I. V. Matveychikov

Overview of Dynamic Operating System's Kernel Hooking Methods (Study Case of Linux Kernel)

Keywords: hooking, kernel, Linux

The article presents an overview of dynamic integration in the kernel Linux, allowed to modify (add, change) its functionality. Traditional methods of integration based on changing in the kernel code (patching), and methods based on using system capabilities (for example, LSM) are considered. Special attention is paid to interception key mechanisms of the kernel, such as exception handlers and system call dispatcher.

И. В. Матвейчиков

ОБЗОР МЕТОДОВ ДИНАМИЧЕСКОГО ВСТРАИВАНИЯ В ЯДРО ОПЕРАЦИОННОЙ СИСТЕМЫ (НА ПРИМЕРЕ LINUX)

1. О встраивании

В современном представлении программная система представляет собой совокупность взаимосвязанных программных модулей (компонентов), образующих общую вычислительную систему.

Под встраиванием в программную систему понимается процесс внедрения в нее дополнительных (сторонних) программных элементов, осуществляемый таким образом, чтобы, с одной стороны, сохранялось ее функционирование, а с другой — расширялись или изменялись ее функциональные возможности.

Ядро ОС по праву считается центральной ее частью, так как является промежуточным звеном (посредником) между приложениями и устройствами, осуществляющими обработку данных на аппаратном уровне. При этом основная задача ОС — эффективное управление ресурсами, такими как процессорное время, память и устройства ввода/вывода.

По своей архитектуре ядро Linux представляет собой монолитное ядро с поддержкой возможности расширения функционала за счет модулей, загружаемых (выгружаемых) по необходимости в процессе работы системы. Учитывая данную особенность, для ОС семейства Linux существует возможность разрабатывать различные расширения ядра, которые, фактически являясь частью ядра, могут переопределять/дополнять различные его функции, то есть изменять порядок работы его подсистем.

1.1. Цель и задачи встраивания

Говоря о методах встраивания, будем рассматривать следующую практическую задачу. Пусть есть некоторый целевой компонент программной системы, функционирующий в соответствии с заданным (базовым) алгоритмом. Необходимо осуществить модификацию работы данного компонента таким образом, чтобы иметь возможность вносить конкретные изменения в этот базовый алгоритм.

Таким образом, основной целью встраивания является получение возможности контроля, модификации и расширения функций компонентов программной системы, тогда как основной задачей встраивания является обеспечение внедрения в существующую программную систему при сохранении ее работоспособности.

Успешно выполненное встраивание характеризуется сохранением функционирования программной системы при наличии в ее составе нового компонента.

1.2. О методах и средствах встраивания

Многообразие программных систем характеризуется разнообразием их архитектуры, состава компонентов, а также используемых подходов и технологий. Вследствие этого существует множество различных методов встраивания, каждый из которых более других подходит к применению в конкретной ситуации.

Рассматривая компоненты программных систем в качестве разного рода прикладных и системных программ, выполняющихся на соответствующем оборудовании, можно установить связь между встраиванием и получением возможности перехвата управления в ходе выполнения частей программы.

Согласно устоявшемуся определению, в программировании под термином перехвата понимается набор методов, используемых для изменения или дополнения поведения операционной системы, приложений или других программных компонентов путем перехвата вызовов функций, сообщений или событий, передаваемых между программными компонентами. При этом код, который обрабатывает подобные перехваченные вызовы функций, события или сообщения, называется перехватчиком, или *хуком*.

В контексте данной статьи термины «перехват (управления)» и «встраивание» используются как тождественные, если это не оговаривается специально. Однако следует иметь в виду существующее различие между ними: «встраивание» представляет собой процесс внедрения в общем смысле, тогда как «перехват» скорее указывает на конкретный методический прием.

2. Традиционные методы встраивания

Прежде всего, стоит отметить, что среди методов встраивания существуют методы, предполагающие необходимость модификации загрузочного образа ядра системы, расположенного на жестком диске [1]. И хотя такое встраивание допустимо и, возможно, целесообразно в некоторых ситуациях, в рамках данной работы оно рассматриваться не будет, так как не является «динамическим». Действительно, предполагая необходимость внесения изменений в образ ядра на диске, данные методы требуют перезагрузки системы для того, чтобы эти изменения вступили в силу.

