Содержание

Введение

Иногда при работе с Linux-системами необходимо осуществлять перехват вызовов функций внутри ядра (например, открытие файлов или каталогов) для обеспечения возможности мониторинга активности в системе или превентивного блокирования деятельности подозрительных процессов. Перехват вызовов функций внутри ядра может осуществляться различными способами, такими как использование LSM, модификация таблицы системных вызовов, использование kprobes, использование сплайсинга и использование ftrace.

В настоящее время в официальное ядро Linux входят, например, такие security-модули, как AppArmor, SELinux, Smack и TOMOYO. Кроме того, с версии Linux 2.6.1 введена поддержка systrace. Systrace – это служебная программа для обеспечения компьютерной безопасности, которая ограничивает доступ приложений к системе, применяя политики доступа для системных вызовов. Systrace особенно полезен при запуске ненадежных приложений.

Данная работа посвящена исследованию способов перехвата системных вызовов. Целью проекта является разработка загружаемого модуля ядра, позволяющего перехватывать системные вызовы для отслеживания событий файловой системы.

1 Аналитический раздел

В соответствии с заданием на курсовой проект необходимо разработать загружаемый модуль ядра, перехватывающий системные вызовы, связанные с событиями в файловой системе Linux. Модуль должен осуществлять наблюдение за всеми файлами и директориями, записанными в конфигурационный файл модуля. Модуль должен отслеживать следущие события (в скобках указаны соответсвтующие системные вызовы):

- открытие файла (open(), openat());
- создание файла (open(), creat());
- запись данных в открытый файл (write());
- удаление записи из файла каталога (unlink(), unlinkat());
- создание каталога (mkdir(), mkdirat()).

Все произошедшие события модуль должен записывать в log-файл для того, чтобы впоследствии эту информацию можно было считать из пространства пользователя.

Для понимания алгоритма перехвата системных вызовов необходимо сначала рассмотреть, как происходит системный вызов.

1.1 Траектория системного вызова

Системный вызов - это фундаментальный интерфейс между приложением уровня пользователя и ядром Linux. Большую часть времени программы выполняются в пользовательском режиме и переключаются в режим ядра только тогда, когда им требуется служба операционной системы. Услуги операционной системы предоставляются через системные вызовы. Системные вызовы – это «ворота» в ядро, реализованные с помощью программных прерываний. Программные прерывания – это прерывания, создаваемые программой и обрабатываемые операционной системой в режиме ядра. Операционная система поддерживает «таблицу системных вызовов», в которой есть указатели на функции, реализующие системные вызовы внутри ядра.

В любой (в том числе и микроядерной) операционной системе системный вызов выполняется некоторой выделенной процессорной инструкцией, прерывающей последовательное выполнение команд и передающий управление коду режима супервизора. Это обычно некая команда программного прерывания, в зависимости от архитектуры процессора в разные времена это были команды с мнемониками вида: svc, emt, trap, int и им подобными. Если обратиться только к архитектуре Intel x86, то в ней для этого традиционно используется команда программного прерывания с различным вектором. Начиная с определенного момента (примерно с начала 2008 года или момента выхода Windows XP Service Pack 2) многие операционные системы (Windows, Linux) отказались от использования программного прерывания int, и перешли к реализации системного вызова и возврата из него через новые команды процессора sysenter (sysexit) [1], однако ничего принципиально нового не появилось.

Системные вызовы обычно вызываются не напрямую, а через функции оболочки в glibc (или, возможно, в какой-либо другой библиотеке). Все системные вызовы далее преобразуются в вызов ядра функцией syscall(), 1-м параметром которого будет идентификатор выполняемого системного вызова, например __NR_execve.

Системный вызов syscall(), попав в ядро, всегда попадает в таблицу sys_call_table, и далее переадресовывается по индексу (смещению) в этой таблице на величину 1-го параметра вызова syscall() - идентификатора требуемого системного вызова [1].

Рассмотрим теперь возможные способы перехвата системных вызовов в Linux.

