Защита памяти и ядра ROSA Fresh. Практическое руководство.

Калмыков К.В.
Новоселов М.Е.

Декабрь 2023, версия 0.1

Оглавление

Licensing Information

Copyright 2023 by Konstantin Kalmykov (constacalm@yandex.com), Novoselov Michael (m.novosyolov@rosalinux.ru), under The BSD Document License (BDL).

The BSD Documentation License (BDL) terms are:

Redistribution and use in source (Docbook format, LaTeX format, Markdown format or others similar) and "compiled" forms (PDF, PostScript, HTML, RTF, etc), with or without modification, permitted provided that the following conditions are met:

1. Redistributions of source code (Docbook format, LaTeX format, Markdown format or others similar) must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in compiled form (transformed to other DTDs, converted to PDF, PostScript, HTML, RTF, and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this documentation without specific prior written permission.

THIS DOCUMENTATION IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHER-WISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Лицензионная информация

Copyright 2023, Калмыков К.В. (constacalm@yandex.com), Новоселов М.Е. (m.novosyolov@rosa

Разрешается повторное распространение и использование как в виде исходного кода (LaTeX, Markdown и т.п.), так и в двоичной форме (в виде конвертируемого или откомпилированного текста PDF, PostScript, HTML, RTF и т.д.), с изменениями или без, в коммерческих или некоммерческих целях, при соблюдении следующих условий:

1. При повторном распространении исходного кода должно оставаться указанное выше уведомление об авторском праве, этот список условий и последующий отказ от гарантий.

2. При повторном распространении конвертированного или откомпилированного кода должна сохраняться указанная выше информация об авторском праве, этот список условий и последующий отказ от гарантий в документации и/или в других материалах, поставляемых при распространении.

3. Имена авторов не могут быть использованы для или в качестве поддержки или продвижения продуктов, основанных на этой документации (этом произведении) без предварительного письменного разрешения.

ЭТА ДОКУМЕНТАЦИЯ ПРЕДОСТАВЛЕНА БЕСПЛАТНО ВЛАДЕЛЬЦАМИ АВТОРСКИХ ПРАВ И/ИЛИ ДРУГИМИ СТОРОНАМИ "КАК ОНА ЕСТЬ" БЕЗ КАКОГО-ЛИБО ВИДА ГАРАНТИЙ, ВЫРАЖЕННЫХ ЯВНО ИЛИ ПОДРАЗУМЕВАЕМЫХ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ИМИ, ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ КОММЕРЧЕСКОЙ ЦЕННОСТИ И ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЦЕЛИ. НИ В КОЕМ СЛУЧАЕ, ЕСЛИ НЕ ТРЕБУЕТСЯ СООТВЕТСТВУЮЩИМ ЗАКОНОМ, ИЛИ НЕ УСТАНОВЛЕНО В УСТНОЙ ФОРМЕ, НИ ОДИН ВЛАДЕЛЕЦ АВТОРСКИХ ПРАВ И НИ ОДНО ДРУГОЕ ЛИЦО, КОТОРОЕ МОЖЕТ ИЗМЕНЯТЬ И/ИЛИ ПОВТОРНО РАСПРОСТРАНЯТЬ ДОКУМЕНТАЦИЮ, КАК БЫЛО СКАЗАНО ВЫШЕ, НЕ НЕСЁТ ОТВЕТСТВЕННОСТИ, ВКЛЮЧАЯ ЛЮБЫЕ ОБЩИЕ, СЛУЧАЙНЫЕ, СПЕЦИАЛЬНЫЕ ИЛИ ПОСЛЕДОВАВШИЕ УБЫТКИ, В СЛЕДСТВИИ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ДОКУМЕНТАЦИИ (ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ПОТЕРЕЙ ДАННЫХ, ИЛИ ДАННЫМИ, СТАВШИМИ НЕПРАВИЛЬНЫМИ, ИЛИ ПОТЕРЯМИ ПРИНЕСЕННЫМИ ИЗ-ЗА ВАС ИЛИ ТРЕТЬИХ ЛИЦ, ИЛИ ОТКАЗОМ ДОКУМЕНТАЦИИ РАБОТАТЬ СОВМЕСТНО С ДРУГИМИ ПРОИЗВЕДЕНИЯМИ), ДАЖЕ ЕСЛИ ТАКОЙ ВЛАДЕЛЕЦ ИЛИ ДРУГОЕ ЛИЦО БЫЛИ ИЗВЕЩЕНЫ О ВОЗМОЖНОСТИ ТАКИХ УБЫТКОВ.