Как правило, в основе большинства традиционных методов встраивания лежит патчинг — техника внесения изменений в код или данные, позволяющая модифицировать поведение целевого алгоритма требуемым образом. Здесь стоит отдельно отметить, что патчинг не ограничивается только модификацией кода, так как на практике вполне возможны ситуации, когда целевой алгоритм может быть изменен требуемым образом только лишь воздействием на используемые в нем переменные, то есть его данные.

Технически результатом патчинга является изменение содержимого ячеек в оперативной памяти. Однако модификация кода, в отличие от модификации данных, имеет свои особенности, связанные, прежде всего, с фундаментальным отличием кода от данных. Код исполняется процессором, и это обстоятельство, в совокупности с особенностями его работы (например, кешированием), накладывает свои ограничения.

По всей видимости, первое упоминание о применении патчинга для работающего ядра Linux принадлежит Сильвио Цезаре (Silvio Cesare) и относится к 1999 г. [2]. Автор статьи описывает метод встраивания, применяемый им для модификации поведения подсистемы учета ресурсов ядра Linux 2.0.35 таким образом, чтобы в ней не учитывалась активность определенного процесса. Для этого в начало системной функции acct_process записывается блок команд, завершающийся командой косвенного безусловного перехода JMP REG, осуществляющего передачу управления на функцию-обработчик.

Тема динамического встраивания в ядро Linux с использованием патчинга получила дальнейшее развитие в работах многих исследователей. Как правило, целью перехвата являются

функции — элементы кода ядра, реализующие тот или иной функционал. Реже объектами перехвата оказываются такие системные механизмы, как обработчики исключений и, в частности, диспетчер системных вызовов. Отмечая также наличие работ, не относящихся непосредственно к ядру Linux, но представляющих определенный интерес при изучении вопросов встраивания [3, 4], стоит отдельно рассмотреть известные методы, используемые при перехвате функций и системных вызовов.

Вообще, говоря о традиционных методах встраивания, можно отметить следующие:

- перехват функций ядра;
- перехват системных вызовов;
- перехват обработчиков исключений;
- использование возможностей системы.

Далее рассмотрим каждый из них отдельно в предположении, что никаких специальных ограничений на реализацию не накладывалось.

