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 Обзор существующих систем безопасности уровня ядра ОС

1.1 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 попыток различных разработчиков создать более современные и продвинутые системы 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).

Рассмотрим самые значимые и распространенные системы из перечисленных и сравним их по некоторым критериям, о которых речь пойдет ниже.

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/
Trusted Solaris	SUN	http://www.sun.com/software/solaris/trustedsolaris/index.xml
СБ ОС Феникс	СЦЗИ СПбГТУ	http://www.ssl.stu.neva.ru/fenix
Vista kernel security	Microsoft	http://www.microsoft.com

Таблица 1.1

1.2 Критерии сравнения

Для сравнительного анализа были выбраны следующие критерии:

- Класс угроз. Классы угроз, которым способна противостоять система. Большинство из рассмотренных систем не способны противостоять угрозам всех классов. Поэтому на практике часто требуется комбинировать различные СБ.

- Предоставляемые системой методы защиты. Описывает методы, которые СБ способна предложить для защиты от различных классов угроз.

- Удобство настройки и использования. Чем сложнее настройка и использование системы, тем выше вероятность неверной конфигурации и отказа системы. Данный критерий отвечает за то, насколько легко производить настройку.

1.3 Результаты сравнения.

Класс угроз.

SELinux Предоставляет возможность для 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 используется злоумышленником, но лекарства еще не было выпущено. В таких случаях уязвимость закрывается 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). SELinux при необходимости может быть использована для снижения вероятности инсайдерских атак, к примеру, менеджер архивов может архивировать файлы, но иметь к ним доступ на чтение ему совершенно не обязательно.

AppArmor По классам угроз во многом схож с SELinux, предоставляя возможности для изоляции приложений, подверженных эксплойтам. Эти две системы можно охарактеризовать как довольно универсальные, без ярко выраженного направления атак, от которых они защищают.

PaX 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.

Windows Vista kernel level Security Model Система безопасности Vista защищает от атак на драйверы и от изменения системных объектов.

Trusted BSD Класс атак аналогичен классам SELinux и AppArmor.

ОС Феникс На данный момент Феникс находится в стадии разработки. Микроядро обеспечивает изоляцию адресных пространств процессов, тем самым предотвращая изменение одного процесса другим. Кроме этого, обеспечивается доверенность сообщений, то есть, сообщения между процессами не могут быть подделаны

Apple Seatbelt
Аналогично TrustedBSD, SELinux, AppArmor.

1.4 Предоставляемые системой методы защиты.

SELinux является модулем LSM. Предоставляет набор измененных системных утилит, внедряет вызовы в ядро (hooks). Если приложение пытается выполнить какое-либо действие, критичное для безопасности и если это действие разрешено на уровне стандартной системы безопасности Unix, ядро обращается к SELinux за решением. Решение принимается на уровне политики безопасности.

AppArmor является модулем LSM так же, как и SELinux. 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. Кроме этого AppArmor предоставляет инструменты автоматической генерации профилей на основе поведения приложения и возможность производить контроль в двух режимах: режиме обучения и режиме принуждения.

Vista защищает свои системные структуры и процессы от изменения злоумышленником. Для этого ей служат инструменты контроля целостности файлов и процессов, контроля доступа к памяти, система цифровых подписей для драйверов.

ОС Феникс. Основой средств защиты является монитор взаимодействий — специальный компонент, работающий в связке с ядром и осуществляющий контроль взаимодействий в соответствии с политикой безопасности и поставляющий информацию для протокола аудита. Поскольку монитор взаимодействий не реализует никакой модели безопасности, а только контролирует операции УНИДО, для

принятия решения о доступе он взаимодействует со специальным набором компонентов, обеспечивающих возможность реализации любой модели безопасности, основанной на отношениях субъект-объект и атрибутах безопасности.

Trusted BSD. Списки контроля доступа позволяют контролировать взаимодействия между объектами и субъектами в системе, основываясь на некоторой добавочной информации о них. Контроль производится при помощи Mandatory Access Control Framework. Кроме этого система предоставляет широкие возможности аудита событий безопасности.

PaX предлагает механизмы для защиты от исполнения стека, рандомизации размещения адресов внутри адресного пространства (address space layout randomization) .

