V. A. Sartakov, A. S. Tarasikov

Protection of Microkernel Environment L4Re from Stack-smashed Attacks

Keywords: microkernel, operating systems security, stack protection

Microkernel-based operating systems provide high level of protection due to the strong isolation of components, small size of Trusted Computing Base and execution of drivers in user space. At the same time, such systems are vulnerable to a stack overflow attacks, because these attacks exploit the hardware features of the platform, such as shared memory space for data and code. Modern architectures, such as AMD64 and ARM, provide opportunities to counteract attacks at the hardware level by disallowing memory allocation for storing executable stack and heap, but this protection mechanism requires additional support from the operating system. This paper presents memory management, program execution model and IPC mechanism of microkernel Fiasco.OC and environment L4Re from non-execution memory support point of view.

В. А. Сартаков, А. С. Тарасиков

ЗАЩИТА МИКРОЯДЕРНОГО ОКРУЖЕНИЯ L4RE ОТ АТАК НА СТЕК

Введение

Операционная система (ОС) — фундаментальная часть вычислительного комплекса, на которую ложится основная нагрузка по обеспечению защиты и сохранности пользовательских данных. Ученые и разработчики длительное время трудятся над созданием различных архитектур и систем безопасности ОС. В процессе этой работы был выявлен ряд техник и особенностей ОС, способствующих повышению ее защищенности. К ним относятся уменьшение размера исходного кода, которому необходимо доверять (minimal Trusted Computing Base), высокая степень изоляции компонентов, уменьшение кода, исполняемого в привилегированном режиме. Примером ОС, построенной по этим критериям, является семейство экспериментальных микроядерных операционных систем архитектуры L4.

Заложенные в архитектуру L4 средства изоляции компонентов, минимальный размер исходного кода, вынесение множества компонентов ядра в пространство пользователя позволяют в случае вторжения минимизировать доступные злоумышленнику ресурсы. Например, в монолитно-модульных системах семейства Unix, где эксплуатация уязвимости ядра открывает злоумышленнику путь к использованию всех ресурсов, включая доступ к пользовательским программам, вторжение как в драйвер, так и в любую другую программу микроядерной ОС не дает ему никаких ресурсов, кроме тех, доступ к которым он получил внутри драйвера.

Такие средства защиты предполагают, что вторжение возможно, и не препятствуют ему, а уменьшают влияние скомпрометированного компонента на работу всей системы. Это один из подходов построения модели нарушителя. Исходя из нее, разработчики распределяют защищаемые данные в системе таким образом, чтобы ни один из скомпрометированных модулей не мог повлиять на ее работу в целом или получить доступ к данным.

В то же время среди множества вторжений существует класс атак, в рамках которых злоумышленник не стремится получить доступ к другим компонентам системы, а целенаправленно меняет функции атакуемого модуля. Например, таким модулем может быть программа, обрабатывающая пользовательские данные, которые в случае вторжения могут быть переданы злоумышленнику. Или, например, отображаемые данные могут не соответствовать действительности, так как они заменены вредоносным кодом, внедренным в модуль вывода данных.

Такие атаки изменяют последовательность исполнения программы посредством эксплуатации какой-либо уязвимости. Эти изменения чаще всего связаны с использованием атаки на стек, когда применяемая уязвимость является следствием архитектурных особенностей современных вычислительных систем, в частности — механизмов разделения регионов памяти на память данных и память команд. Вторым отличительным свойством этой уязвимости является то, что описанные выше основные технологии защиты микроядерных ОС не препятствуют вторжению, то есть бесполезны против атак на стек.

Целью данной работы являются анализ класса атак на стек и разработка превентивных средств защиты от них для микроядерного окружения L4Re [1]. Для этого создан набор тестов, различных методов вторжения на основе срыва стека, проведены анализ существующих решений в других ОС, детальный анализ подсистемы управления памятью и запуска программ в микроядре Fiasco. ОС [2] и его окружении L4Re. Это, в конечном итоге, позволит разработать и реализовать архитектуру усиленной микроядерной ОС.