1 Защита памяти и ядра. Свод требований.

В данном разделе приведены рекомендации по настройке ядра и опций защиты памяти. Критически важно защищать ядро и системную память. Если допустить, что ядро или память скомпрометированы, то нет никаких гарантий в правильной работе любых механизмов безопасности и системы в целом.

Сводная информация с рекомендуемыми настройками для ядра ОС и проекция к Методическому документу ФСТЭК «Рекомендации по обеспечению безопасной настройки операционных систем Linux» от 25 декабря 2022 г. – приведена в таблице Таблица 1:

№ п/п	Параметр и рекомендуемое значение	Интерфейс	См. документ ФСТЭК	См. раздел	Значение по умолчанию
1	`kernel.dmesg_restrict=1`	`/etc/sysctl.conf`	2.4.1	2.3	`0`
2	`kernel.kptr_restrict=2`	`/etc/sysctl.conf`	2.4.2	2.1	`0`
3	`init_on_alloc=1`	`/etc/default/grub`	2.4.3	2.9	Неактивно
4	`slab_nomerge`	`/etc/default/grub`	2.4.4	2.9	Неактивно

5	`iommu=force`
	`iommu.strict=1`
	`iommu.passthrough=0`
		`/etc/default/grub`	2.4.5	2.10	Неактивно

6	`randomize_kstack_offset=1`	`/etc/default/grub`	2.4.6		
7	`mitigations=auto,nosmt`	`/etc/default/grub`	2.4.7	2.4	Неактивно
8	`net.core.bpf_jit_harden=2`	`/etc/sysctl.conf`	2.4.8	2.7	`0`
9	`vsyscall=none`	`/etc/default/grub`	2.5.1	2.11	Неактивно
10	`kernel.perf_event_paranoid=3`[1]	`/etc/sysctl.conf`	2.5.2	Нет	2
11	`debugfs=no-mount`	`/etc/default/grub`	2.5.3	2.16	Неактивно
12	`kernel.kexec_load_disabled=1`	`/etc/sysctl.conf`	2.5.4	2.14	`0`
13	`user.max_user_namespaces=0`	`/etc/sysctl.conf`	2.5.5	2.13	`509894`[2]
14	`kernel.unprivileged_bpf_disabled=1`	`/etc/sysctl.conf`	2.5.6	2.7	2
15	`vm.unprivileged_userfaultfd=0`	`/etc/sysctl.conf`	2.5.7	2.15	`1`
16	`dev.tty.ldisc_autoload=0`	`/etc/sysctl.conf`	2.5.8	2.8	`1`
17	`tsx=off`	`/etc/default/grub`	2.5.9	2.5	Неактивно
18	`vm.mmap_min_addr=4096`	`/etc/sysctl.conf`	2.5.10	2.12	`65536`
19	`kernel.randomize_va_space=2`	`/etc/sysctl.conf`	2.5.11	2.12	`2`
20	`kernel.yama.ptrace_scope=3`	`/etc/sysctl.conf`	2.6.1	2.2	`1`

Таблица 1: *Сводная таблица рекомендуемых настроек*

[1] Для установки переменной в значение **3** требуется специальный патч, который пока отсутствует в ядре ОС ROSA Fresh.

[2] Может отличаться в зависимости от конфигурации оборудования

2 Защита памяти и ядра ОС

В чем опасность

Опасность заключается в возможности получения доступа к данным, обрабатывающимся ядром. Данные, которые обрабатывает ядро носят критическую ценность для ОС и её пользователей. Если они неправомочно доступны, то с их помощью можно реализовать атаки любого вида. По умолчанию в Linux имеются возможности загрузки другого ядра (подмены), или использовании потенциально опасных модулей (драйверов) ядра, в том числе без перезагрузки. Это может привести к обходу механизмов защиты и любой другой опасной активности.