Apple MAC представляет собой порт интерфейсов TrustedBSD MAC. На них базируется модуль безопасности Seatbelt. Модули из TrustedBSD и SEDarwin пока остаются за пределами системы безопасности Mac OS Leopard. Используя данный модуль возможно как повышение уровня безопасности приложения программистом, так и контроль за выполнением приложения, нормальный ход которого описывается в специальных файлах конфигурации политики.

1.5 Удобство использования

SELinux. Конфигурация политик является весьма сложной задачей, учитывая наличие специального языка написания политик, сложности 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.

AppArmor. Гораздо меньше вероятность необходимости изменения существующих профилей при генерации новых профилей. Правила для приложений основываются на путях к файлам, что упрощает понимаие политик и не требует производить перепомечание всех объектов в системе.

Vista. Не актуально.

PaX. Наложение патча на ядро.

TrustedBSD. Определение удобства использования требует дополнительных исследований.

ОС Феникс. На данный момент находится в стадии разработки и обладает инструментами для администрирования системы из среды ОС Windows.

Apple Seatbelt — Использование готовых конфигураций, либо создание собственных. Языком конфигурации является Scheme-подобный язык, сложный для восприятия и описания профилей. Официально это решение пока отсутствует в Mac OS X.

Система	Угрозы	Возможности	Удобства
SELinux	Предотвращение компрометации системы путем ограничения поведения	Возможность создания пользовательских ролей и контекстов безопасности	Не очень удобен в плане устновки и настройки, сложный язык описания политик.
AppArmor	Аналогично SELinux	Четкое описание объектов, к которым может иметь доступ то или иное приложение и прав доступа.	Язык описания профилей интуитивно понятен, есть возможность автоматической генерации профилей
PaX	Позволяет предотвратить угрозы, в которых злоумышленник атакует некоторые уязвимости в системе	Защита памяти от исполнения, рандомизация адресов в программе.	Патч ядра
Vista	Модификация системных структур и кода ядра, уязвимости драйверов, прямой доступ к памяти	Предоставляет более широкие возможности для настройки и обеспечения безопасности системы, чем более ранние версии	Ничего не нужно делать
TrustedBSD	Предоставляет собой повышенный уровень защиты для соответствия стандартам «оранжевой книги»	Предоставляют более широкие возможности для настройки и обеспечения безопасности системы, чем стандартные версии	Затраты на полное администрирование всей системы
Seatbelt	Предоставляет возможности определения ограничений на поведение приложения, в том числе и помещение приложений в «песочницу»	Предположительно предоставляет более широкие возможности для настройки безопасности системы, но все еще находится в стадии разработки. Отсутствуют порты модулей из родственной TrustedBSD	Сложный язык конфигурации политики, отсутствие документации.

1.6 Обзор общих принципов работы рассматриваемых систем безопасности.

SELinux. Основные понятия.

Принудительное присвоение типов (TE). И для процессов и для объектов используется один и тот же тип атрибутов. Поэтому достаточно одной матрицы достаточно для описания доступа к взаимодействию между разными типами, при этом объекты одного типа могут рассматриваться по-разному, если их их ассоциированные классы безопасности различны. Пользователи не привязаны к типам безопасности напрямую, вместо этого используется RBAC.

Ролевой контроль доступа (RBAC) 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.

Практический обзор

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). Действия, которые субъекты SELinux могут предпринимать 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 (а.к.а. 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/security/selinux. Бинарная политика собирается при помощи make, исходные коды, как правило, находятся в /etc/security/selinux/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.

Краткий обзор анатомии политики SELinux.

Файлы политики организованы в виде дерева каталогов, корнем которого, как правило, является /etc/security/selinux/src/policy/. Основными поддиректориями являются: appconfig (определяет дефолтные типы контекстов безопасности); domains (определяют домены принудительного присвоения типов); file_contexts (определяют контексты безопасности файлов), flask (определяет символы, используемые ядром, совместимым с SELinux), macros (определяет макрос M4, используемый в исходных текстах политик), tmp (хранит сорцы политик во время компиляции), types (определяет несколько главных типов, которые не ассоциируются с конкретными доменами). Как правило существует два файла, которые определяют домен: FC file (file context), определяет контексты безопасности директорий и файлов, связанных с данным доменом); TE file (type enforcement, определяет вектор правил доступа и операций, связанных с доменом). Целью данного обзора не является.