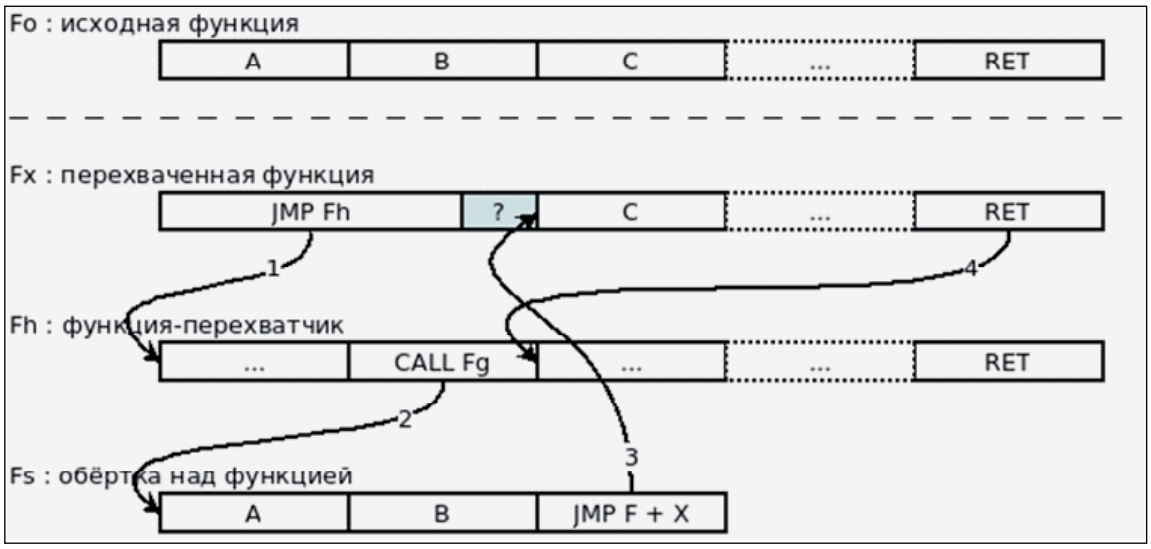

Рис. 1. Иллюстрация типичной схемы перехвата функции

2.1. Перехват функций ядра

Перехват функций ядра является базовым методом, позволяющим переопределять (дополнять) различные его механизмы. Исходя из того, что, за исключением небольших архитектурно-зависимых частей, ядро Linux почти полностью написано на языке C, можно утверждать, что для осуществления встраивания в большинство компонентов ядра достаточно иметь возможность перехвата соответствующих функций, реализующих ту или иную логику [6, 7].

Целью перехвата любой функции является получение управления в момент ее вызова. Дальнейшее поведение зависит от конкретных задач. В одних случаях необходимо заменить системную реализацию алгоритма своей, в других — дополнить. При этом бывает важно оставить возможность использования перехватываемой функции в своих целях. Традиционным стал подход, при котором используется концепция «оберток» [8], позволяющая реализовать пре- и постобработку с сохранением возможности доступа к исходному функционалу, предоставляемому перехватываемой функцией. На рис. 1 приведена иллюстрация типичной схемы организации перехвата функции.

Как видно, в процессе перехвата исходная функция Fo трансформируется в функцию Fx, представляющую собой модифицированную версию первоначальной функции. Суть данной модификации заключается в перезаписи начала исходной функции командой безусловного перехода на функцию-перехватчик — JMP Fh. Ввиду того, что длина команды JMP может быть отлична от длины первой команды функции Fo, есть вероятность, что ее запись перетрет не только

первую, но и последующую команды. Все это необходимо учитывать для того, чтобы правильным образом сформировать трамплин (Fs), позволяющий осуществлять доступ к функциональности, предоставляемой исходной (модифицированной) функцией.

Стоит отметить, что для создания трамплинов необходимо иметь возможность определять количество затираемых инструкций начала целевой функции, которые необходимо скопировать. Для этих целей применяется дизассемблирование. Зная длину блока команд, которые необходимо записать в начало кода функции, можно определить минимальную длину блока команд, которые необходимо сохранить в начале трамплина, и, после того как трамплин создан, можно приступать непосредственно к модификации [10].

Важным моментом при реализации перехватов функций описанными выше способами является необходимость соответствия прототипов и соглашений о вызовах, используемых при описании функций-обработчиков, таковым в целевых функциях. В противном случае может возникать рассогласование между схемами передачи аргументов и использования регистров, что может приводить к нежелательным последствиям. В силу особенностей реализации (использование языка C), применительно к ядру Linux можно ограничиться соглашениями cdecl и fastcall. Таким образом, становится понятным, почему при осуществлении перехвата функций необходимо, как минимум, учитывать используемые в них соглашения. Иначе будет сложно получить доступ к их аргументам (на стеке или в регистрах), а также будет затруднён корректный возврат из обработчиков.

2.2. Перехват системных вызовов

Под термином «системный вызов» (англ. system call) в программировании и вычислительной технике понимается обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции. Ввиду того, что такое взаимодействие является базовым, перехват системных вызовов оказывается одним из важнейших среди методов встраивания, так как позволяет осуществлять контроль ключевого компонента ядра — интерфейса системных вызовов, что, в свою очередь, позволяет осуществлять инспекцию запросов прикладного ПО к сервисам ядра Linux.

Говоря об этом интерфейсе, также оперируют понятием диспетчера системных вызовов, под которым подразумевают часть ядра, осуществляющую диспетчеризацию запросов от ПО к ядру и обратно.

Встраивание в интерфейс системных вызовов может быть осуществлено различными способами, но, прежде всего, стоит отметить, что с точки зрения выполнения этой задачи перехват единичного системного вызова вполне может быть осуществлен с использованием метода перехватов функций ядра, описанного ранее. Действительно, в силу того, что большинство системных вызовов представлены функциями ядра (например, sys_open), задача их перехвата равнозначна задаче перехвата этих функций.

