ASLR) позволяет предотвратить все три класса атак в той ситуации, когда атакующий заранее закладывается на адреса в атакуемом процессе и не может узнать о них в процессе исполнения. Так как PaX полностью внедрен в ядро, предполагается то, что ядро является Trusted Computer Base. Инструментарий позволяет предотвратить исполнение стека, обеспечить рандомизацию размещения адресов внутри адресного пространства (address space layout randomization) .

Основная цель данного проекта — изучение различных защитных механизмов, защищающих от эксплойтов уязвимостей ПО, которые предоставляют злоумышленнику полные права на чтение/запись в системе. Исполнение кода связано с необходимостью изменять ход выполнения процесса используя уже существующий код. Одна из основных проблем — подмена адресов возврата из функций и подмена самих адресов функций. Для установки PaX требуется наложить патч на дерево исходных кодов ядра, после чего собрать ядро и установить в систему.

3.4 Trusted BSD

Проект TrustedBSD – проект разработки расширения существующей системы безопасности FreeBSD, который включает в себя расширенные атрибуты UFS2, списки контроля доступа, OpenPAM, аудит событий безопасности с OpenBSM, мандатное управление доступом и TrustedBSD MAC Framework. Trusted BSD была задумана как система, удовлетворяющая стандартам «оранжевой книги». Расширенные атрибуты UFS2 позволяют ядру и пользовательским процессам помечать файлы именованными метками. В этих метках хранятся данные, необходимые системе безопасности. ACL и метки MAC в их числе. Списки контроля доступа — расширения дискреционного контроля доступа. Аудит системных событий позволяет вести избирательный логгинг важных системных событий для последующего анализа, обнаружения вторжений, и мониторинга . Начиная с версии 5.0 в ядре FreeBSD появилась поддержка MAC Framework, прошедшая испытания в TrustedBSD. Данный фреймворк позволяет создавать политики, определяющие принудительное присвоение доменов и типов (DTE), многоуровневую систему безопасности (MLS). Данный фреймворк предоставляет интерфейсы управления фреймворком, примитивы для синхронизации, механизм регистрации политик, примитивы для разметки объектов системы, разные политики, реализованные в виде модулей политики MAC и набор системных вызовов для приложений. При регистрации политики, происходит регистрация специальной структуры (struct mac_policy_ops), содержащей функции MAC framework, реализуемые политикой. На данный момент существуют следующие политики:

mac_biba – Реализация политики Biba, во многом схожей с MLS. Позволяет присваивать объектам и субъектам системы атрибуты доступа, которые образую иерархию уровней. Все операции над информацией в системе контролируются исходя из уровней взаимодействующих сущностей.

mac_ifoff позволяет администраторам контролировать сетевой трафик.

mac_lomac (Low-watermark MAC) еще одна реализация многоуровневого контроля

доступа.

mac_bsdextended (file system firewall) Система защиты файлов, основанная на определении прав доступа на основании роли пользователя.

mac_mls — реализация политики MLS. Объекты классифицируются некоторым образом, субъектам присваивают уровень доступа.

Предоставляемые системой методы контроля за внутренним состоянием приложения

Фреймворк MAC позволяет реализовать в модуле безопасности возможность изменения приложением собственных прав.

3.5 Недостатки существующих методов контроля за внутренним состоянием приложения

Основной идеей существующих методов является предоставление приложениям определенных интерфейсов для изменения собственного состояния. Это является очередным шагом в сторону более безопасного программирования и альтернативой рекомендуемому подходу, который заключается в разбиении приложения на несколько более мелких приложений, которые соответствовали бы внутренним состояниям исходного приложения. При этом, изменения прав могут осуществляться традиционно на основании наследования доменов во время execx(). Основным недостатком данного подхода является тот факт, что внутри ядра невозможно отличить вызовы функций смены доменов, внедренных в код приложения разработчиком, от вызова аналогичных функций злоумышленником в результате изменения нормального хода исполнения программы. Кроме этого, данный подход предполагает, что разработчики сами должны определять необходимые места для внедрения таких вызовов, что не всегда возможно в силу ряда причин. В этом случае, изменения кода должны производиться третими лицами, как следствие появляются патчи, зависимые от версии, плюс необходимость пересобирать приложение с выходом каждой новой версии.