1. Атаки на стек

В основе всех атак, использующих переполнение стека, лежит особенность архитектуры фон Неймана, в соответствии с которой оперативная память (ОП) используется одновременно для памяти данных и памяти команд [3]. Это совместное использование памяти приводит к тому, что загруженные в ОП данные могут быть использованы как команды процессору (благодаря передаче им управления). А к памяти, где располагаются инструкции процессора, могут быть применены операции для работы с данными. Иными словами, регионы памяти, предназначенные для исполнения (память команд), могут быть перезаписаны, а регионы памяти, доступные для записи (память данных), могут быть исполнены.

Рассмотрим «классическую» атаку на стек [4]. При вызове функции в памяти формируется стек, на вершину которого кладется адрес возврата из функции. Далее в нем хранятся локальные переменные. Предположим, одной из этих переменных является статический массив на десять элементов. В теле функции присутствует операция сохранения данных в этот массив, и такая операция не использует проверку длины данных. В процессе сохранения, в первую очередь, происходит перезапись статического массива foo[], затем — других локальных переменных, а в конечном итоге — и адреса возврата. После окончания работы функции процессор использует перезаписанный адрес возврата и передает ему управление. В случае вторжения этот адрес может указывать на размещенный ранее в памяти вредоносный код, например, при сохранении данных в массив. Такая атака называется переполнением или срывом стека.

В рассмотренном примере видны характерные особенности такой атаки. Во-первых, она состоит из двух ключевых элементов. Первый — внедрение вредоносного ПО, часто называемого payload. Это происходит при сохранении входного буфера. Такое ПО может быть внедрено без непосредственного переполнения стека; важно лишь то, чтобы это ПО оказалось в оперативной памяти атакуемой программы. Второй элемент — нарушение последовательности исполнения программы посредством перезаписи адреса возврата. Адрес возврата из стека может указывать на уже загруженный payload, и вместо возврата из функции программа передает ему исполнение. Во-вторых, пример демонстрирует некорректное, с точки зрения безопасности, использование регионов памяти. Регион памяти, предназначенный для данных и доступный для записи, используется как память команд из-за передачи ему управления при вторжении. Аналогичная ситуация со стеком: регион памяти, хранящий адрес возврата и доступный на исполнение, может быть перезаписан.

Это далеко не единственный метод атаки на стек. Помимо отсутствия проверки при сохранении входных данных, рассмотренных выше, существуют другие методы: например, использующие нулевые или недействительные указатели в ядре при некорректной инициализации

callback-структур драйверов. Другими популярными источниками уязвимостей являются ошибки в коде операций с беззнаковыми числами или при миграции некорректного кода из 32- в 64-битную архитектуру. Эти методы различны, но всех их объединяет одно: ошибки в исходном коде дают возможность исполнять его с произвольного адреса в памяти.

2. Средства противодействия атакам на стек

Проблемой защиты стека разработчики и исследователи занимаются более десяти лет. На аппаратном уровне такая защита реализуется посредством эмуляции разделения памяти данных и памяти команд. В популярных архитектурах, таких как ARM и Intel, добавлен аппаратный бит, специальным образом ($W \oplus X$) маркирующий страницы памяти. В AMD64 и ARM эта технология называется «аппаратный NX-бит, No eXecute Bit», в Intel x86_64 — «XD-Bit, Execute Disable Bit». Аппаратная защита требует специальной программной поддержки, реализуемой в операционной системе.

С программной точки зрения, помимо поддержки аппаратного NX-бита, важным средством противодействия атакам на стек является механизм рандомизации адресного пространства (Address space layout randomization, ASLR [5]), смысл которого заключается в произвольном размещении в памяти ключевых элементов пользовательской программы. К таким элементам относятся стек, «куча», исполняемый код и общие библиотеки. Рандомизированное адресное пространство, вкупе с неисполнимым стеком, существенно усложняет вторжение в систему, поскольку, с одной стороны, скрывает размещение компонентов (в частности, адреса возврата из функции), а с другой — не дает исполнить загружаемый в память вредоносный код.