Более универсальным является способ модификации таблиц системных вызовов, содержащих наборы указателей на функции системных вызовов. Эти таблицы используются при диспетчеризации, где по номеру запрошенного системного вызова из таблицы выбирается соответствующий обработчик. При этом стоит учитывать, что в зависимости от конфигурации ядра одновременно может быть несколько таких таблиц: sys_call_table и ia32_sys_call_table. Однако и это не единственное обстоятельство успешной реализации данного метода, ведь эти таблицы еще необходимо отыскать, а также модифицировать при условии, что они не экспортируются и находятся в read-only-области данных ядра. В конечном счете, после выполнения этих операций перехват будет состоять в простом переопределении элемента таблицы.

Другим способом осуществления встраивания является модификация кода диспетчера системных вызовов. Как известно, ядро Linux обеспечивает возможность обращения к интерфейсу

системных вызовов посредством различных команд: INT80, SYSENTER и SYSCALL. Каждая из этих команд, используя аппаратный механизм, обеспечивает вызов соответствующего диспетчера системных вызовов ядра. При этом следует учитывать, что в зависимости от конфигурации ядра управление получают различные функции.

В силу функциональных особенностей структура кода всех диспетчеров одинакова. Как было отмечено ранее, конечной целью диспетчеризации является вызов функции, реализующей логику системного вызова. Опуская некоторые подробности, представим часть кода обработчика интерфейса SYSCALL, реализованного для 64-битного ядра (здесь и далее — debian-2.6.32-5-amd64).

```
system_call:
// точка входа в обработчик
..0ab0: 0f 01 f8            swapgs
..0ab3: ...
// проверка корректности номера системного вызова (Nsys)
..0b2c: 48 3d 2a 01 00 00       cmp $0x12a,%rax
..0b32: 0f 87 ce 00 00 00       ja ffffffff81010c06
..0b38: ...
// диспетчеризация по таблице (call sys_call_table[Nsys])
..0b3b: ff 14 c5 40 82 30 81  callq -0x7ecf7dc0(,%rax,8)
..0b42: ...
```

Листинг 1. Структура кода диспетчера системных вызовов (x86_64)

Как видно, в процессе обработки осуществляется проверка корректности номера системного вызова (Nsys, регистр RAX) с целью пресечения выходов за пределы таблицы системных вызовов. Ключевой точкой всего действия является вызов функции-обработчика системного вызова, реализованный командой вызова подпрограммы (CALL). Именно эта 7-байтовая команда с кодом FF.14.C5 (FF.14.85 для 32-битного ядра) и является целью для последующей модификации.

Один из вариантов модификации — переопределение адреса таблицы sys_call_table, к которому обращается эта команда (константа -0x7ecf7dc0 в примере). Например, можно подготовить копию системной таблицы, модифицировать ее, как необходимо, и после этого внести изменения в код диспетчера. В результате при выполнении данной команды каждый раз будет осуществляться выборка значения указателя функции-обработчика из модифицированной таблицы.

2.3. Перехват обработчиков исключений

Обработка исключений — ключевой процесс функционирования многих системных механизмов. К примеру, обработчик PageFault (#PF) участвует в механизме управления виртуальной памятью и, в частности, страничной подкачки (swap). Перехват обработчиков исключений ядра Linux достаточно просто реализуется с использованием элементов описанных выше техник. В качестве примера приведем иллюстрацию перехвата обработчика исключений, генерируемых командой INT3 (#BP), листинг кода которого представлен далее.

```
int3:
// точка входа в обработчик
..d070: ff 15 ea 1b 19 00  callq *0x191bea(%rip)
..d076: ...
// паразитный «call» (нужно пропускать)
..d07c: e8 3f ff ff ff          callq ffffffff812fcfc0
..d081: ...
```

```
// вызов подпрограммы обработки (do_int3)
..d09e: e8 8d 06 00 00   callq 0xffffffff812fd730
..d0a3: ...
```

Листинг 2. Структура кода диспетчера исключений (на примере int3, x86_64)

Целью модификации в данном случае является вторая по счету 5-байтовая команда CALL с кодом E8. Техника модификации представляет собой простую замену 4 байт данных этой команды, в результате чего изменяется адрес функции, на которую осуществляется переход. Данная техника носит название «сплайсинг» и удобна тем, что осуществляемая операция не затрагивает соседних инструкций.