5.2. AppArmor.

AppArmor является системой безопасности, поддерживаемой компанией Novell, включена в дистрибутивы openSUSE и SUSE Enterprise. В AppArmor для определения того, к каким системным ресурсам и с какими привилегиями может получить доступ то или иное приложение используются политики безопасности (profiles). В отличие от SELinux, в которой настройки глобальны для всей системы, профили AppArmor разрабатываются индивидуально для каждого приложения. 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 эксплойта, 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 пред-

писывает 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», l — жесткая ссылка. Чтобы подключить готовый профиль к AppArmor, достаточно его скопировать в каталог /etc/apparmor.d.

PaX

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.

Trusted BSD.

Проект TrustedBSD – проект разработки расширения существующей системы безопасности FreeBSD, включая 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. Это предоставляет место для хранение данные, необходимые системе безопасности, такие, как ACL и метки MAC. Списки контроля доступа — расширения дискреционного контроля доступа. Аудит системных событий позволяет вести избирательный логгинг важных системных событий для последующего анализа, обнаружения вторжений, и мониторинга в реальном времени. 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.

ОС «Феникс».

ОС «Феникс» является отечественной разработкой — разработка СпбГУ, целью которой является создание специальной защищенной операционной системы класса Unix, отвечающей отечественным требованиям и стандартам информационной безопасности. «Феникс» представляет собой микроядерную, многопользовательскую, многозадачную, многопоточную операционную систему класса UNIX со встроенными механизмами защиты, обеспечивающими контроль взаимодействий, управление доступом, контроль целостности, идентификацию/аутентификацию пользователей и возможность подключения средств шифрования. Микроядерная архитектура отвечает принципу интегрированности, поскольку только в микроядерных системах для взаимодействий используется единственный способ — обмен сообщениями. Контроль доступа органично встраивается в этот механизм, причем, установив тотальный контроль над потоками сообщений, можно быть уверенным в том, что контролируются все взаимодействия в системе. Принцип инвариантности определил организацию всех взаимодействий в «Феникс» на основе архитектуры клиент-сервер. В соответствии с принципом унификации доступ к объектам в «Феникс» осуществляется через Унифицированный Интерфейс Доступа к Объектам (УНИДО), определяющий множество операций, универсальных

для всех типов объектов, в виде универсального набора методов, позволяющего выполнять все операции доступа к объектам, их создания и уничтожения, управления их свойствами. Использование УНИДО единственный способ выполнения операций над объектами в «Феникс». Интерфейс оформлен в виде абстрактного класса, от которого наследуются интерфейсы всех серверов «Феникс», реализуемых УНИДО. Наличие набора типовых операций упрощает реализацию контроля доступа, поскольку определено однозначное соответствие между методами УНИДО и операциями доступа, описываемыми моделями безопасности.

Vista Kernel-Mode Security.
В ОС Windows Vista была расширена модель безопасности, присутствовавшая в предыдущих версиях системы(вплоть до XP SP2). Среди нововведений стоит отметить цифровые подписи драйверов, PatchGuard, Kernel-mode Code Integrity Checks, optional support for Secure Bootup using a TPM hardware chip, restricred user-mode access to
Device
PhysicalMemory.

Driver Signing . Анализируя эксплойты уязвимостей прошлых версий ОС, мы можем прийти к выводу, что наиболее распространенный способ, используемый вредоносным кодом для проникновения в ядро — проникновение через драйверы. Поэтому Vista не только требует подписи от драйвера, но и требует подпись именно от одного из восьми доверенных сертификатов.

PatchGuard был разработан для предотвращения патчей ядра ОС Виста x64. Защищает ядро путем периодической проверки на валидность некоторых структур данных и образов системы. PatchGuard кроется в NTOSKRNL.EXE и проверяет особо критичные системные структуры через случайные промежутки времени, обычно порядка 5-10 минут. Если была обнаружена модификация, «system will blue screen with the following bugcheck(which will obviously cause the user to lose all unsaved data)». (возможно ли обнаружить и убить тред PatchGuard?)

Disabling
Device
PhysivalMemory Отказ от возможности доступа к Disabling
Device
PhysivalMemory из пользовательского пространства тоже является серьезным шагом на пути предотвращения доступа вредоносного кода к ядру.