Что можно сделать

Можно активизировать собственные механизмы защиты ядра ОС, направленные на противодействие атакам разного рода, такие как: защита памяти, защита от переполнения буфера, контроль целостности ядра и загружаемых драйверов, ограничить отладку и т.п. Информация о механизмах защиты ядра с пояснениями приведена дальше в этом разделе.

Ядро является основным и наиболее критическим компонентом любого дистрибутива операционной системы Linux, в том числе и с точки зрения выполнения функций безопасности. Большинство их сосредоточено именно в ядре, либо так или иначе выполняется при посредничестве ядра ОС.

Важно знать

Ядро обрабатывает данные программ, управляет страницами памяти и взаимодействием между процессами, управляет файловыми системами, сетью, всей периферией, занимается разграничением доступа, генерирует первичные сообщения аудита, может контролировать целостность программ, управляет жизненным циклом каждого процесса, управляет вводом и выводом, и т.п. Следовательно, во время работы, в ядре обрабатывается или хранится множество важнейшей информации – ключи шифрования, пароли или хеши паролей (аутентификационная информация), защищаемые данные, а также принимаются решения о доступе.

Кроме того, поскольку ядро еще и отслеживает все другие компоненты ОС. Важно, чтобы выполнялось только то ядро, которое является доверенным. Иначе может произойти раскрытие информации, и данные пользователей, а также пароли или ключи шифрования, могут быть скомпрометированы. Кроме того, если не защищать ядро ОС, то нарушитель может попытаться изменить состав модулей (драйверов) ядра. Это может

привести к самым непредсказуемым последствиям. Под угрозой окажутся любые обрабатываемые данные и функции безопасности. Так можно преодолеть защиту, отключив важную подсистему ядра ОС Linux, например механизм аудита, функции разграничения доступа (SELinux), выключить защиту памяти или фильтр пакетов. Любым возможностям несанкционированного взаимодействия с ядром необходимо препятствовать.

Подробная информация об известных техниках эксплуатации уязвимостей в ядре ОС Linux приведена по ссылке:

https://github.com/xairy/linux-kernel-exploitation

Если надо оперативно применить опции ядра, направленные на повышение безопасности, а изучать руководство нет времени, то можно сразу перейти к разделу 2.17. А ознакомиться с руководством и аргументацией можно позднее.

2.1 Защита от просмотра адресов указателей

Переменная ядра ОС, отвечающая за доступ к интерфейсам ядра `/proc/kallsyms` и `/proc/modules` – это `kernel.kptr_restrict`. Просматривая эти файлы, можно получать значения адресации памяти для указателей ядра. То есть можно, например, узнать, на какой адрес в памяти ссылается указатель той или иной программы, или загруженного модуля ядра. Переменная может принимать значения `0`, `1` и `2`. Более подробное описание рисков, связанных с атакой на адреса указателей и прототип эксплойта, демонстрирующий такую возможность, приведены по ссылке:

https://kernsec.org/wiki/index.php/Bug_Classes/Kernel_pointer_leak

Важно знать

Если для переменной `kernel.kptr_restrict` определено значение `0`, то просматривать значения адресов в памяти может любой пользователь ОС. Если задано значение `1`, то просматривать адресацию функций может только `root`. Если значение равно `2`, то никто не получит информацию об адресации. Рекомендуемое значение – два. При значении единица – отображение адресов заменяется на нули для всех пользователей, кроме `root`. При значении два – отображение адресов заменяется на нули для всех пользователей, включая `root`.

Для проверки текущего значения этой переменной выполнить:

Листинг 1: Проверка политики ограничений для /proc/kallsyms

```
# sysctl -a | grep kptr
kernel.kptr_restrict = 2
```

Если значение отличается, то выполнить установку этой переменной:

Листинг 2: Настройка политики ограничений для /proc/kallsyms

```
# sysctl -w kernel.kptr_restrict=2
# echo 'kernel.kptr_restrict = 2' >> /etc/sysctl.conf
```

2.2 Отключение трассировки процессов

На первом этапе нужно проверить, есть ли в выполняющемся ядре LSM модуль YAMA:

Листинг 3: Проверка модуля YAMA

```
# cat /boot/config-5.10.118-generic-2rosa2021.1-x86_64 | grep YAMA
CONFIG_SECURITY_YAMA=y
```

Если есть, то в ОС может использоваться настройка, позволяющая производить ограничения на трассировку процессов. Если нет (выдано значение `CONFIG_SECURITY_YAMA is not set`), то можно пропустить эту настройку.