2.1. Операционные системы

В операционной системе Linux существует расширение PaX [6], добавляющее поддержку NX-бита, а также содержащее дополнительные средства защиты системы от вторжения, в частности — ASLR. Кроме того, PaX реализует эмуляцию NX-бита для систем, где его аппаратная поддержка отсутствует. PaX содержит в себе технологию UDEREF [7], позволяющую контролировать исполнение доступных на запись буферов в процессорах Intel i386. Эти процессоры, в отличие от x86_64, содержат разделение памяти по сегментам, которые можно использовать для хранения памяти разного типа. Сегменты кэшируются различными TLB, а при попытке исполнить код из сегмента данных генерируется Page Fault. Он обрабатывается ядром ОС, которое, зная адрес Page Fault, может определить, возможна ли в этом регионе операция исполнения или нет.

Операционная система Windows, в частности Windows 8, поддерживает аппаратный NX-бит и рандомизацию адресного пространства [8]. Подходы к ASLR, применяемые в Windows 8 и PaX Linux, схожи:

- Рандомизация применяется для исполнимых программ и динамических библиотек.
- Адресное пространство пользовательского процесса делится на три части — исполнимую (executable) область, отображаемую (mapped) область и область стека. Исполнимая область содержит код программы и неинициализированные данные. Область отображения содержит «кучу», динамические библиотеки, регионы разделяемой памяти и стек нитей. Область стека содержит только пользовательский стек.
- OS рандомизирует каждый регион по отдельности, в процессе запуска программы добавляя к стартовому адресу каждого региона произвольное значение. Для архитектуры x86 эти значения имеют размерность 16, 16 и 24 бита, для каждого региона соответственно [9].
- Осуществляют поддержку Bottom-up- и Top-down-рандомизации, которая подразумевает расширение в разных направлениях компонентов программы.

3. Усиление микроядра и окружения

Эффективное использование NX-бита и ASLR требует поддержки в ядре и прикладном ПО. Прикладное ПО инициирует процесс запуска программы и определяет, возможно ли для нее использовать рандомизацию, в то время как ядро выделяет ресурсы и контролирует исполнение программы. При запуске программы загрузчик анализирует заголовки исполнимого файла, извлекая из него информацию о возможности использования рандомизированной памяти, а также размеры исполнимой части и ее саму. Далее запрашиваются память и процессорное время у ядра системы, после этого ядро контролирует выделение $W \oplus X$ памяти.

В монолитно-модульных ядрах все компоненты системы, имеющие отношение к управлению памятью, находятся в одном адресном пространстве (контексте ядра). В микроядерных же системах правила управления (policy) памятью вынесены из привилегированного режима и могут находиться в разных компонентах окружения. В этой главе мы рассмотрим управление памятью в микроядре Fiasco.OC и его окружении L4Re. Управление памятью затрагивает не только выделение/освобождение, но и передачу памяти от одного процесса к другому, совместное использование и т. д. Кроме того, в разделе рассматривается механизм порождения процессов в микроядре и окружении, поскольку именно в этот момент выделяется память под стек, данные и код.

3.1. Управление памятью в ядре

Ядро Fiasco.OC предоставляет независимую от аппаратной реализации плоскую модель памяти, для описания страниц которой используется абстракция L4_Fpage. Такие страницы памяти могут быть разных типов, например: Io, Special, Memory, Object. Разные типы страниц по-разному оперируют с флагами доступа и регионами памяти. Например, Io-память, используемая для описания региона устройств ввода-вывода, игнорирует бит свойства исполнения целиком. Для памяти программ используется абстракция Memory, для нее нет специальных ограничений на флаги, и пользовательский процесс может сам контролировать флаги доступа. Как следствие, мы должны доверять этому процессу, и обычно им является Init — первый процесс, стартующий после запуска системы, инициирующий загрузку всех сервисов системы.

