УДК 004.451; 004.72:004.451

Особенности доступа к системным функциям ядра ОС GNU/Linux

А. М. Каннер
Закрытое акционерное общество "ОКБ САПР", Москва, Россия

В. П. Лось, д-р техн. наук
Московский государственный индустриальный университет (МГИУ), институт криптографии, связи и информатики Академии ФСБ России, Москва

Рассмотрены проблемы разграничения доступа в ОС GNU/Linux, связанные с особенностями семейства ОС.

Ключевые слова: ОС GNU/Linux, интерфейс системных вызовов, перехват системных вызовов, LSM.

Ядро Linux на сегодняшний день, пожалуй, является единственной унифицированной и единой составляющей каждого дистрибутива Linux. Большинству крупных вендоров (RedHat, Novell, Canonical и т. д.) доступно внесение определенных изменений в исходные коды ядра, но такие изменения либо не должны конфликтовать с эталонным кодом (так называемым "ванильным" ядром), либо будут в дальнейшем внесены в основную ветку ядра, таким образом *ядро едино для всех*.

Учитывая данную особенность, для ОС семейства Linux существует возможность достаточно просто (за исключением некоторых вопросов совместимости с предыдущими версиями ядра) разрабатывать различные расширения — модули ядра, которые фактически являются частью ядра и могут переопределять/дополнять различные его функции, т. е. *изменять порядок работы ядра ОС Linux*. Причем такие модули, с некоторыми оговорками, будут совместимы с большинством дистрибутивов Linux без внесения в их код каких-либо изменений.

Таким образом, ядро Linux является монолитным (т. е. содержит в себе достаточный функционал для нормального функционирования системы без прочих дополнений/расширений), но при этом поддерживает *загружаемые модули ядра* (LKM — Linux Kernel Modules или Loadable Kernel Modules), которые выполняются в 0-м кольце защиты, с полным доступом к оборудованию, причем загрузка/выгрузка таких модулей может осуществляться во время работы системы (ядра ОС) без перезагрузки.

На первый взгляд такой подход может показаться проблемным с точки зрения безопасности, но необходимо понимать, что:

- все модули ядра могут загружаться/выгружаться в пространство ядра Linux только с правами суперпользователя (root);
- в самом ядре существуют специальные механизмы, предотвращающие выгрузку критичных модулей ядра в момент их работы (по умолчанию ядро собирается с опцией MODULE_FORCE_UNLOAD=0, т. е. без возможности принудительной выгрузки модулей ядра с параметром --force — 'rmmod --force module.ko');
- самим модулям в явном виде не разрешено совершать действия, которые могут влиять на работающую систему (например, изменять данные структур запущенных процессов, осуществлять доступ к памяти ядра и т. п.). Для таких действий требуются предварительные манипуляции, однако потенциально эти действия возможны и не запрещены при отключении блокировок типа GFP (General Fault Protection).

В связи с этим можно утверждать, что загружаемые модули ядра не могут сами по себе являться средством повышения привилегий в системе и/или быть уязвимостью системы. Целесообразность использования загружаемых модулей ядра с точки зрения злоумышленника заключается в сокрытии собственного присутствия в системе и не более, т. е. в действиях непосредственно после взлома отдельно взятой системы.

Чем могут быть полезны загружаемые модули ядра ОС Linux в плане повышения безопасности системы и/или разработки "навесных" средств защиты? Такие модули можно использовать для перехвата системных функций ядра с целью организации своей, внешней по отношению к ОС, подсистемы разграничения доступа в ОС.

Интерфейс системных вызовов ядра ОС Linux

Интерфейс системных вызовов (system call interface) является прослойкой в ядре ОС, используя которую прикладное ПО (из пользовательского режима) может получать доступ к оборудованию, работать с файловыми системами и т. п. Таким образом, фактически любое действие в системе требует вызова того или иного системного вызова (будь то запись/чтение файла, изменение прав доступа, запуск нового процесса или любые действия с выделением/освобождением памяти), а общую схему работы с системными вызовами можно схематично представить на рис. 1.