4 Решение задачи разметки исполнимого кода программы контроль ными точками и контроля за ее состояниями с использованием системы без опасности уровня ядра SELinux

4.1 Постановка задачи

Необходимо реализовать механизм, позволяющий размечать код приложения контрольными точками, для последующего контроля за внутренним состоянием приложения из ядра ОС. При этом необходимо избавиться от необходимости передавать информацию из пользовательских приложений в ядро, так как такой информации, в общем случае, нельзя доверять. Кроме этого, необходимо реализовать механизм, интегрированный с модулем SELinux, который бы определял изменения внутренних состояний в контролируемых приложениях и менял их домены согласно этим изменениям. При этом модулю должна быть доступна информация о связи между сменами доменов, внутренними состояниями приложений и контрольными точками.

4.2 Реализация механизма наблюдения за состояниями процесса.

Итак, в данной работе под контрольной точкой подразумевается некоторый адрес в виртуальном адресном пространстве процесса. Попадание исполнения на один из таких адресов, в общем случае, означает изменение состояния процесса. В первую очередь, возникает необходимость некоторым образом разметить код приложения контрольными точками, а точнее, получить адреса в виртуальном адресном пространстве приложения. В данной работе будет рассматриваться только разметка кода приложений, написанных на C/C++ на основании их исходных текстов. Это возможно сделать при помощи получения адресов меток, которые можно проставлять тех местах кода, где предполагается изменение состояния приложения.

Пример:

```
#include <stdio.h>
int main (int argn, char *argv[])
{
    static void * ret[2] __attribute__((section(".mylabels"),used)) =
        {&& ret1,&& ret2};
    if (argn > 2) {
```

```
    ret1:
        printf("1 n");
    { else {
    ret2:
        printf("2 n");
    {
    return 0;
{
}
```

В данном примере есть две метки. Можно сказать, что здесь они определя ют две различные ветви исполнения программы. Средства компилятора gcc позоляют управлять размещением данных в бинарном файле программы при п омощи команды __attribute__. При помощи этой команды в исполнимом файле возможно создать отдельную секцию, содержащую эти адреса. Исполн имые файлы с данной секией и без нее будут отличаться только наличием этой секции, при этом в файле с данной секцией все адреса останутся те ми же, что и в файле без секции. Таким образом мы получаем очень удобн ый способ хранения адресов прямо в бинарном файле программы, откуда их можно извлекать для дальнейшей обработки, либо читать эту информцию п рямо перед запуском приложения.

Проблема наблюдения за данными адресами может быть реализована по-разн ому. Можно использовать вызов ptrace и создавать сложную систему метод ов для наблюдения за событиями в наблюдаемом приложении. При этом обяз ательно нужно следить за такими событиями, как fork и exec для определ ения, в какое состояние переходит приложения. Так же наблюдение за exe с обеспечит определение факта запуска определенного приложения. Такой контроль предлагается осуществлять при помощи системы utrace и ее клие нта — uprobes. Utrace является патчем ядра от Red Hat, позволяющим с троить отладочные движки, работающие в пространстве ядра в качестве за гружаемых модулей. Uprobes является клиентом utrace и позволяет устана вливать точки останова на определенные адреса в коде и для каждой из н их регистрировать функции-обработчики, которые будут срабатывать ккажд ый раз, как управление в приложении попадет на одну из точек останова.

4.3 Описание отладочной системы Uprobes-utrace

Utrace.

Обычным интерфейсом отладки программ под Linux является системный вызо в ptrace(). Как правило, он используется отладчиками. Данный интерфейс очень тяжело использовать, так как приходится создавать целую систему методов для разметки точками, наблюдения за событиями и прочих манипу ляций над отлаживаемым приложением.

С недавних пор появилась гораздо более удобная альтернатива использова нию ptrace для отладки пользовательских приложений из пространства ядр а в виде utrace. Основной код данной системы не имеет интерфейсов в по льзовательском пространстве. Вместо этого есть интерфейсы в ядре, кото рые позволяют создавать отладочные механизмы, работающие в пространств е ядра. Эти интерфейсы основаны на концепции “отладочного движка”, к оторый представляет собой обычную структуру, содержащую указатели на ф ункции. У данной стркутуры есть 14 функций-коллбеков, которые будут вы званы в случае определенных событий в отлаживаемом приложении.

