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.

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) [1], AppArmor (Immulinux, Novell) [2], PaX(GRSecurity)[3], Seatbelt(Apple). Кроме этого, разработчики некоторых систем пытаются расширить стандартные системы безопасности, улучшая их и добавляя новые способы защиты.

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 распространенных 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 счет 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. Учj т внутреннего состояния контролируемого приложения позволит избежать жj стких обобщj нных ограничений на его поведение.

2.3 Цель работы

Расширение функциональности систем защиты уровня ядра Linux (SELinux) за счj т 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 простановки контрольных точек на уровне исходных текстов и бинарных программ.
- Разработка программной инфраструктуры поддержки контроля состояния приложения по контрольным точкам для ОС GNU/Linux.
- Расширение профилей SELinux, добавление возможности переключения контекста безопасности при изменении состояния приложения.

3 Существующие системы безопасности уровня ядра ОС

Рассмотрим некоторые системы безопасности, приведенные выше. Так как решение задачи контроля за текущим состоянием приложения планируется реализовать

с использованием SELinux, будет рассмотрена архитектура этой системы и системы AppArmor, которая является во многом схожей с SELinux.

Кроме этого, кратко будут рассмотрены особенности систем PaX и TrustedBSD.

3.1 SELinux

SELinux является системой безопасности уровня ядра Linux, основанной на подсистеме LSM. LSM позволяет создвать модули безопасности. Данные модули должны реализоывать определенную логику принятия решений о разрешении или запрещении различных взаимодействий между объектами и субъектами системы.

Под объектами системы здесь понимаются файлы, объекты межпроцессного взаимодействия, объекты сетевого взаимодействия и прочие. Субъекты представляют собой пользовательские процессы, демоны, ядро и.т.д..

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 sатаками нулевого дняї , когда уязвимость уже известна злоумышленнику, но лекарства еще не было выпущено. В таких случаях уязвимость локализируется 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 aw (oct 2007).

Конфигурация политик является сложной задачей из-за необходимости описывать профили для каждого прилоежния вручную на на специальном языке описания политик. Добавление новых профилей может повлечь за собой необходимость в модификации уже имеющихся профилей, что может привести к появлению ошибок и росту накладных расходов на администрирование системы.

Основные понятия.

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_ le, chr_ le, 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 различны для различных классов объектов. 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 процессов Домены) делят субъекты и объекты на родственные группы. Это главный атрибут безопасности, используемый SELinux для принятия решений.

Типы позволяют помещать процессы в sпесочницыї и предотвращать повышение привилегий. К примеру, роль суперпользователя - sysadm_r, его тип sysadm_t. Политика безопасности SELinux загружается системой из бинарного файла политики, ко-

торый, 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 ипользования 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; set les 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. Последнее реализуется путем изменения домена приложения. Приложение должно запросить у ядра ОС 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

AppArmor не использует явную разметку всех объектов в системе. AppArmor контролирует поведение приложений, опираясь на профили поведения приложений, описанные на некотором интерпретируемом языке. 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.

Профиль состоит из записей, которые содержат полные пути до файлов или каталогов каталогов с указанием полных и указания прав доступа к ним. 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 sисполняемыйї , 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.

PaX может рассматриваться как дополнение к таким системам безопасности, как SELinux и AppArmor. 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 внедрение и исполнение кода с повышенными привилегиями. Address Layout Randomisation (ASLR) позволяет предотвратить все три упомянутые вида 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. 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_ifo — позволяет администраторам контролировать сетевой трафик.

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

mac_bsdextended (le system rewall) 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 Резульаты рассмотрения существующих средств

Существующие системы безопасности уровня ядра ОС Linux предоставляют широкие возможности контроля за поведением приложений. В частности, SELinux предоставляет возможность создавать приложения, интегрированные с этой системой безопасности. Тем не менее, этот метод не является удовлетворительным по ряду причин. Предлагается решить задачу контроля за состоянием приложения на основании системы безопасности SELinux. Основной целью данного решения должно стать обеспечение независимости приложений в системе от системы, наблюдающей за их состоянием.

4 Решение задачи

4.1 Схема работы решения

Работу инфраструктуры поддержки контроля состояния приложения можно представить в виде последовательности шагов.

- Выделение некоторого набора адресов контрольных точек из программы.
- Сопоставление информации о контрольных точках информации о смене состояний приложения
- Наблюдение за немодифицированным приложением во время его исполнения и смена его контекстов согласно информации об изменении состояний. При этом смена контекста должна происходить абсолютно прозрачно для приложения.

Это значит, что в том случае, если злоумышленнику удается получить контроль над приложением, для изменения его контекста безопасности ему придется воспроизвести нормальный ход исполнения этого приложения до смены конекста в какой-либо контрольной точке.

Рассмотрим эти шаги более подробно.

Пусть в наличии имеются исходные тексты приложения и информация о том, как приложение было собрано. Изначально, на этапе подготовки, человек расставляет метки в исходных текстах программы. Эти метки являются адресами в виртуальном адресном пространстве приложения. При помощью метода, описанного в разделе "Реализация механизма получения адресов контрольных точек при компиляции приложения попадают в отдельную секцию в результирующем исполняемом файле. Далее выполняются все операции именно с этими метками.