Другой важной частью ядра, имеющей отношение к памяти, является абстракция factory. Она позволяет пользовательскому процессу создавать другие абстракции, в частности — передавать память для программ пользователя через протокол dataspace. В этом случае при выделении памяти права доступа определяются битами в capability[1], задаваемых в момент запуска данного процесса.

В соответствии с идеологией L4, в системе присутствует протокол выделения памяти Sigma0, но в Fiasco.OC он используется только для выделения страниц памяти fpage (flex page). Для настройки же флагов кэша и прав доступа применяется отдельный вызов ядра, называемый L4_Fpage.

3.2. Управление памятью в окружении

В пространстве пользователя память описывается и управляется при помощи двух абстракций: dataspace и region mapper. Dataspace описывает блок памяти определенного размера. Для этого блока могут быть заданы права доступа на запись (w), на исполнение (x) или только на чтение (r). Dataspace может как передаваться от одного процесса другому, так и разделяться между двумя процессами в роли общей памяти (shared memory). Region mapper управляет адресным пространством процесса, отображая в его виртуальное пространство определенные физические страницы. Dataspace состоит из набора fpage, для которых должны быть выставлены биты доступа, определяемые аргументами, с которыми был создан dapaspace и вызван region mapper.

На рис. 1 представлен фрагмент кода выделения памяти при помощи вызова *mem_alloc*. Последний аргумент, равный нулю, указывает на запрет выделения исполнимой памяти. Далее выделенная память отображается в пространство процесса при помощи вызова *rm()->attach*, одним из аргументов которого также является запрет на исполнение.

[1] Capability — инструмент разграничения доступа к объектам системы. Является именованной ссылкой на объект.

```
1 L4Re::Env::env()->mem_alloc()->alloc(size, ds, L4Re::Mem_alloc::Executable)
```

```
1 L4Re::Env::env()->rm()->attach(&ptr, size,
2         L4Re::Rm::Search_addr | L4Re::Rm::Eager_map
3         | L4Re::Rm::Executable, ds)
```

Рис. 1. Выделения памяти при помощи вызова mem_alloc

3.3. Загрузка программ

Как уже говорилось ранее, процесс запуска программ является ключевым элементом в защите их от атак на стек, поскольку именно в этот момент формируются регионы памяти для стека, кода и данных программ. Как следствие, именно в этот момент должен быть выделен стек с запретом на исполнение.

Рассмотрим процесс запуска программ в микроядерном окружении. Программа может быть создана любым процессом, обладающим capability на выделение памяти и запуска процесса. На рис. 2 представлена диаграмма вызовов при запуске динамических программ. Все компоненты запуска, за исключением операции выделения памяти, находятся в одном депривилегированном адресном пространстве. Для выполнения операции выделения памяти выполняется IPC-запрос к сервису Moe. Moe — базовый компонент окружения, конфигурацией которого, в отличие от всех остальных, управлять нельзя. Он реализует большую часть низкоуровневых функций, в частности — управление ресурсами, и входит в TCB (Trusted Computing Base), то есть в ту часть кода, которой мы доверяем. Чтобы Moe не выполнял бесконтрольно операции, IPC-запросы к нему проксируются через Ned. Ned также является базовым компонентом системы, занимающейся динамическим управлением настройками сервисов. Capability на выделение памяти и запуск процесса создаются и управляются Ned.

Как видно на рис. 2, для запуска динамических программ используется несколько взаимосвязанных библиотек, таких как libloader, ldso, dso, libc. Важно заметить, что библиотека ldso, являющаяся частью libc, использует POSIX-интерфейс. Поскольку микроядро Fiasc. OC и окружение L4Re не являются POSIX-совместимыми, в системе присутствует уровень совместимости с POSIX, названый L4Re_VFS. Именно этот слой экспортирует malloc, mmap, mprotect и некоторые другие POSIX-вызовы.