Пример:

```
u32 (*report_syscall_entry)(struct utrace_attached_ engine *engine,
                struct task_struct *tsk,
                struct pt_regs *regs);
```

Как только отлаживаемый процесс совершит системный вызов, будет вызван а соответствующая функция отлаживаемого движка - report_sysc all_entry() (разумеется, если она была зарегистрирована). Вызов данн ого обработчика происходит до выполнения системного вызова, отладчик м ожет безопасно полчать доступ к остановленному отлаживаемому процессу. Функция-обработчик возвращает битовую маску, которая определяет, что должно произойти далее — можно изменять состояние отладки, прекращат ь отладку, скрывать событие от других отладочных движков и многое друг ое.

Отладочный движок регистрируется следующей функцией:

```
struct utrace_attached_engine *
    utrace_attach(struct task_struct *target, int flags,
              const struct utrace_engine_ops *ops,
          unsigned long data);
```

Данный вызов прицепит отладочный движок к указанному процессу. Возможн а регистрация более чем одного отладочного движка для одного и того же процесса —

серьезное отличие от ptrace(). Только что зарегистрирова нный движок ничего не делает и находится в состоянии idle. Для запуска необходимо указать соотвествующие флаги в вызове функции

```
int utrace_set_flags(struct task_struct *target,
             struct utrace_attached_engine *engine,
             unsigned long flags);
```

Существует специальный флаг - UTRACE_EVENT(QUIESCE), которы й может переключать процесс в состояние ожидания. В общем случае, все операции с процессом в первую очередь требуют установки этого флага, п осле чего можно ожидать исполнения коллбека report_quiesce() , который извещает об остановке процесса. Естьмножество других событи й, извещения о которых могут быть получены отладочным движком. В их чи сле fork(), exec(), получение сигнала, завершение процесса, вызов сист емного вызова и др..

Uprobes.

Uprobes является клиентом системы utrace и входит в состав утилит для наблюдения за событиями в системе Systemtap в качестве модуля ядра. Кр оме этого, существуют патчи, позволяющие собрать uprobes непосредствен но в ядро Linux. Основной функцией данного набора функций является обе спечение возможности проставления контрольных точек в код отлаживаемог о процесса и регистрация функций, обрабатывающих события, связанные с данными точками. Есть два типа таких контрольных точек: uprobes и uret probes. Uprobe может быть установлена на любой адрес в виртуальном адр есном пространстве процесса и сработает при попадании исполнения на ин струкцию, расположенную по этому адресу. Uretprobe сработает при завер шении работы указанной функции в отлаживаемом процессе. При регистраци и точки останова, uprobes сохраняет копию инструкции, расположенной по этому адресу в приложении, останавливает его исполнение, подменяет пе рвые байты по этому адресу на инструкцию точки останова (int3 на i386 x86_64) и вновь запускает исполняемое приложение. Когда исполнение по падает на эту инструкцию, срабатывает ловушка и генерируестя сигнал SI GTRAP. Uprobes получает этот сигнал и находит связанную с ним точку ос танова и ее функцию-обработчик. Отлаживаемый процесс будет остановлен до завершения работы функции-обработчика. После завершения работы функ ции-обработчика uprobes исполняет сохраненную команду, которая первона чально располагалась по адресу точки останова в пользовательском проце ссе и вновь запускает пользовательский процесс.

Регистрация контрольной точки может быть произведена с помошью функции

```
#include <linux/uprobes.h>
int register_uprobe(struct uprobe *u);
```

Будет установлена точка установа в виртуальном адресном пространстве п роцесса u->pid по адресу u->vaddr и с обработчиком v->handler, который может быть определен следующим образом:

```
#include <linux/uprobes.h>
#include <linux/ptrace.h>
void handler(struct uprobe *u, struct pt_regs *regs);
```

При завершении отлаживаемого процесса, либо при вызове функции exec() uprobes автоматически удаляет все контрольные точки и их обработчики. При выполнении вызова fork() во вновь созданном процессе удаляютя все контрольные точки.

SELinux Итак, как уже было сказано, в SELinux три аттрибута безопасности: иден тификатор пользователя, роль и тип вместе образуют так называемый конт екст безопасности. SELinux хранит контексты безопасности в своих табли цах, каждая запись в которых определяется идентификатором безопасности , (SID), который представляет собой целочисленную переменную. Разным к онтекстам ставятся в соответствие разные идентификаторы безопасности. При этом Security Server принимает все решения, описаные в логике поли тики на основании двух идентификаторов взаимодействующих объектов.