1.2 Анализ подходов реализации перехвата системных вызовов

Сущетсвуют следующие способы реализации перехвата системных вызовов в Linux:

- Linux Security Modules (LSM);
- модификация таблицы системных вызовов;
- использование kprobes;
- сплайсинг;

- использование ftrace.

1.2.1 Linux Security Modules

Интерфейс LSM позволяет ядру Linux поддерживать различные модели компьютерной безопасности. LSM был создан для решения проблемы контроля доступа и является частью ядра начиная с Linux версии 2.6. LSM вставляет перехватчики (security-функций, которые в свою очередь вызывают обратные вызовы, установленные security-модулем) в каждую критическую точку ядерного кода, где системные вызовы уровня пользователя получают доступ к важным внутренним объектам ядра, таким как inode. Security-модуль может изучать контекст операции и принимать решение о её разрешении или запрете.

В частности, для файловых операций были определены три набора перехватчиков: перехватчики файловой системы, перехватчики inode и перехватчики файлов. LSM добавляет поле безопасности в каждую из связанных структур данных ядра: суперблок, индексный дескриптор и файл. Перехватчики файловой системы позволяют модулям безопасности управлять такими операциями, как, например, монтирование.

Название «модуль» несколько неверно, поскольку security-модули на самом деле не являются загружаемыми модулями ядра, а подключаются к ядру во время его сборки. Соответственно, Linux Security API имеет важное ограничение: security-модули не могут быть загружены динамически, являются частью ядра и требуют его пересборки.

Таким образом, несмотря на то, что LSM был разработан именно для мониторинга системных вызовов, для его использования необходимо поставлять собственную сборку ядра, а также интегрировать дополнительный модуль с SELinux или AppArmor, которые используются популярными дистрибутивами.

1.2.2 Модификация таблицы системных вызовов

Сохранив старое значения обработчика и подставив в таблицу системных собственный обработчик, мы можем перехватить любой системный вызов. Таблица указателей на функции ядра, которые реализуют системные вызовы, расположена в массиве sys_call_table. Такой подход известен программистам еще со времен MS-DOS.

Для перехвата системных этим способом используется механизм загружаемых модулей ядра. Для реализации модуля, перехватывающего системный вызов, необходимо определить алгоритм перехвата. Алгоритм следующий:

1. сохранить указатель на оригинальный (исходный) вызов для возможности его восстановления;
2. создать функцию, реализующую новый системный вызов;
3. в таблице системных вызовов sys_call_table произвести замену вызовов, т.е настроить соответствующий указатель на новый системный вызов;
4. по окончании работы (при выгрузке модуля) восстановить оригинальный системный вызов, используя ранее сохраненный указатель.

Данный подход имеет следующие преимущества:

- полный контроль над любыми системными вызовами;
- минимальные накладные расходы;
- минимальные требования к ядру.

Однако метод имеет и недостатки:

- техническая сложность реализации (необходимо найти таблицу системных вызовов, обойти защиту от модификации таблицы, выполнить замену атомарно и безопасно);
- невозможность перехвата некоторых обработчиков (некоторые обработчики реализованы на языке ассемблера, и их сложно или даже невозможно заменить на свои обработчики, написанные на C);
- перехватываются только системные вызовы (точки входа ограничиваются только системными вызовами, а все дополнительные проверки выполняются либо до непосредственного сиситемного вызова, либо после, поэтому необходимо дублировать проверки на адекватность аргументов).

1.2.3 Использование kprobes

Kprobes – это механизм отладки для ядра Linux, который также можно использовать для мониторинга событий внутри ядра. Этот механизм позволяет вставлять точки останова в работающее ядро. С помощью kprobes можно прервать выполнение ядерного кода в любом месте и вызвать свой обработчик. Этот интерфейс позволяет устанавливать пред- и постобработчики для любой инструкции в ядре, а также обработчики на вход и возврат из функции.