С точки зрения защиты стека, запуск программы сводится к выделению dataspace с корректной маской доступа. И поскольку этот dataspace создает Moe, то все функции, имеющие отношение к выделению памяти, начиная от loader и заканчивая IPC к Moe, должны содержать в себе возможность запроса W ⊕ X памяти.

Загрузка статических программ

Загрузка статических программ осуществляется иначе. Поскольку для них нет необходимости в динамическом связывании с другими библиотеками, в процессе запуска отсутствуют lds, dso и другие библиотеки. Весь процесс загрузки сведен к передаче Moe capability на dataspace с программой. Далее Moe выделяет необходимые регионы, загружает содержимое ELF-файла и запускает процесс.

IPC

Взаимодействие программ в операционной системе осуществляется через IPC. Особенностью реализации IPC в окружении L4Re является то, что интерфейс каждого сервиса отделен от его реализации. Как следствие, многие процессы могут предоставлять один и тот же сервис. Выбор конкретной реализации сервиса происходит динамически, на основе capability,

доступных процессу. Контроль доступных capability осуществляется при помощи Ned. Анализ IPC-подсистемы окружения показал, что IPC не влияет на флаги выделенной памяти (например, при передаче dataspace от одного процесса к другому), в то же время отделение интерфейса от реализации приводит к размножению интерфейсов в исходном коде, что негативно сказывается на возможности его анализа и контроля.

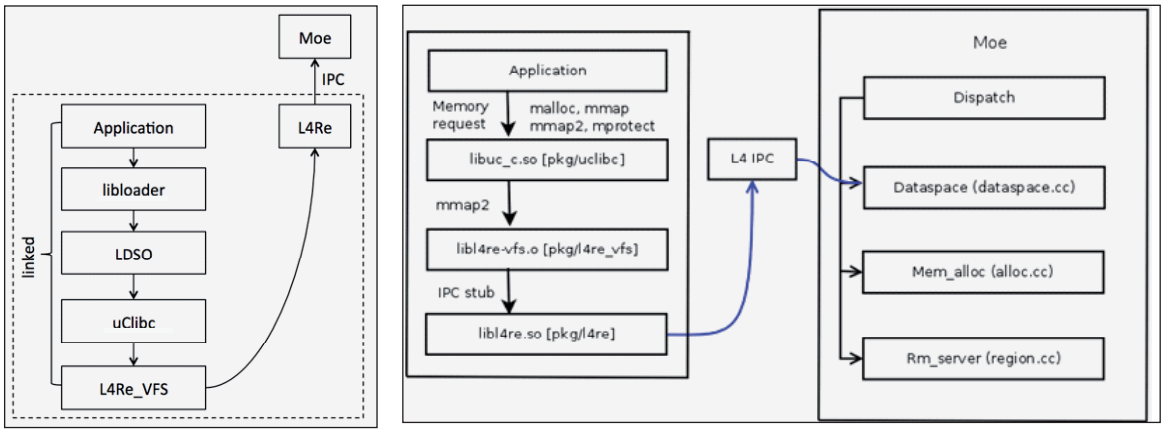

Рис. 2. Запуск программы

4. Реализация и тестирование

Анализ архитектуры микроядерной ОС показал, что поддержка W ⊕ X памяти должна быть реализована на уровне ядра, а также в сервисах окружения, работающих с памятью. Поддержка в ядре сводится к маркированию страниц памяти соответствующими битами доступа. То, какими битами маркировать страницы, определяется при помощи IPC-запроса, сформированного окружением. Задача сервисов окружения — правильно определить тип необходимой памяти и сформировать соответствующий запрос к ядру.

Исходя из этого мы реализовали поддержку NX-бита для классов dataspace и region mapper. Теперь можно выделить новый тип памяти, доступный для записи/чтения, но недоступный для исполнения. Он может быть выделен для любого сервиса, имеющего на это capability. В частности, теперь эта память используется при выделении памяти в процессе запуска новых программ.