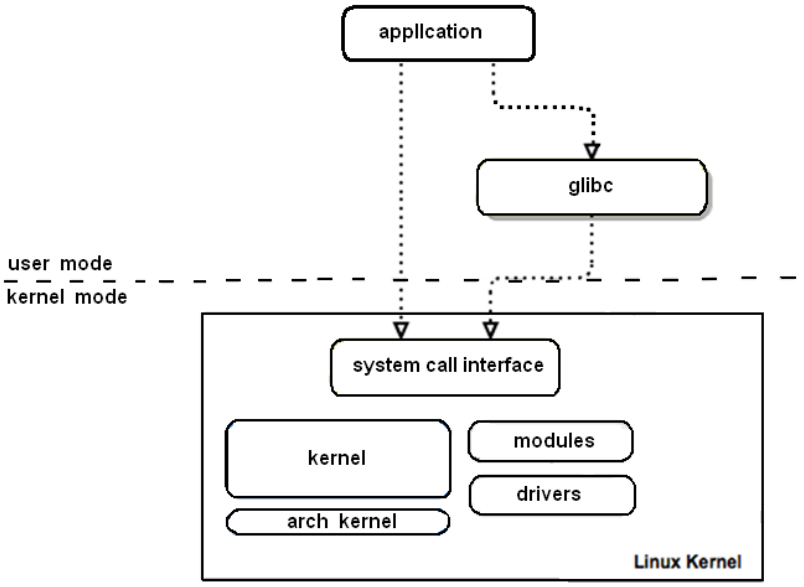

Рис. 1. Использование системных функций ядра ОС Linux пользовательскими приложениями

Итак, любое приложение, используя функции стандартной библиотеки (glibc), неявно для себя вызывает определенные системные вызовы (system_call), специальная процедура ядра ОС просматривает таблицу системных вызовов (sys_call_table), находит адрес нужной функции ядра, вызывает эту функцию и передает управление обратно приложению только после выполнения этого системного вызова (при этом ядро также осуществляет разграничение доступа разных приложений к одним и тем же системным вызовам по времени).

Перехват системных вызовов

Что требуется для внедрения собственных механизмов безопасности в ОС Linux? Фактически требуется написать собственные реализации основных системных вызовов, которые в зависимости от успешности/неуспешности определенных проверок будут вызывать/не вызывать выполнение эталонных системных вызовов. Таким образом становится возможным реализовать собственные дискреционные и мандатные механизмы разграничения доступа (или любые другие), которые будут работать непосредственно до отработки всех штатных подсистем разграничения доступа ОС. Сами функции, "переопределяющие" работу системных вызовов логично описать в загружаемом модуле ядра, а при инициализации этого модуля необходимо заменять адреса системных вызовов в таблице системных вызовов на адреса переопределяющих их функций (при этом желательно при выгрузке модуля совершать обратное действие) (рис. 2).

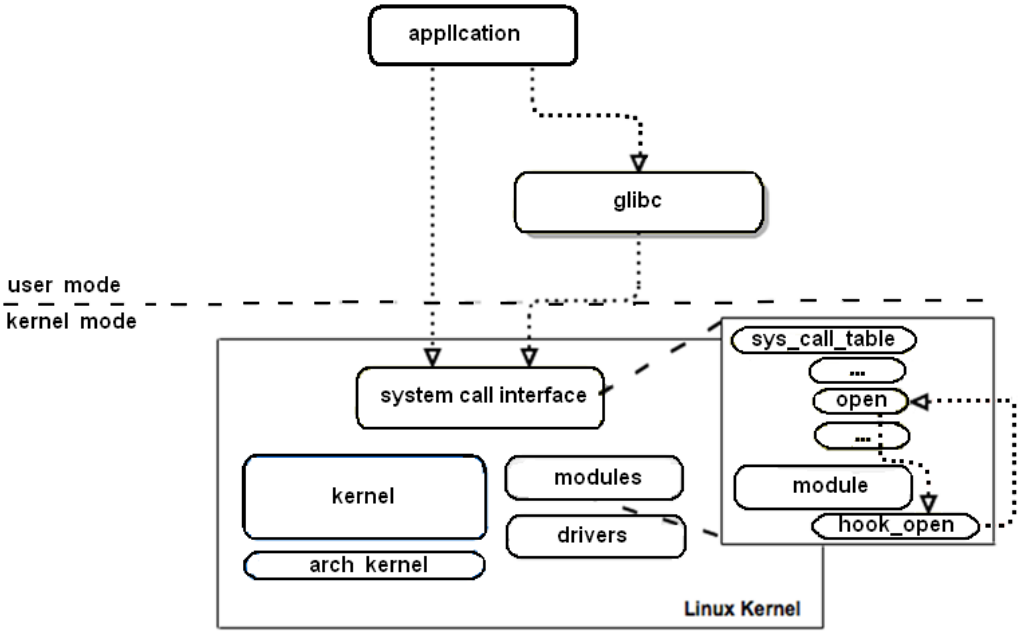

Рис. 2. Перехват системного вызова с помощью модуля ядра Linux

Реализовать описанный выше подход можно в случае, когда нам известен адрес таблицы системных вызовов в памяти ядра Linux. Кроме того, несмотря на всю простоту данного подхода, в нем кроется большая проблема, попробуем описать ее на примере.