Для проверки текущих значений трассировки выполнить (от имени администратора `root`):

Листинг 4: Проверка текущей политики трассировки процессов

```
# sysctl -a | grep ptrace
kernel.yama.ptrace_scope = 2
```

Важно знать

Рекомендуется установить запрет трассировки, используя значение `2` для переменной `kernel.yama.ptrace_scope`, или более строгое. Значение `2` определяет, что трассировку процессов может осуществлять только `root`. Значение `3` полностью отключает трассировку. В описанной ниже конфигурации трассировка разрешается только пользователю `root`:

Листинг 5: Запрет трассировки процессов для обычных пользователей

```
# sysctl -w kernel.yama.ptrace_scope=2
# echo "kernel.yama.ptrace_scope = 2" >> /etc/sysctl.conf
```

2.3 Ограничения на просмотр сообщений ядра

Команда `$ dmesg` является очень популярной в ОС Linux. Она выводит на экран сообщения ядра ОС. По умолчанию эту команду может использовать любой пользователь в системе, следовательно её будет применять злоумышленник, чтобы получить информацию о системе и некоторых её характеристиках. Соответственно, любой пользователь ОС сможет или напрямую обратиться к этому файлу (`$ cat /dev/kmsg`), или использовать программы чтения кольцевого буфера аудита ядра, такие как `$ dmesg` или служба `syslog`. Рекомендуется ограничивать пользователей в возможности получать сообщения кольцевого буфера аудита ядра.

Важно знать

Переменная ядра ОС `kernel.dmesg_restrict` отвечает за доступ к интерфейсу кольцевого буфера аудита ядра (файлу `/dev/kmsg`). По умолчанию доступ пользователей к этому интерфейсу не запрещен. Значение `1` предписывает, что обращаться к нему может только `root`. Если установлено значение `0`, то доступ пользователей к буферу аудита ядра не ограничивается.

Проверить текущее значение политики доступа к интерфейсу кольцевого буфера аудита ядра можно так:

Листинг 6: Проверка текущей политики ограничений для `dmesg` и `/dev/kmsg`

```
# sysctl -a | grep dmesg
kernel.dmesg_restrict = 1
```

Если при проверке значение отличается от единицы, то нужно выполнить настройку запрета чтения из этого интерфейса всем, кроме `root`:

Листинг 7: Настройка политики для `dmesg` и `/dev/kmsg`

```
# sysctl -w kernel.dmesg_restrict=1
# echo 'kernel.dmesg_restrict = 1' >> /etc/sysctl.conf
```

2.4 Противодействие Meltdown и Spectre

Использование уязвимостей типа Meltdown или Spectre заключается в возможности несанкционированного выполнения кода в пространстве ядра. Например, возникающих при спекулятивном выполнении инструкций процессора (уязвимости типа Meltdown), и/или связанных с особенностями функционирования модуля прогнозирования ветвлений (уязвимости типа Spectre). Эти уязвимости были обнаружены более пяти лет назад, но все еще продолжают появляться различные их варианты.

Современные процессоры семейства x86 (производства компаний Intel© и AMD©) предоставляют возможность параллельного использования нескольких «нитей» или «потоков» на каждом процессорном ядре. Такая возможность называется SMT (Symmetric Multi-Threading, симметричная многопоточность). Поскольку «нити» или «потоки» (исходя из конструктивных особенностей ЦП) сохраняют возможности обмена информацией между собой, то это потенциально может привести к несанкционированному обмену информацией между процессами, выполняющимися в разных потоках, но на одном ядре ЦП. Либо может привести к нежелательному раскрытию информации в памяти и т.п. Поэтому в защищаемой системе опции процессора, отвечающие за SMT желательно отключить. При этом настоятельно рекомендуется использовать комплексный подход – отключать поддержку SMT как на уровне системы ввода-вывода, так и на уровне операционной системы. Данный подход обеспечивает бóльшую уверенность в том, что уязвимости, связанные с недостатками SMT не будут проэксплуатированы, например в том случае, если производитель оборудования предоставит ошибочное обновление системы ввода-вывода, повторно включающее SMT после отключения.