4.4 Архитектура SELinux

SELinux состоит из следующих основных компонент:

- Код в ядре (Security Server, hooks,selinuxfs)
- Библиотека для взаимодействия с ядром
- Политика безопасности
- Различные инструменты
- Размеченные файловые системы

Код в ядре

Задачей SELinux в ядре является наблюдение за событиями в системе и пр инятие решений о разрешении о различных операциях в соответствии с пол итикой безопасности. Кроме этого, Security Server ведет логи для опред еленных разрешенных или запрещенных операций, список которых описан в политике. Кроме этого Security Server заполняет соответствующие структ уры безопасности в запускаемых приложениях. Такой структурой является следующая структура:

```
struct task_security_struct {
    u32 osid;
    u32 sid;
    u32 exec_sid;
    u32 create_sid;
    u32 keycreate_sid;
    u32 sockcreate_sid;
};
```

На нее указывает поле security в структуре task_struct. Поля структур ы безопасности включают в себя следующую информацию.

Поле	Описание
osid	Старый идентификатор, который был у процесса, до выполнения execve.
sid	Текущий идентификатор
exec-sid	Идентификатор, использующийся для определения прав на выполнение exec.
create_sid	Идентификатор, которым будут помечены объекты ФС, созаваемые данным пр оцессом
keycreate_sid	Идентификатор, который будет присвоен ?
sockcreate_sid	Идентификатор для сокетов данного процесса.

В нашем случае интересно поле sid. Именно на основании значения данног о поля в Security Server принимаются решения согласно логике политики SELinux.

Библиотка работы с интерфейсами SELinux Данная библиотека (libselinux.so) используется большинством из компоне нт SELinux, находящихся в пользовательском пространстве.

Политика безопасности SELinux Сервер безопасности принимает все свои решения на основании политики бе зопасности, описанной администратором системы.

При запуске системы SEL inux загружает политику безопасности из бинарного файла, который, как правило находится в /etc/security/selinux.

Инструменты В первую очередь SELinux предоставляет набор инструметнов для администт рирования системы, компиляции политики в бинарное представление, добав ления новых ролей, изменения меток файлов. Корме этого некоторые систе мные команды и программы заменяются модифицированными аналогами, среди них cp, mv, install, id, ls, ps, login, logrotate, pam, ssh и прочие. Существуют различные инструменты, призванные упростить работу с SELin ux, в том числе инструменты с графическим интерфейсом такие как Apol, SeAudit, SeCmds, SePCuT, SeUser.

4.5 Существующий в SELinux метод динамического переключения контекста

Рассмотрим более подробно инструментарий динамического изменения контекста приложения в SELinux, о котором упоминалось ранее. Данная система предполагает, что приложение должно быть тесно интегрировано с существующей политикой и в зависимости от своего текущего состояния сообщать SELinux о смене контекста. Такой подход позволил бы создавать более безопасные приложения, при разработке которых возможно было бы выделять состония, в которых приложению нужны различные минимальные права. Такими интерфейсами стали функции

```
#include <selinux/selinux.h>

    int getcon(security_context_t *context);
    int getprevcon(security_context_t *context);
    int getpidcon(pid_t pid, security_context_t *context);
    int getpeercon(int fd, security_context_t *context);
    int setcon(security_context_t context);
```

Основной целью данной системы является предоставление возможности доверенному приложению изменять свои права непосредственно в процессе выполнения, отказываясь от определенных прав, когда они не нужны и запрашивая некоторые права, когда в них есть необходимость.

Эта система появилась несмотря на то, что основной идеологией безопасного программирования с участием SELinux является разбиение приложения на некоторое количество меньших приложений, за поведением которых гораздо легче наблюдать,

при этом у каждого из них могут быть различные права. Причиной создания системы стал тот факт, что многие приложения по тем или иным причинам не могут быть спроектированы таким образом.

Данная система реализована следующим образом. Данные функции пишут контекст безопасности, который является строкой символов, в /proc/PID/attr/current. Для того, чтобы приложение могло использовать указанные функции в политике для него должно быть описано соответствующее разрешение, которое выглядит следующим образом:

```
allow XXX_t self:process setcurrent
```

Но данное предложение, по сути, всего лишь разрешает приложению использовать интерфейсы динамического изменения типа, никак не определяя, в какой контекст может перейти приложение. За это отвечает предложение следующего вида:

```
allow XXX_t YYY_t:process dyntransition
```

Важно отметить, что логика работы Security Server в данном случае такова, что решения относительно возможности приложения динамически менять свой домен на указанный принимаются независимо от смены домена при вызове exec*.