Предположим, у нас есть 2 загружаемых модуля ядра ОС Linux module_A и module_B и каждый из них изменяет адрес одного и того же системного вызова 'open' на адрес своей функции (open_A и open_B, соответственно). Первым загружается module_A и заменяет адрес в sys_call_table[__NR_open] на адрес своей функции open_A. Затем загружается module_B и заменяет адрес open_A адресом своей функции open_B (module_B при этом подозревает, что подменил оригинальный системный вызов).

Теперь если module_B выгрузится первым — в системе ничего плохого не произойдет — при выгрузке module_A в таблице системных вызовов будет восстановлен оригинальный вызов 'open'. Однако, если же первым выгрузится module_A, будет восстановлен оригинальный системный вызов 'open', а при выгрузке module_B системный вызов 'open' будет заменен на open_A (которой вообще говоря уже нет в памяти).

На первый взгляд проблему можно решить, заменяя адрес обратно только в случае, если адрес в таблице системных вызовов совпадает с адресом функции модуля (open_A или open_B), но в таком случае при выгрузке в качестве первого модуля module_A — адрес sys_call_table переписан не будет (т. е. не вернется оригинальный системный вызов 'open'), таким образом данный подход также не решает проблему.

Для предотвращения потенциальной опасности, связанной с изменением адресов системных вызовов, начиная с версии ядра ОС Linux >= 2.5.41, более не экспортируется адрес таблицы системных вызовов (sys_call_table). Таким образом, физически найти и подменить адрес нужного нам системного вызова стало сложнее.

Использование механизма LSM для перехвата системных вызовов

Однако в ядрах версии 2.6* для перехвата системных вызовов появился новый механизм, с помощью которого знание адреса таблицы системных вызовов стало вообще ненужным. Данный механизм имеет название LSM (Linux Security Modules)* и позволяет перехватывать системные вызовы, вставляя свои обработчики системных вызовов, без необходимости самому подменять адреса системных вызовов и/или вникать в структуру и последовательность использования функций и ресурсов ядра, т. е. грубо говоря LSM — набор предустановленных в ядре ОС хуков (от англ. hook), которые предоставляют API для внедрения собственных обработчиков непосредственно перед выполнением определенного системного вызова.

Рассмотрим пример LSM-модуля ядра, который будет подменять стандартный системный вызов mkdir своим обработчиком, который дополнительно к созданию каталога будет выводить в dmesg некоторое сообщение, каждый раз когда вызывается команда mkdir:

1. Для того чтобы модуль ядра начал использовать механизм LSM, в его функции инициализации и деинициализации (module_init и module_exit) необходимо добавить вызов следующих 2 функций, соответственно:

```
/**
 * Регистрация перехватчиков системных вызовов
 *
 * @return 0 в случае успешной регистрации, иначе — код ошибки
 */
int hook_register()
{
        int res = register_security(&hook_security_ops);
        if (res) {
             printk(KERN_ERR "failed to register module: %d\n", res);
             return res;
        }
        return 0;
}

/**
 * Дерегистрация перехватчиков системных вызовов
 */
void hook_unregister()
{
        int res = unregister_security(&hook_security_ops);
        if (res)
              printk(KERN_ERR "failed to unregister module: %d\n", res);
}
```

2. Как видно из примера выше LSM-модуль ядра регистрирует собственные обработчики с помощью функций register_security() (дерегистрирует с помощью unregister_security(), обе функции объявлены в <linux/security.h>), передавая в эту функцию структуру вида:

```
/** Операции, перехватывающие системные вызовы */
 static struct security_operations hook_security_ops = {
        .inode_mkdir = inode_mkdir,
};
```

3. В данном случае в структуре типа security_operations перечислены системные вызовы вместе с функциями (нашими хуками), которые выполняются непосредственно перед выполнением указанных системных вызовов (в нашем случае перед системным вызовом inode_mkdir выполняется наша функция inode_mkdir, которая должна быть объявлена в этом же модуле ядра).