2.3. Использование возможностей системы

Процесс встраивания в систему нельзя считать рамочным, так как для встраивания может использоваться любой способ, вписывающийся в заданные ограничения. Не исключением здесь является использование штатных механизмов ядра, позволяющих осуществлять требуемые операции.

Зачастую бывает так, что целевая система содержит в себе средства, позволяющие тем или иным образом способствовать решению задач встраивания. В некоторых случаях, если это допускается, имеет смысл использовать такие возможности, ведь это может помочь сэкономить на разработке и тестировании. Кроме того, использование нативных средств встраивания, имеющих достаточно давнюю историю развития, с большей долей вероятности позволит добиться переносимости кода между версиями.

Среди возможных способов встраивания в ядро Linux с использованием возможностей системы стоит отметить способ, основанный на использовании фреймворка Linux Security Modules (LSM) [8], предназначенного для интеграции различных моделей безопасности, служащих целью расширения базовой дискреционной модели безопасности Linux (DAC).

Будучи средством расширения базовой модели безопасности ядра Linux, LSM не является его неотъемлемым компонентом. Наличие в ядре данного фреймворка определяется состоянием конфигурационной переменной CONFIG_SECURITY и задается на этапе компиляции. На практике это означает, что ядро целевой системы может быть собрано без LSM. Тем не менее в остальных случаях использование данной возможности для встраивания представляется в достаточной степени практичным, однако имеет свои особенности, связанные, прежде всего, с архитектурой самой LSM. Действительно, возможности встраивания определяются полнотой покрытия LSM-хуками кода ядра. Иными словами, встраиваться можно только в те места, в которых это предусмотрено.

В соответствии с архитектурой, ключевым объектом ядра Linux, обеспечивающим функционирование LSM-механизма, является указатель security_ops, представляющий собой ячейку памяти ядра, хранящую адрес структуры-описателя действующей (активной) LSM-модели. При этом, в силу того, что все операции, связанные с обращением к модели, осуществляются через данный указатель, можно утверждать, что для встраивания необходимо и достаточно иметь возможность управления этим значением.

Говоря о реализации LSM-модели модулем ядра, стоит отметить, что существуют особенности, связанные с поэтапным введением в ядро Linux ограничений, накладываемых на LSM. Так, из соображений безопасности из ядра последовательно были исключены интерфейсы, позволяющие осуществлять смену действующей модели безопасности в процессе работы системы, а именно:

- начиная с версии v2.6.24, указатель security_ops не экспортируется;
- начиная с версии v2.6.35, функция register_security не экспортируется.

Перечисленные особенности призваны ограничить область применения LSM для встраивания, однако стоит отметить, что на деле они не являются критическими и существует способ их обхода. Данное обстоятельство может быть легко преодолено, если будет найден способ

определения адреса указателя текущей модели — security_ops. Одним из простых и надежных способов, позволяющих определить это значение, является дизассемблирование кода ядра, а именно одной из его функций, где используется данный указатель (например, security_sb_copy_data). Таким образом, получив указатель на указатель security_ops, можно заменить активную модель безопасности на собственную реализацию простой заменой значения этого указателя [9].