Уязвимости типа Meltdown и Spectre могут привести к реализации атак, при которых злоумышленник сможет получить доступ к защищенной памяти из программы, не обладающей соответствующими привилегиями (путём анализа данных, записываемых в кэш процессора).

Важно знать

Одним из наиболее эффективных способов противостояния атакам семейств Meltdown и Spectre является отключение SMT (помимо обновлений инструкций самих процессоров и разнообразных патчей к ядру Linux и компиляторам). Для этого ядро ОС Linux имеет нужные параметры, которые позволяют отключить SMT.

С другой стороны, минусом этого решения является снижение производительности. Отключение SMT (любым способом, программным или аппаратным), приведет к тому, что количество виртуальных процессорных ядер (vCPU) уменьшится кратно числу потоков в каждом ядре ЦП (то есть, не менее, чем вдвое). Наличие большого количества ядер vCPU особенно важно при необходимости использовать виртуальные машины.

Но если без SMT все же можно обойтись – то лучше отключить эту функциональность. Желательно планировать покупку оборудования с учетом того, что SMT будет отключаться.

Для проверки того, используется ли технология SMT, требуется от имени любого пользователя выполнить следующую команду:

Листинг 8: Проверка поддержки технологии SMT

```
$ cat /sys/devices/system/cpu/smt/active
0
```

Где `0` свидетельствует об отсутствии поддержки.

Иначе, если вывод `1`, то рекомендуется отключить поддержку SMT, сначала в BIOS/UEFI, если это поддерживается оборудованием. А затем выключить её и в ядре операционной системы.

Для отключения поддержки SMT в ОС, требуется от имени `root` выполнить изменение строки `GRUB_CMDLINE_LINUX` конфигурационного файла `/etc/default/grub`, дописав в ее конец следующие директивы:

Листинг 9: Пример отключения SMT

```
GRUB_CMDLINE_LINUX_DEFAULT=' splash smem=1 mitigations=auto,nosmt'
```

После чего обновить конфигурацию загрузчика и перезагрузить ОС, так как указано в листинге раздела 2.17.

Далее выполнить перезагрузку и перепроверить.

2.5 Отключение технологии Intel© TSX

Важно знать

Технология Intel© TSX применялась в процессорах Intel© до 8-го поколения. Эта технология была направлена на увеличение производительности вычислений, путем попыток предсказания и слияния данных в памяти процессора. Ядро ОС Linux по умолчанию поддерживает эту технологию. Однако эта технология небезопасная, устаревшая. Даже сама компания Intel© её больше не использует, и не рекомендует. Поддержку TSX рекомендуется отключить, если ваш процессор Intel© поколения 8 или ниже.

Для отключения поддержки TSX в ОС, требуется от имени `root` выполнить изменение строки `GRUB\_CMDLINE\_LINUX` конфигурационного файла `/etc/default/grub`, дописав в ее конец следующие директиву:

Листинг 10: Отключение Intel TSX

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 tsx=off'
```

После чего обновить конфигурацию загрузчика и перезагрузить ОС, так как указано в листинге раздела 2.17.

2.6 Защита от переполнения буфера

Важно знать

Обычно в стеке памяти программы содержится критически важный для безопасности данных адрес, который может привести к выполнению произвольного кода. Им является сохраненный адрес возврата, т.е. адрес памяти, по которому выполнение должно продолжиться после завершения выполнения некоей текущей функции. Злоумышленник может перезаписать это значение нужным ему адресом памяти, к которому у него также есть доступ на запись, и в который он помещает свой код. Этот код будет запускаться с полными привилегиями уязвимой программы. Например, он может подставить адрес нужного ему вызова, скажем, POSIX `system()`, оставив аргументы вызова в стеке. Это часто называют эксплойтом возврата в `libc`, поскольку злоумышленник обычно заставляет программу во время возврата перейти к интересующей его процедуре в стандартной библиотеке C (`libc`). Другие важные данные, обычно находящиеся в стеке, включают указатели стека и кадра — два значения, которые указывают смещения для вычисления адресов памяти. Изменение этих значений также может использовать злоумышленник для организации деструктивных воздействий на программу. Чтобы этому противостоять рекомендуется использовать все доступные способы защиты от переполнения буфера.

Защиту от переполнения буфера важно использовать как на уровне «железа», так и на уровне операционной системы. Это важно потому, что исполняемые файлы в системе, могут быть скомпилированы без использования техник защиты от переполнения буфера (например, опции компилятора GCC типа `fstack-protector` не задействовались). Либо некоторые процессоры (особенно старые или дешёвые) могут не иметь соответствующих аппаратных возможностей.

2.6.1 Аппаратная защита от переполнения буфера

Современные процессоры семейства x86 (производства компаний Intel и AMD) предоставляют возможность запрета выполнения кода в некоторых страницах памяти. У процессоров такая возможность активизируется включением специального бита, который у AMD называется No Execute Bit (NX bit), а у Intel - Execute Disable Bit (XD bit). Включение таких битов процессора – это способ минимизации негативных последствий, вызванных переполнением буфера.

Для проверки того, задействованы ли в BIOS или UEFI функции аппаратной защиты от переполнения буфера, требуется от имени любого пользователя (пользователя) выполнить:

Листинг 11: Проверка аппаратной защиты от переполнения буфера.