Поддержка реализована для платформ ARM и AMD64. Обе используют аппаратный NX-бит. Для платформы i386 поддержки нет, ввиду отсутствия NX-бита в ее архитектуре.

4.1. Тесты на проникновение

Для оценки защищенности был разработан набор тестовых программ. В целях унификации в качестве основы был использован набор тестов PaX, проверяющий стандартные методы вторжения, эксплуатирующие переполнение стека. Портирование тестов PaX в среду L4Re дало возможность запускать одни и те же тесты на разных платформах (таких как Linux, микроядро) и сравнивать их между собой.

Тесты разбиты на две подгруппы. Представители первой подгруппы, такие как *anonmap*, *execbss*, *execdata*, *execheap*, *execstack*, предпринимают попытку внедрения кода в различные части пользовательского процесса. Все они построены по общей схеме: в область памяти помещается инструкция возврата, указатель на начало инструкции приводится к типу указателя на функцию и выполняется ее вызов.

Тесты второй подгруппы, такие как *mprotanon*, *mprotbss*, *mprotdata*, *mprotheap*, *mprotstack*, используют системный вызов *mprotect* для модификации выделенных регионов памяти, в частности для снятия NX-флага. Вызов *mprotect* не реализован в среде L4Re. Вместо него в библиотеке L4Re_VFS есть функция-заглушка, возвращающая код ошибки при запросе памяти

с доступом на запись. В библиотеке ldso реализована функция-обертка mprotect, вызывающая функцию VFS::mprotect. При статическом связывании функция недоступна в среде L4Re.

Ожидаемое поведение тестов первой группы: в системе с поддержкой выделения неисполнимой памяти должна возникнуть ошибка Page Fault, приложение-тест должно принудительно завершиться. Если ошибка не возникла и приложение завершилось корректно с кодом возврата <<0>> и печатью сообщения "Vulnerable", это означает, что для используемого региона памяти не реализована защита от исполнения кода. Ожидаемое поведение тестов второй группы не должно отличаться от поведения первых, не использующих вызов mprotect, в случае статического связывания; и тесты должны завершаться с ошибкой (код возврата, неравный нулю) в случае динамического связывания.

В таблице 1 представлены описания первой подгруппы тестов и результаты их выполнения.

Таблица 1. Тесты на проникновение и результаты их исполнения

Название	Описание	Результат в среде L4Re	Результат в среде L4Re+NX
anonmap	Тест проверяет возможность исполнения кода из анонимной памяти, выделяемой функцией mmap.	Vulnerable	killed
execbss	Тест проверяет возможность исполнения кода из секции .bss файла ELF. Для этого в .bss-секции создается переменная с опкодом инструкции возврата (RETN для X86, "bx lr" для ARM) и проверяется возможность ее вызова.	Vulnerable	killed
execdata	Тестирование возможности исполнения кода из секции .data файла ELF. Для этого создается константа с инструкцией возврата и проверяется возможность ее вызова.	Vulnerable	killed
execstack	Инструкция возврата помещается в область стека программы, и проверяется возможность ее вызова.	Vulnerable	killed
execheap	Тестируется возможность вызова кода из динамически созданного региона памяти.	Vulnerable	killed

Заключение

В рамках работы проведен анализ различных методов атак на стек и разработана усиленная версия микроядерного окружения. Усиленная версия L4Re препятствует изменению последовательности исполнения программы благодаря использованию неисполнимой памяти, а высокая степень изоляции компонентов позволяет минимизировать последствия вторжения для системы. В то же время поддержка операционной системой неисполнимой памяти, безусловно, не гарантирует полную защищенность системы: она лишь повышает ее, отчасти являясь одним из обязательных компонентов современной защищенной ОС [10]. Тем более что неисполнимая память должна использоваться в сочетании с рандомизацией адресного пространства (ASLR) [9].

Реализация рандомизации адресного пространства — одно из направлений дальнейших разработок и исследований. Для использования ASLR в микроядре Fiasco.OC потребуется значительная переработка его внутренней архитектуры, поскольку сейчас все функции, таблицы, ресурсы ядра имеют постоянные, статические адреса.