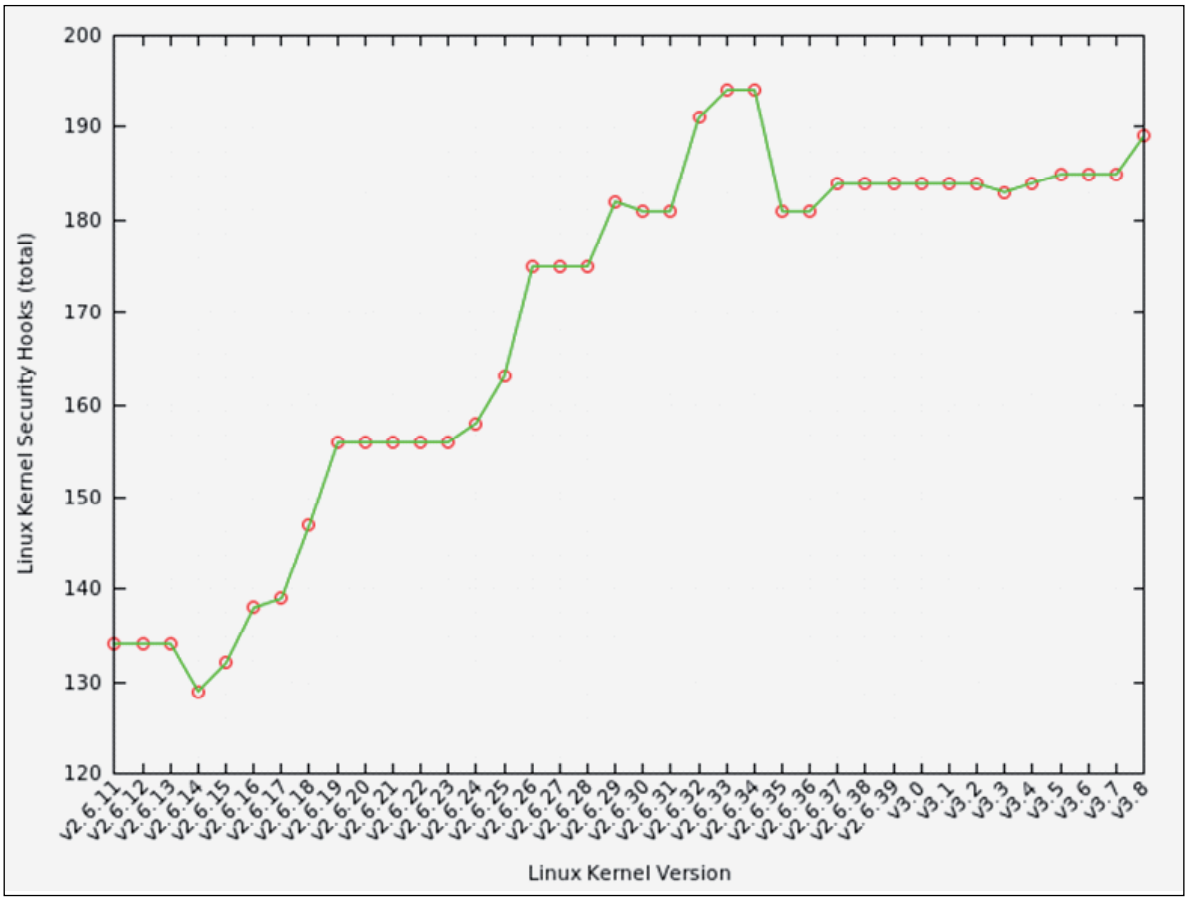

Рис. 2. Количество LSM-хуков в зависимости от версии ядра Linux

Подводя итог рассмотрению LSM, стоит отдельно остановиться на интересной статистике, отражающей динамику развития этого фреймворка в различные периоды времени. Собранная специально для этих целей информация представлена на графике, отражающем общее количество LSM-хуков в зависимости от версии ядра (рис. 2).

Во-первых, обращает на себя внимание их число. Действительно, если на ядро версии 2.6.11 приходилось 134 хука, то на ядро версии 3.8 их приходится уже 189, что является показателем степени глубины интеграции LSM в ядро. Во-вторых, динамика изменения количества хуков свидетельствует о бурном развитии LSM до версии v2.6.34 и некоторой стабилизации после, что является показателем зрелости подсистемы, ее относительной полноты и архитектурной стабильности.

Заключение

Анализ информации, посвященной методам динамического встраивания в ядро Linux, позволяет сформулировать следующие выводы:

- Основой большинства методов встраивания является патчинг — технология модификации кода, описанная в применении к встраиванию в ядро Linux более 10 лет назад и до сих пор не потерявшая своей актуальности.

- Актуальность методов, рассматриваемая с точки зрения их теоретического обоснования, позволяет говорить о *возможной* применимости многих из них на практике, но фактически это требует проведения дополнительных исследований.
- Что касается рассмотренных в процессе составления обзора практических реализаций, то здесь можно констатировать отсутствие работающих вариантов, готовых к «промышленному» применению на целевых ядрах.