4. Сама функция, вызываемая до выполнения системных вызовов может быть, например, такой:

* Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel, 17 aug., 2002.

```
/**
 * Перехват запроса на создание каталога
 */
static int inode_mkdir(struct inode *dir, struct dentry
        *dentry, int mode)
{
  printk ("mkdir hijacked!\n");
  return 0;
}
```

5. В данном случае, загрузив модуль в ядро Linux и выполнив команду mkdir, в выводе dmesg можно будет увидеть новую строчку с фразой "mkdir hijacked!", свидетельствующую о том, что LSM-модуль успешно перехватывает системный вызов inode_mkdir.

Одной особенностью использования механизма LSM является невозможность одновременного использования нескольких модулей ядра для регистрации хуков — при попытке регистрации LSM-модуля ядра будет выведено соответствующее предупреждение, таким образом при использовании стандартного вкомпилированного в большинство версий ядра SELinux использовать собственный модуль безопасности LSM не получится (отключение SELinux с помощью параметра ядра в загрузчике типа selinux=0 также может не принести никакого результата). Поэтому для свободного использования собственного LSM-модуля ядра приходится перекомпилировать ядро Linux, исключая из него любые дополнительные средства защиты (такие, как SELinux, AppArmor, Tomoyo и прочие).

Несмотря на это неудобство механизм LSM решает очень важную задачу — он предоставляет API, который защищает систему от возможной порчи таблицы системных вызовов в случае последовательного изменения адресов одних и тех же системных вызовов двумя и более модулями ядра, а также в случаях наличия ошибок в модуле ядра, производящем такие изменения (или например если модуль ядра при выгрузке не заменяет адреса системного вызова обратно).

Так или иначе, в целях безопасности начиная с версии ядра Linux 2.6.24 и выше механизм LSM (а именно функция, позволяющая использовать данный механизм) перестал экспортироваться ядром ОС Linux (данное обстоятельство связано с тем, что большинство вредоносного ПО скрывало свое присутствие в системе именно с помощью механизма LSM).

Перехват системных вызовов в загружаемых модулях ядра Linux (LKM)

Возвращаясь к обычному перехвату системных вызовов (и отбрасывая проблему возможной порчи таблицы системных вызовов, т. е. фактически не имея других вариантов для корректного перехвата системных вызовов) предстоит дополнительно решить сразу 2 задачи:

- найти адрес таблицы системных вызовов в памяти ядра Linux (так как с версии 2.5.41 этот адрес перестал экспортироваться, а в ядрах 2.6.* память ядра защищена от записи), а если подробнее, то необходимо:
  - найти адрес sys_call_table;
  - получить возможность изменения sys_call_table;
- корректно заменить нужные нам системные вызовы.

Задачу поиска таблицы системных вызовов можно решить сразу несколькими способами:

- найти адрес sys_call_table в файле /boot/System.map, содержащем все используемые ядром символы и их адреса в памяти ядра (такой файл создается при каждой компиляции ядра), например так — `grep sys_call_table/ /boot/System.map | awk '{print \$$1}'` и передать в модуль ядра через Makefile. Таким образом, данный метод фактически не позволяет распространять модуль ядра в собранном виде (в виде бинарного файла с расширением .ko) — его необходимо будет компилировать на каждой новой ОС;
- найти адрес sys_call_table в файле /boot/System.map с помощью функции модуля ядра — в данном случае пересобирать модуль под конкретную систему не обязательно;
- найти адрес sys_call_table в памяти ядра ОС с помощью разбора адресов соседних структур (например "loops_per_jiffy" и "cpu_boot_data"). В данном случае необходимо учитывать, что не во всех версиях ядра ОС таблица системных вызовов будет располагаться именно между указанными структурами;
- найти адрес sys_call_table в памяти ядра ОС, для примера это можно сделать обычным перебором со сравнением определенного элемента таблицы системных вызовов с тем значением, которое должно быть в качестве этого элемента, например так:

```
//значения для 32-разрядных ядер ОС Linux
#define START_MEM    0xc0000000
#define END_MEM      0xd0000000
unsigned long *syscall_table;
unsigned long **find_syscall_table() {
        unsigned long **sctable;
        unsigned long int i = START_MEM;
        while (i < END_MEM) {
            sctable = (unsigned long **)i;
            if (sctable[__NR_close]== (unsigned long *) sys_close)
            return &sctable[0];
            i += sizeof(void *);
        }
        return NULL;
}
…
```

```
//найти таблицу системных вызовов можно следующим образом
syscall_table = (unsigned long *) find_syscall_table();
…
```