Другим направлением работы является поддержка неисполнимой памяти в паравиртуализированном L4Linux [11], который является одним из компонентов окружения L4Re, позволяющим выполнять двоичные программы Linux в микроядерном окружении. Он построен на основе исходного кода ядра Linux, в котором аппаратные операции ядра заменены на вызовы L4. В процессе анализа L4Linux было выявлено, что для своей работы L4Linux получает целиком один dataspace, который он самостоятельно использует для выделения памяти программам [12]. Это значит, что в текущей реализации системы паравиртуализации невозможно выделить отдельные неисполнимые фрагменты памяти для процесса. Также в L4Linux отсутствует поддержка ASLR.

СПИСОК ЛИТЕРАТУРЫ:

1. L4 Environment website. URL: http://os.inf.tu-dresden.de/L4Re/ (дата обращения: 15.09.2014).
2. Fiasco website. URL: http://os.inf.tu-dresden.de/fiasco/ (дата обращения: 15.09.2014).
3. *Perla E., Oldani M.* A Guide to Kernel Exploitation: attacking the core. Access Online via Elsevier. 2010.
4. *Etoh H., Yoda K.* Protecting from stack-smashing attacks. 2000.
5. *Whitehouse O.* An analysis of address space layout randomization on Windows Vista // Symantec advanced threat research. 2007. P. 1–14.
6. GrSecurity website. URL: https://grsecurity.net/ (дата обращения: 15.09.2014).
7. *Spengler B.* PaX's UDEREF-Technical Description and Benchmarks. 2007.
8. ASMONIA. D2.1: Evaluating Methods to assure System Integrity and Requirements for Future Protection Concepts.
9. *Shacham H. et al.* On the effectiveness of address-space randomization // Proceedings of the 11th ACM Conference on computer and communications security. ACM. 2004. P. 298–307.
10. *Bickford J., LagarCavilla H. A., Varshavsky A. [et al.]* Security versus energy tradeoffs in host-based mobile malware detection // Proceedings of the 9th international conference on mobile systems, applications, and services. ACM. 2011. P. 225–238.
11. L4Linux website. URL: http://os.inf.tu-dresden.de/L4/LinuxOnL4/ (дата обращения: 15.09.2014).
12. *Lackorzynski A. et al.* L4Linux porting optimizations // Master's thesis, Technische Universitat. Dresden. 2004.

REFERENCES:

1. L4 Environment website. URL: http://os.inf.tu-dresden.de/ L4Re
2. Fiasco website. URL: http://os.inf.tu-dresden.de/fiasco/.
3. *Perla E., Oldani M.* A Guide to Kernel Exploitation: attacking the core. Access Online via Elsevier. 2010.
4. *Etoh H., Yoda K.* Protecting from stack-smashing attacks. 2000.
5. *Whitehouse O.* An analysis of address space layout randomization on Windows Vista // Symantec advanced threat research. 2007. P. 1–14.
6. GrSecurity website. URL: https://grsecurity.net/.
7. *Spengler B.* PaX's UDEREF-Technical Description and Benchmarks. 2007.
8. ASMONIA. D2.1: Evaluating Methods to assure System Integrity and Requirements for Future Protection Concepts.
9. *Shacham H. et al.* On the effectiveness of address-space randomization // Proceedings of the 11th ACM Conference on computer and communications security. ACM. 2004. P. 298–307.
10. *Bickford J., LagarCavilla H. A., Varshavsky A. [et al.]* Security versus energy tradeoffs in host-based mobile malware detection // Proceedings of the 9th international conference on mobile systems, applications, and services. ACM. 2011. P. 225–238.
11. L4Linux website. URL: http://os.inf.tu-dresden.de/L4/LinuxOnL4/.
12. *Lackorzynski A. et al.* L4Linux porting optimizations // Master's thesis, Technische Universitat. Dresden. 2004.