Рассмотрим, что происходит в ядре при динамической смене контекста приложения. Как уже было сказано, SELinux использует набор интерфейсов LSM. При записи в указанный выше интерфейс /proc/PID/attr/current, цепляется функция security_setprocattr(), которая производит определенные проверки на основании политики, и в том случае, если в политике описана возможность такого изменения контекста, производится все необходимые действия. Важной особенностью является тот факт, что такие изменения невозможны для многопоточных приложений. Это является весьма логичным ограничением, так как множество нитей одного процесса используют одно и то же пространство памяти и гарантировать реальное разделение данных невозможно.

Рассмотрим более детально механизм смены контекста приложения. SELinux реализует функцию LSM security_setprocattr() методом selinux_setprocattr().

```
static int selinux_setprocattr(struct task_struct *p,
    char *name, void *value, size_t size)
```

Аргументами этой функции является процесс, смену контекста которого нужно произвести, опция того, что в контексте нужно менять (поля в стркутуре task_security_struct

), сам контекст в своем строковом представлении, и длина строки представления контекста. В первую очередь проверяется, что тот процесс, в котором происходят изменения — текущий процесс. Далее проверяется возможность процесса менять указанное поле в своей структуре безопасности. После этого функция ставит целочисленный идентификатор безопасности в соответствие строковому представлению контекста. После этого проверяется возможность смены контекста на указанный и в случае, если это возможно, текущий идентификатор безопасности в структуре процесса меняется на полученный из строкового представления процесса.

Недостатки данного подхода.

Основным недостатком данного подхода является необходимость внедрять вызовы интерфейсов динамического изменения контекста непосредственно в приложение. Это влечет за собой сразу несколько серьезных проблем. Во-первых, маловероятно, что разработчики будут делать это самостоятельно, тем более, что у них получится корректно выделить те участки кода, на которых приложению нужны различные привилегии, и корректно определить необходимые контексты. В таком случае, для обеспечения возможности использования этого метода, приложение должны изменять третьи разработчики, следовательно, такие приложения будут отличны от основной ветки и патчи вместе с пересборкой придется осуществлять при выходе каждого очередного релиза приложения. Но более серьезной проблемой является то, что информация передается непосредственно из пользовательского пространства в ядро. В данном случае на стороне ядра невозможно определить, был ли сделан данный вызов в ходе нормального хода выполнения приложения, либо злоумышленник изменил нормальный ход выполнения и выполнил данный вызов с целью повышения прав.

4.6 Предлагаемый подход

Предлагаемый подход заключается в создании модуля ядра, который имел бы возможность наблюдать за ходом исполнения приложения используя систему utrace-uprobes. Информацию о состояниях приложения возможно получать из внешнего файла, который был бы подобен файлу политики SELinux, но содержал бы лишь необходимую информацию и был бы очень компактен. При этом предлагается не вносить никаких изменений в язык SELinux и использовать те конструкции, которые использует система динамического изменения контекста. При этом в ядре понадобится функция, подобная selinux_setprocattr(), которая должна экспортироваться в остальное пространство ядра, либо возможно добавление еще одного метода в LSM,

который бы эта функция реализовывала. Преимуществами такого подхода являются:

- Изменения существующей архитектуры SELinux минимальны, необходимо лишь добавление одной функции в пространство ядра.
- Исчезает необходимость изменения существующих приложений путем добавления в их код вызовов методов динамического изменения контекста, использование которых является спорным с точки зрения безопасности.
- С использованием utrace-uprobes появляется удобный механизм наблюдения за практически любыми событиями в приложении и выполнения любых действий в случае их возникновения, в том числе за:
  - Попаданием исполнения на определенные адреса в коде.
  - Исполнением процессом вызовов fork/execx.
  - Системными вызовами.
- Существенно повышается скорость работы из-за отсутствия добавления системных вызовов в пользовательские приложения.