Имея в наличии адреса в коде и информацию о смене состояний, связанную с этими контрольными точками, человеком создается конфигурационный файл, формализующий связь между информацией о состояниях и контрольными точками. После этого данный файл транслируется в специальное бинарное представление, подходящее для использования в модуле контролья за состояниями приложения. Данный способ более подробно описывается в подразделе "Сопоставление информации о контрольных точках информации о смене состояний приложения".

Во процессе работы программы при помощи средства utrace, рассмотренном в разделе ***, обнаруживаются "попадания"на метки. Данные события приводят к смене контекста приложения. Смена контекста производится при помощи добавленных в ядро функций, которые описаны в разделе "Особенности реализации".

4.2 Реализация механизма получения адресов контрольных точек.

Рассмотрим процесс этап подготовки программы и получение адресов меток непосредственно из бинарного исполняемого файла.

В данной работе под контрольной точкой подразумевается адрес в виртуальном адресном пространстве процесса. Данные контрольные точки разграничивают внутренние состояния приложения. 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 расширения GCC, позволяющего управлять размещением данных в результирующем коде приложения. Предлагается при помощи этого расширения создавать секцию в бинарном исполняемом файле, содержащую ареса контрольных точек.

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 определяют две различные ветви исполнения программы. Средства компилятора gcc позоляют управлять размещением данных в бинарном файле программы при помощи команды __attribute__. При помощи этой команды в исполнимом файле возможно создать отдельную секцию, содержащую эти адреса. Исполнимые файлы с данной секией и без нее будут отличаться только наличием этой секции, при этом в файле с данной секцией все адреса останутся теми же, что и в файле без секции. Таким образом мы получаем очень удобный способ хранения адресов прямо в бинарном файле программы, откуда их можно извлекать для дальнейшей обработки, либо читать эту информцию прямо перед запуском приложения. Важно, что в большинстве случаев адреса в оптимизированном коде с метками не отличаются от адресов в оптимизированном коде без меток. Это было показано в ходе серии экспериментов.

4.3 Связь информации об адресах контрольных точек с изменениями контекста приложения

Предлагается хранить информацию о состояниях приложения и контрольных точках в файле, который содержал бы необходимую информацию и был бы компактен. 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.

4.4 Наблюдение за немодифицированным приложением во время его исполнения и смена его контекстов согласно информации об изменении состояний.

Обычным интерфейсом отладки программ в ОС Linux является системный вызов ptrace(). Этот вызов является основой для построения отладчиков в пользовательском пространстве. Тем не менее, было решено весь механизм наблюдения за контрольными точками внести в ядро. Такое решение было принято с целью минимизации накладных расходов, так как отладка приложений, собранных с отладочной информацией ведет к серьезной потере производительности.

Не так давно появилась гораздо более удобная альтернатива использованию ptrace для отладки пользовательских приложений из пространства ядра в виде utrace [6]. Основной код данной системы не имеет интерфейсов в пользовательском пространстве. Вместо этого есть интерфейсы в ядре, которые позволяют создавать отладочные механизмы, работающие в пространстве ядра. Эти интерфейсы основаны на концепции отладочного движка , который представляет собой обычную структуру, содержащую указатели на функции. У данной стркутуры есть 14 указателей на функции, которые будут вызваны в случае определенных событий в отлаживаемом приложении. Данная

подсистема позволяет отслеживать самые разные события в отлаживаемом приложении такие, как системные вызовы, сигналы, изменения данных. Клиентом подсистемы utrace является uprobes — набор функций для расстановки точек останова в отлаживаемое приложение. Предлагается использовать uprobes и utrace для наблюдения за контрольными точками и такими событиями, как fork/exec. Далее данная подсистема будет описана более подробно. Информацию о точках останова и о контекстах модуль будет получать из бинарного файла конфигурации.

5 Особенности реализации

5.1 Описание подсистемы utrace-uprobes

Как было сказано ранее, система utrace-uprobes используется для определения "попадания"на контрольные точки. При этом utrace предоставляет средства создания отладчиков в виде модулей ядра, в то время, как uprobes использует utrace для расставления точек останова в теле процесса и наблюдения за попаданием исполнения на них. Рассмотрим особенности использования этой системы более подробно. Как было сказано выше, utrace является подсистемой ядра, позволяющей создавать отладчики, работающие в пространстве ядра в виде модулей. При этом, модуль должен реализовать некоторые функции-обработчики событий, происходящих 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 вызвана соответствующая функция отлаживаемого движка - report_sysc all_entry() (разумеется, если она была зарегистрирована). Вызов данного обработчика происходит до выполнения системного вызова, отладчик может безопасно полчать доступ к остановленному отлаживаемому процессу. Функция-обработчик возвращает битовую маску, которая определяет, что должно произойти далее — можно изменять состояние отладки, прекращать отладку, скрывать событие от других отладочных движков и многое другое.