Code Integrity (CI.DLL) Импортируется статически NTOSKRNL. Защищает систему тем, что проверяет системные исполняемые файлы на наличие изменений, в том числе и из-за внедрения вредоносного кода, наличие в системе неподписанных драйверов, запущенных в режиме ядра. В чем же отличие CI от PatchGuard, если они предоставляют схожую функциональность? CI может быть отключен, если отключены integrity checks, PatchGuard всегда включен. Кроме этого, эти методы разрабатывались разными командами внутри Microsoft и по заявлениям разработчиков, служат разным целям.

Возможные направления атак. Kernel-Mode Network Drivers. Виста поддерживает некоторые сетевые протоколы в виде драйверов уровня ядра. Если уязвимость обнаружена в одном их этих подписанных драйверах, это бы дало возможность заполучить удаленный контроль над всей машиной.

Disabling Driver Signing and Code Integrity

Самый простой путь преодоления всех сложностей, связанных с подписями драйверов — патч исполняемых файлов и отключение проверок подписей вообще. Для загрузки неподписанных драйверов во время выполнения NTOSKRNL.EXE должен быть пропатчен. Но, патч ядра несет угрозу его цифровой подписи, следовательно, WINLOAD.EXE откажется загружать ядро.

Apple Seatbelt.

Кроме возможности использования интерфейсов при программировании, позволяет помещать приложения в «песочницу», где их поведение будет контролироваться на основании определенных профилей. Данные профили находятся в /usr/share/sandbox и состоят из allow/deny определений и регулярных выражений для определения прав доступа к ресурсам системы. Объекты определяются по абсолютному пути (POSIX). Пример конфигурационного файла:

```
(version1)
(debug deny)
(allow default)
(allow process*)
(deny network-outbound)
(allow file-read-data file-read-metadata
(regex "^/.*"))
(deny file-write*
(regex "^/.*"))
(allow file-write*
(regex "^/Users/johndoe/Library/Preferences.*"))
(allow file-write* file-read-data file-read-metadata
(regex "^(/private)?/tmp/"))
(import "bsd.sb")
```

Помещение в «песочницу» приложения на Cocoa производится следующим образом: % sandbox-exec -n localonly /Applications/TextEdit.app /Contents/MacOS/TextEdit

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

Задача

Задача контроля текущего состояния приложения требует ввода некоторых определений. Определим состояние, как некоторый участок кода программы , относительно которого может быть принято решение, что для его исполн ения нужны те или иные минимальные права. Одним из решений задачи конт роля является

патчинг кода программы функциями, которые бы сообщали о текущих изменениях состояния приложения. Такой подход не является не у довлетворительным, так как следствием успешной атаки на приложение мож ет стать исполнение произвольного кода, который может исполнить одну и з функций изменения состояний. Другое решение заключается в расстановк е точек останова на определенные адреса в коде. Контроль должен осущес твляться из пространства ядра. Этому есть несколько причин: модуль дол жен вносить некоторые изменения в логику работы SELinux Security Serve r, кроме этого, мы не можем доверять пользовательскому пространству. К роме этого, информация о связи между адресами в коде и изменениях сост ояния приложения должна быть доступна для модуля, осуществляющего конт роль. Таким образом, можно выделить две основные части работы - реализ ация механизма контроля за адресами в коде и расширение профилей SELin ux.

Реализация механизма наблюдения за состояниями процесса.

Итак, в данной работе под контрольной точкой подразумевается некоторый адрес в виртуальном адресном пространстве процесса. Попадание исполне ния на один из таких адресов, в общем случае, означает изменение состо яния процесса. В первую очередь, возникает необходимость некоторым обр азом разметить код приложения контрольными точками, а точнее, получить адреса в виртуальном адресном пространстве приложения. В данной работ е будет рассматриваться только разметка кода приложений, написанных на C/C++ на основании их исходных текстов. Это возможно сделать при помо щи 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.

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.

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 администрирования 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.

Существующий в SELinux метод динамического переключения контекста

В 2004-м году в SELinux началась работа над системой интерфейсов для пользовательских приложений, позволяющих приложениям динамически изменять свой контекст. 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);
```

Логика работы системы динамических изменений контекстов.

Основной целью данной системы является предоатвление возможсности 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, в общем случае 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.

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.