Файл, содержащий информацию о смене контекстов приложений Как уже говорилось ранее, идентификатор безопасности целочисленной величиной, которой ставится в соответствие символьное представление контекста безопасности. Для описания необходимости изменить контекст приложения при попадании исполнения на определенный адрес предлагается использовать следующую конструкцию:

```
context_1 context_2 addr
```

Данная конструкция будет объявлять, что для приложения с контекстом context_1 необходимо произвести смену контекста на context_2 при попадании исполнения на инструкцию по адресу addr. При этом конфигурационный файл, содержащий такие предложения, довольно просто перевести в формат, содержащий следующие структуры данных:

```
typedef struct dyntran_info {
    uint32_t ssid;
    uint32_t tsid;
    long     bpt;
}dyntran_info_t;
```

Преимущества создания такого бинарного файла состоят в следующем: в ядро попадает информация уже в той форме, в которой ее удобно представлять и осуществлять все необходимые манипуляции. В противном же случае приходилось бы при загрузке файла, содержащего символьную информацию производить массу операций связанных с определением наличия указанных контекстов в политике. При работе модуля приходилось бы постоянно производить приведение целочисленных идентификаторов к их строковму представлению и обратно.

Получить целочисленное представление контекста безопасности из строкового можно при помощи функции

```
#include <sepol/policydb/services.h>

int sepol_context_to_sid(const sepol_security_context_t scontext,/*IN*/
        size_t scontext_len,    /* IN */
        sepol_security_id_t * out_sid); /* OUT */
```

Используя лексический и синтаксический разбор строится бинарный файл с указанной структурой. При этом, данная утилита тесно интегрируется с политикой. Стоит отметить, что создание конфигурационного файла, отдельного от политики позволяет применять все изменения в нем прямо на ходу, без пересборки основной политики и без перезагрузки системы. В ряде случаев это помогает экономить время и не останавливать работу системы.

Проведем несколько тестов компилятора конфигурационных данных. Рассмотрим его на следующем файле, который содержит некоторую информацию о двух изменениях типов.

```
system_u:system_r:kernel_t system_u:object_r:security_t 080456AA
system_u:object_r:fs_t     system_u:object_r:file_t     0804234B
```

Если запустить в отладочном режиме checkpolicy и проверить идентификаторы, соотвествующие данным строковым контекстам, мы обнаружим, что они равны 1,2,4 и 5. Дадим на вход нашему модулю данный файл. На выходе получим (с отадочными комментариями)

```
Got token:system_u:system_r:kernel_t
The sid of the token is: 1
Got token:system_u:object_r:security_t
The sid of the token is: 2
Got token:080456AA
```

```
The address of the bpt is 134502058
Got token:system_u:object_r:fs_t
The sid of the token is: 4
Got token:system_u:object_r:file_t
The sid of the token is: 5
Got token:0804234B
The address of the bpt is 134488907
```

Очевидно, что транслятор работает правильно. Заметим, что он должен статически связываться с libsepol — библиотекой работы с бинарными файлами политик.

4.7 Изменения в коде ядра

Особенностью реализации LSM является то, что доступ к реализации методов безопасности имеет лишь код ядра, они не экспортируются для модулей. При этом есть некоторые функции, которые экспортируются из SELinux, они описаны в файле ./linux/security/selinux/exports.c. Предлагается перенести реализацию методов изменения контекста и получения информации о контексте приложения из selinux/hooks.c в сам сервер безопасности в selinux/ss/services.c и экспортировать их в ядро из exports.c. При этом вместо функций, реализующих реакцию на запись в интерфейсы /proc/PID/ в hooks.c предлагается заменить заглушками, которые ничего не делают. Действительно, отсутствует необходимость в двух механизмах смены доменов приложений, кроме этого, было бы правильно полностью убрать получение информации об изменении доменов от самих пользовательских приложений из ядра. Таким образом в exports.c появятся две новые функции

```
int selinux_kern_getprocattr(struct task_struct *p,
        char *name,char **value)
{
    if (selinux_enabled)
        return security_kern_getprocattr(p,name,value);
    else {
        *value = NULL;
        return 0;
    }
}
EXPORT_SYMBOL_GPL(selinux_kern_getprocattr);

int selinux_kern_setprocattr(struct task_struct *p,
```

```
        char *name, void *value, size_t size)
{
    if (selinux_enabled)
        return security_kern_setprocattr(p,name,value,size);
    else {
        value = NULL;
        return 0;
    }
}
EXPORT_SYMBOL_GPL(selinux_kern_setprocattr);
```

Данные функции позволяют получать и изменять контекст приложения из модуля.