```
$ journalctl | grep "protection: active"
kernel: NX (Execute Disable) protection: active
```

В том случае, если вывод отличается от приведенного выше, то требуется включить соответствующие опции в BIOS/UEFI и перепроверить.

2.6.2 Программная защита («канарейка»)

Техника программной защиты от переполнения буфера на жаргоне программистов называется «канарейка» (*canary*). Такое название употребляется по аналогии с канарейками, которые использовались шахтерами в угольных шахтах до появления газоанализаторов. Задача «канарейки» умереть прежде шахтера. То есть, в данном случае – прервать выполнение программы до того, как будет осуществлено какое-то деструктивное воздействие. Для этого небольшое целое число (значение которого случайно выбирается при запуске программы) записывается в память непосредственно перед указателем возврата адреса в стеке, и все дальнейшие обращения к нему отслеживаются.

Это хорошо автоматизируемая техника, и она может быть выполнена не программистом, а прямо компилятором во время сборки исходного кода. Естественно, такая концепция защиты немного влияет на производительность программы. Но влияние это не слишком значительное, и лучше немного пренебречь производительностью во имя защиты. Однако эта защита пока никак не настраивается прямо в операционной системе. Она, как уже было сказано, осуществляется на этапе сборки прямо в сборочной среде с помощью специальных опций компилятора GCC типа `fstack-protector`.

Проиллюстрировать поведение этой опции можно на следующем примере. Можно подготовить простую программу на языке Си, откомпилировать с нужной опцией `fstack-protector` и запустить. Так можно убедиться в том, что при сборке программы с такой опцией отслеживается переполнение стека (вследствие контроля к областям оперативной памяти системным вызовом `mprotect()`) и выполнение программы прерывается.

Пример такой проверочной программы на языке Си указан ниже:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct no_chars {
  unsigned int len;
  unsigned int data;
};
int main(int argc, char * argv[])
{
  struct no_chars info = { };
  if (argc < 3) {
    fprintf(stderr, "Usage: %s LENGTH DATA...\n", argv[0]);
    return 1;
```

```
  }
  info.len = atoi(argv[1]);
  memcpy(&info.data, argv[2], info.len);
  return 0;
}
```

Если эту программу сохранить под именем, например, `program1.c`, затем откомпилировать с опцией защиты от переполнения буфера и запустить, то можно убедиться в том, что в результате срабатывания защиты, программа прерывается:

```
$ gcc -O0 -fstack-protector-strong -o ./program1 ./program1.c
$ ./program1 64 AAA
*** stack smashing detected ***: terminated
Aborted
$ echo $?
134
```

Как видно, программа получила шестой сигнал (128 + 6 = 134) `SIGABRT`. Для получения списка сигналов можно выполнить `kill -l` или прочесть справку `signal(7)`.

Кроме того, можно откомпилировать программу и без опции защиты от переполнения буфера и повторно проверить, что будет.

Для проверки опций (в том числе опций защиты от переполнения буфера), с которыми была собрана та или иная программа, существует несколько способов.