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 ассоциирует отладочный движок к указанным процессом. Возможна регистрация более чем одного отладочного движка для одного и того же процесса серьезное отличие от ptrace(). Только что зарегистрированный 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), который может переключать процесс в состояние ожидания. В общем случае, все операции с процессом в первую очередь требуют установки этого флага, после чего можно ожидать исполнения коллбека report_quiesce(), который извещает об остановке процесса. Естьмножество других событий, извещения о которых могут быть получены отладочным движком. В их числе fork(), exec(), получение сигнала, завершение процесса, вызов системного 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. Кроме этого, существуют патчи, позволяющие интегрировать uprobes непосредственно в ядро Linux. Основной функцией данного набора функций является обеспечение возможности проставления контрольных точек в код отлаживаемого процесса и регистрация функций, обрабатывающих события, связанные с данными точками. Есть два типа таких контрольных точек: uprobes и uretprobes. Uprobe может быть установлена на любой адрес в виртуальном адресном пространстве процесса и сработает при попадании исполнения на инструкцию, расположенную по этому адресу. Uretprobe сработает при завершении работы указанной функции в отлаживаемом процессе. При регистрации точки останова, uprobes сохраняет копию инструкции, расположенной по этому адресу в приложении, останавливает его исполнение, подменяет первые байты по этому

адресу на инструкцию точки останова (int3 на i386 x86_64) и вновь запускает исполняемое приложение. Когда исполнение попадает на эту инструкцию, срабатывает ловушка и генерируестя сигнал SIGTRAP. Uprobes получает этот сигнал и находит связанную с ним точку останова и ее функцию-обработчик. Отлаживаемый процесс будет остановлен до завершения работы функции-обработчика. После завершения работы функции-обработчика uprobes исполняет сохраненную команду, которая первоначально располагалась по адресу точки останова в пользовательском процессе 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 процесса 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.

5.2 Архитектура SELinux и ее изменения.

Рассмотрим более подробно архитектуру SELinux, существующую систему динамической смены контекста и изменения, которые были в нее внесены.

Итак, как уже было сказано, в SELinux три аттрибута безопасности: идентификатор пользователя, роль и тип вместе образуют так называемый контекст безопасности. SELinux хранит контексты безопасности в своих таблицах, каждая запись в которых определяется идентификатором безопасности, (SID), который представляет собой целочисленную переменную. Разным контекстам ставятся в соответствие разные идентификаторы безопасности. При этом Security Server принимает все решения, описаные в логике политики на основании двух идентификаторов взаимодействующих объектов.

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 принятие решений о разрешении о различных операциях в соответствии с политикой безопасности. Кроме этого, Security Server ведет логи для определенных разрешенных или запрещенных операций, список которых описан в политике. Кроме этого Security Server заполняет соответствующие структуры 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 структуры 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 данного поля в Security Server принимаются решения согласно логике политики SELinux.

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

Политика безопасности SELinux Сервер безопасности принимает все свои решения на основании политики безопасности, описанной администратором системы. При запуске системы SELinux загружает политику безопасности из бинарного файла, который, как правило находится в /etc/security/selinux.

Динамическое переключение контекстов в SELinux

Как уже упоминалось выше, в SELinux существует метод динамического переключения контекста. Рассмотрим его более подробно. 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 исполнения, 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. Упомянутые выше 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.

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.

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.

Конфигурационный файл для модуля

Конфигурационный файл, из которого модуль ядра должен получать информацию об адресах точек останова и изменениях контекстов безопасности имеет простую структуру. Он содержит стуктуры данных:

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

Преимущества создания такого бинарного файла 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.

6 Заключение

Предложенный метод решения задачи позволяет осуществлять контроль за состояниями приложения во время исполнения.

Метод проставления контрольных точек на уровне исходных текстов приложения позволяет отслеживать изменения состояний исходных приложений, без необходимости их изменять. В данной работе задача получения адресов контрольных точек решена при помощи использования расширений компилятора GCC для создания секции, содержащей ареса в виртуальном адресном пространстве приложения. В тех случаях, когда нет доступа к исходным текстам данная задача может быть решена при помощи отладки приложения.

Разработанная программная инфраструктура поддержки контроля состояния приложения по контрольным точкам использует существующие решения, в числе котороых система разработки отладчиков, работающих в пространстве ядра, существующиая в SELinux система динамических изменений контекстов. Система контроля

за состоянием приложения может быть сравнительно легко расширена до контроля за несколькими процессами, в том числе и за процессами, порожденными вызовами fork/exec.

Список литературы

[1] Официальная документация SELinux http://www.nsa.gov/research/selinux/docs.shtml

[2] Документация по проекту AppArmor http://en.opensuse.org/AppArmor_Geeks

[3] Патчи PaX http://pax.grsecurity.net/

[4] Разработка драйверов в ОС Linux (Linux Driver Development by Johnatan Corbet, Greg Kroah-Hartman)

[5] Ядро Linux (Daniel P. Bovet, Marco Cesati)

[6] Патчи utrace http://people.redhat.com/roland/utrace/