Вышеизложенное позволяет сделать вывод об отсутствии общедоступного универсального способа встраивания, который дал бы возможность решать поставленные задачи. Создается парадоксальная ситуация, когда, с одной стороны, существует информация о том, как можно осуществлять встраивание, однако отсутствует конечное решение, которое можно было бы использовать. Все это создает предпосылки для проведения дополнительных исследований и разработки оптимальных методов и средств динамического встраивания с учетом полученных ранее, в большей степени теоретических или устаревших результатов.

СПИСОК ЛИТЕРАТУРЫ:

1. Phrack Magazine. Vol. 60. Issue 0x08. Static Kernel Patching. URL: http://ralf.naegele.net/Security/Phrack/phrack60/p60-0x08.txt (дата обращения: 20.10.2014).
2. *Cesare S.* Kernel Function Hijacking. 1999. URL: http://www.ouah.org/kernel-hijack.txt (дата обращения: 20.10.2014).
3. *Bremer J.* x86 API Hooking Demystified. 2012. URL: http://jbremer.org/x86-api-hooking-demystified/ (дата обращения: 20.10.2014).
4. *Bremer J.* Intercepting System Calls on x86_64 Windows. 2012. URL: http://jbremer.org/intercepting-system-calls-on-x86_64-windows/ (дата обращения: 20.10.2014).
5. Phrack Magazine. Vol. 58. Issue 0x07. Linux on-the-fly kernel patching without LKM. URL: http://ralf.naegele.net/Security/Phrack/phrack58/p58-0x07 (дата обращения: 20.10.2014).
6. Phrack Magazine. Vol. 58. Issue 0x08. IA-32 Advanced Functions Hooking. URL: http://ralf.naegele.net/Security/Phrack/phrack58/p58-0x08 (дата обращения: 20.10.2014).
7. *Fraser T., Badger L., Feldman M.* Hardening COTS software with generic software wrappers // IEEE Symposium on Security and Privacy — S&P. 1999. P. 2–16.
8. Linux Security Modules. URL: http://en.wikipedia.org/wiki/Linux_Security_Modules (дата обращения: 20.10.2014).
9. *Матвейчиков И. В.* Особенности встраивания в ключевые механизмы ядра Linux с использованием LSM. 2013. URL: http://habrahabr.ru/post/196372/ (дата обращения: 20.10.2014).
10. *Матвейчиков И. В.* Перехват функций ядра Linux с использованием исключений (kprobes своими руками). 2014. URL: http://habrahabr.ru/post/206778/ (дата обращения: 20.10.2014).

REFERENCES:

1. Phrack Magazine. Vol. 60. Issue 0x08. Static Kernel Patching. URL: http://ralf.naegele.net/Security/Phrack/phrack60/p60-0x08.txt.
2. *Cesare S.* Kernel Function Hijacking. 1999. URL: http://www.ouah.org/kernel-hijack.txt.
3. *Bremer J.* x86 API Hooking Demystified. 2012. URL: http://jbremer.org/x86-api-hooking-demystified/.
4. *Bremer J.* Intercepting System Calls on x86_64 Windows. 2012. URL: http://jbremer.org/intercepting-system-calls-on-x86_64-windows/.
5. Phrack Magazine. Vol. 58. Issue 0x07. Linux on-the-fly kernel patching without LKM. URL: http://ralf.naegele.net/Security/Phrack/phrack58/p58-0x07.
6. Phrack Magazine. Vol. 58. Issue 0x08. IA-32 Advanced Functions Hooking. URL: http://ralf.naegele.net/Security/Phrack/phrack58/p58-0x08.
7. *Fraser T., Badger L., Feldman M.* Hardening COTS software with generic software wrappers // IEEE Symposium on Security and Privacy — S&P. 1999. P. 2–16.
8. Linux Security Modules. URL: http://en.wikipedia.org/wiki/Linux_Security_Modules.
9. *Matveychikov I. V.* About using LSM as a kernel hooking technique. 2013. URL: http://habrahabr.ru/post/196372/.
10. *Matveychikov I. V.* Hooking Linux kernel functions via the exceptions. 2014. URL: http://habrahabr.ru/post/206778/.