Возможности изменения sys_call_table в ядрах 2.6.* и выше по умолчанию не существует. Связано это с тем, что ядро помещает sys_call_table в специальную область памяти "read-only", защищая таким образом ее от намеренного или непреднамеренного изменения (так как это может вести к некорректной работе системы, о чем было сказано выше). В связи с этим необходимо временно (на время внесения изменений в адреса таблицы sys_call_table) переключать режим доступа к таблице системных вызовов, например следующим способом:

```
/* отключить защищенный режим, установив бит WP в 0 */
write_cr0 (read_cr0 () & (~ 0x10000));

/* выполнить изменения таблицы системных вызовов */

…

/*включить защищенный режим, установив бит WP в 1 */

write_cr0 (read_cr0 () | 0x10000);
```

В данном случае блокировка связана с архитектурой используемого процессора, в качестве примера рассматривается блокировка Intel, при которой 0-й бит CR (управляющего регистра, Control Register) необходимо переключать в 0 для отключения "protected mode", а затем в 1 для включения "protected mode" уже после изменения таблицы системных вызовов. Также существуют другие блокировки, зависящие от архитектуры, на которой используется ОС.

Механизм замены системных вызовов собственными функциями можно организовать следующими способами:

- обычной подменой адреса системного вызова адресом своей функции, при этом таблицу системных вызовов необходимо возвращать в исходное состояние после отработки модуля ядра:

```
/* переменная для сохранения адреса оригинальной функции */
asmlinkage int (* orig_mkdir) (struct inode *dir, struct dentry *dentry, int mode);
/* новая функция, заменяющая стандартный системный вызов */
asmlinkage int new_mkdir (struct inode *dir, struct dentry *dentry, int mode) {
        printk ("\nmkdir hijacked!\n");
        return orig_mkdir (dir, dentry, mode);
}
/* функция изменения таблицы системных вызовов */
static void patch_sys_call_table() {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 5, 0)

        write_cr0 (read_cr0 () & (~ 0x10000));
#endif
        orig_mkdir = sys_call_table[__NR_mkdir];
        sys_call_table[__NR_mkdir] = new_mkdir;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 5, 0)
        write_cr0 (read_cr0 () | 0x10000);
#endif
}

/* функция возврата таблицы системных вызовов в начальное состояние */
static void revert_sys_call_table() {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 5, 0)
        write_cr0 (read_cr0 () & (~ 0x10000));
#endif
        sys_call_table[__NR_mkdir] = orig_mkdir;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 5, 0)
        write_cr0 (read_cr0 () | 0x10000);
#endif
}

/* функция инициализации модуля ядра */
static int init(void) {
            ...
/* применяем патч к таблице системных вызовов */
        patch_sys_call_table();
        printk("sys_call_table patched.\n");
        ...
        return 0;
}

/* функция окончания работы модуля ядра */
static void exit(void) {
            ...
/* возвращаем состояние таблицы системных вызовов */
        revert_sys_call_table();
        printk("sys_call_table reverted.\n");
        return;
}
```

- используя механизм LSM, который более не экспортируется, но позволяет:

1. "обезопаситься" от замены системных вызовов двумя разными модулями;
2. приложить меньше усилий для изменения различных структур ядра ОС, так как сам механизм LSM часть блокировок отключает в своих хуках (таким образом функции обработчиков будут содержать меньше лишнего кода).

Выводы

Создание собственной подсистемы разграничения доступа в ОС семейства Linux в техническом плане не является чем-то сложным или недоступным. Получить контроль над системными функциями ядра (системными вызовами), как было показано в данной статье, достаточно просто — поэтому основной задачей является разработка тех проверочных правил, которые должны вызываться при выполнении того или иного системного вызова ядра ОС.

Getting access to system calls in GNU/Linux

A. M. Kanner
Closed Joint Stock Company "OKB SAPR", Moscow, Russia

V. P. Los
Moscow Governmental Industrial University; Institute of Cryptography, Telecommunications and Informatics of the Academy of the Russian FSS Academy

Highlights the problem of access control in GNU/Linux, associated with the features of such OS family.

Keywords: OS GNU/Linux, system call interface, the interception of system calls, LSM.

Каннер Андрей Михайлович, программист группы программирования ядра СЗИ.
E-mail: kanner@okbsapr.ru
Лось Владимир Павлович, профессор, заведующий кафедрой информационной безопасности.
E-mail: los-vladimir@yandex.ru