Для добавления своего собственного зонда (probe) в работающее ядро необходимо написать загружаемый модуль ядра, который реализует предварительный обработчик и пост-обработчик для зондирования.

Преимущества использования kprobes:

- зрелый API. Kprobes существуют и улучшаются с 2002 года;
- перехват любого места в ядре. Kprobes реализуются с помощью точек останова (инструкции int3), внедряемых в исполнимый код ядра. Это позволяет устанавливать kprobes в буквально любом месте любой функции, если оно известно.

Недостатки kprobes:

- техническая сложность. Kprobes — это только способ установить точку останова в любом места ядра. Для получения аргументов функции или значений локальных переменных надо знать, в каких регистрах или где на стеке они лежат, и самостоятельно их оттуда извлекать;
- ограничения kretprobes. Kretprobes реализуются через подмену адреса возврата на стеке. Соответственно, им необходимо где-то хранить оригинальный адрес, чтобы вернуться туда после обработки kretprobe. Адреса хранятся в буфере фиксированного размера. В случае его переполнения, когда в системе выполняется слишком много одновременных вызовов перехваченной функции, kretprobes будет пропускать срабатывания;
- при обработке зондов (probes) приоритетное прерывание отключено. Это накладывает определённые ограничения на обработчики: в них нельзя выполнять операции ввода-вывода, спать в таймерах и семафорах;

- В текущей реализации kprobes существуют некоторые задержки в работе, причиной которых является kprobe_lock, который сериализует выполнение зондов на всех ЦП на машине SMP. Другая причина – это механизм kprobes, который использует несколько исключений для обработки одного зонда. Обработка исключений – дорогостоящая операция, которая вызывает задержки.

1.2.4 Сплайсинг

Сплайсинг – это метод перехвата функций путём изменения кода целевой функции. Инструкции в начале целевой функциии заменяются на безусловный переход, ведущий в нужный нам обработчик. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в перехваченную функцию. С помощью двух переходов мы вшиваем (splice in) свой дополнительный код в функцию, поэтому такой подход называется сплайсингом. Методом сплайсинга реализована jump-оптимизация для kprobes.

Преимущества сплайсинга:

- минимальные требования к ядру. Сплайсинг не требует каких-либо особенных опций в ядре и работает в начале любой функции. Нужно только знать её адрес;

- минимальные накладные расходы. Необходимо всего лишь два безусловных перехода. Подобные переходы отлично предсказываются процессором и являются очень дешёвыми.

Однако сплайсинг имеет один серьезный недостаток: высокая техническая сложность реализации. Нельзя просто так взять и переписать машинный код. Для этого необходимо синхронизировать установку и снятие перехвата; обойти защиту на модификацию регионов памяти с кодом; дизассемблировать заменяемые инструкций, чтобы скопировать их целыми. В режиме ядра необходимо запретить прерывания для избежания переключения задач, так как при замене кода в начале функции перехватываемая функция может понадобиться другому потоку.

1.2.5 Использование ftrace

Ftrace — это внутренний трассировщик ядра, позволяющий разработчикам посмотреть, что происходит внутри ядра системы. Ftrace был включен в основную

линию ядра Linux в версии 2.6.27, выпущенной в 2008 году. С помощью ftrace можно отслеживать контекстные переключения, измерять время обработки прерываний, высчитывать время на активизацию заданий с высоким приоритетом и многое другое.

Ftrace полагается на механизм профилирования gcc для добавления машинных инструкций к скомпилированным версиям всех функций ядра, которые перенаправляют выполнение функций на плагины трассировщика ftrace, которые выполняют фактическую трассировку. В начало каждой функции добавляется вызов специальной трассировочной функции mcount() или __fentry__().

Ftrace поддерживает динамическое отслеживание вызовов функций ядра. Ядро знает расположение всех вызовов mcount() или __fentry__() и на ранних этапах загрузки заменяет их машинный код на nop — специальную инструкцию, которая предписывает ничего не делать. При включении трассирования в нужные функции вызовы ftrace добавляются обратно. Таким образом, если ftrace не используется, то его влияние на систему минимально.

Преимущества использования данного подхода:

- зрелый API и простой код. Использование готовых интерфейсов в ядре существенно упрощает код. Вся установка перехвата требует пары вызовов функций,заполнение двух полей в структуре;
- подход автоматически совместим с вытеснением, в отличие от kprobes;
- нет ограничений на функции. Подход с ftrace лишён недостатка kretprobes и из коробки поддерживает любое количество активаций перехватываемой функции;
- перехват любой функции по имени. Можно перехватить любую функцию (даже неэкспортируемую для модулей), зная лишь её имя;
- перехват совместим с трассировкой. Очевидно, что этот способ не конфликтует с ftrace, так что с ядра всё ещё можно снимать очень полезные показатели производительности.

Однако, ftrace, как и другие подходы, не лишен недостатков:

- требования к конфигурации ядра. Для поддержки ftrace ядро должно предоставлять целый ряд возможностей (список символов kallsyms для поиска функций по имени; фреймворк ftrace в целом для выполнения трассировки; опции ftrace, критически важные для перехвата);
- оборачиваются целиком вызовы функций. ftrace срабатывает исключительно при входе;

Несмотря на описанные недостатки следует учитывать, что обычно ядра, используемые популярными дистрибутивами, все необходимые ftrace опции в себе всё равно содержат, так как они не влияют на производительность и полезны при отладке. Иметь в виду эти требования стоит, если необходимо поддерживать какие-то особенные ядра. Оборачивание функций целиком в целом удобно, но для каких-либо специфических задач может и не подходить.

При использовании ftrace стоит учитывать, что использование kprobes или сплайсинга может помешать механизмам ftrace.

1.3 Вывод

В данном разделе были проанализированы возможные подходы к реализации перехвата системных вызовов.

Для использования LSM необходима пересборка ядра и интеграция нового модуля с AppArmor, SELinux, Smack и TOMOYO, которые используются в популярных дистрибутивах. Модификация таблицы системных вызовов, использование kprobes или сплайсинга характеризуются высокой технической сложностью реализации. Поэтому наиболее подходящим вариантом является использование трассировщика ftrace.

2 Конструкторский раздел

Для реализации подхода с ftrace необходимо реализовать загружаемый модуль ядра. Также можно реализовать bash-скрипт. В данной работе, согласно заданию, будет реализован загружаемый модуль ядра.

Список литературы

[1] Цилюрик О.И. Модули ядра Linux. Модификация системных вызовов. [Электронный ресурс]. – Режим доступа: http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/08/kern-mod-08-04.html, свободный – (27.11.2020).

[2] Linux Security Modules: General Security Hooks for Linux [Электронный ресурс]. – Режим доступа: https://www.kernel.org/doc/html/latest/security/lsm.html, свободный – (27.11.2020).

[3] Linux Security Modules: General Security Support for the Linux Kernel [Электронный ресурс]. – Режим доступа: https://www.usenix.org/legacy/events/sec02/full_papers/wright/wright_html/index.html, свободный – (27.11.2020).

[4] syscalls(2) — Linux manual page. – Режим доступа: https://man7.org/linux/man-pages/man2/syscalls.2.html, свободный – (27.11.2020).

[5] Kernel Probes (Kprobes). – Режим доступа: https://www.kernel.org/doc/Documentation/kprobes.txt, свободный – (28.11.2020).

[6] Kernel debugging with Kprobes. – Режим доступа: https://www.ibm.com/developerworks/library/l-kprobes/index.html, свободный – (28.11.2020).

[7] Перехват функций в ядре Linux с помощью ftrace: https://habr.com/ru/post/413241/, свободный – (28.11.2020).

[8] Loadable Kernel Module Programming and System Call Interception: https://www.linuxjournal.com/article/4378, свободный – (28.11.2020).

[9] ftrace - Function Tracer: https://www.kernel.org/doc/html/latest/trace/ftrace.html, свободный – (28.11.2020).