Например, можно посмотреть опции сборки, указанные в метаданных RPM пакета с программой. Для этого нужно сделать запрос к менеджеру пакетов `rpm`, например:

```
# rpm -q --queryformat="%{NAME}: %{OPTFLAGS}\n" binutils
```

В ответ система сообщит опции сборки, например для пакета `binutils` (куда входит программа `/bin/ls`):

```
binutils: -O2 -fomit-frame-pointer -gdwarf-4 -Wstrict-aliasing=2 -pipe -Wformat
-Werror=format-security -D_FORTIFY_SOURCE=2 -fPIC -fstack-protector-strong
--param=ssp-buffer-size=4 -m64 -mtune=generic -fstack-protector-strong
```

Как видно из вывода запроса `rpm`, он содержит нужную для защиты от переполнения буфера опцию `-fstack-protector-strong`. Некоторым неудобством этого подхода является то, что нужно знать или уметь выяснять, к какому именно пакету относится та, или иная программа. А если нужно проверить несколько программ в одном каталоге, но которые могут принадлежать разным пакетам, задача проверки может стать совсем мучительной без познаний в программировании на языке оболочки.

Кроме этого способа существует анализатор опций сборки, который представлен в виде сценария `checksec`, доступный по ссылке:

https://github.com/slimm609/checksec.sh

Этот сценарий проверяет содержимое бинарных файлов и библиотек в соответствующих метаданных ELF формата, и выводит опции сборки. Например, можно посмотреть на его вывод для проверки того же бинарного файла `/bin/ls`:

```
$ checksec --output=csv --file=/bin/ls
```

```
$ Partial RELRO,Canary found,NX enabled,PIE enabled,No RPATH,No RUNPATH,No Symbols,Yes,5,17,/bin/ls
```

Как видно из вывода, для программы найдено применение техники защиты от переполнения буфера, т.н. «канарейка»: `Canary found`.

На взгляд авторов, применение этого сценария значительно удобнее, чем получение данных из RPM пакета, так как и результат нагляднее, и проверять можно сразу все файлы в каталоге, и выводить результаты можно еще и в форматах CSV или XML. В любом случае, каждый сам для себя может решить, какой способ проверки защиты от переполнения буфера удобнее использовать.

Естественно, что и само ядро ОС тоже компилируется с опциями защиты от переполнения буфера.

2.7 Защита от выполнения произвольного кода в ядре

Современное ядро Linux содержит фильтр eBPF, который когда то был перенесен из ОС семейства BSD, где выполнял функции фильтра пакетов. А сейчас фильтр eBPF в Linux является механизмом ядра, предоставляющим возможности трассировки и профилирования как самого ядра, так и пользовательских приложений.

Важно знать

Фильтр eBPF имеет встроенный JIT компилятор, и уже давно не является простым фильтром пакетов. Теперь это очень мощная структура в ядре Linux, которая позволяет из непривилегированного пользовательского пространства выполнять произвольный код в ядре, чтобы динамически расширять функциональность ядра. Имеющийся в eBPF JIT компилятор по существу является нарушением принципа «W xor X», поскольку ядро не проверяет код, выполняемый JIT компилятором eBPF.

Несмотря на широчайшие функциональные возможности eBPF, его основным минусом с точки зрения безопасности является то, что этот механизм фактически работает как «песочница» или «виртуальная машина» в пространстве ядра, позволяющая исполнять произвольный код внутри ядра, используя для этого интерфейс из пользовательского пространства. И этот исполняемый код еще и не проверяется на соответствие принципу «W xor X».

Реализация eBPF в ядре Linux пока далека от идеальной. Только за три года (с 2020 по 2022 гг) было зафиксировано[3] более десятка уязвимостей в механизме eBPF, большин-

[3] https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=eBPF&search_type=all&isCpeNameSearch=false

ство из которых имело высокий рейтинг. Были подготовлены и эксплойты, демонстрирующие возможности эксплуатации недостатков в eBPF[4].

Исходя из того, что указано выше, использовать eBPF в защищенной системе не рекомендуется. Ограничить использование eBPF довольно просто. Для проверки того, отключены ли механизмы eBPF, и его JIT компилятор, выполнить:

Листинг 12: Проверка политики ограничения eBPF и его JIT компилятора

```
# sysctl -a | grep unprivileged_bpf_disabled
kernel.unprivileged_bpf_disabled = 1
# sysctl -a | grep net.core.bpf_jit_harden
net.core.bpf_jit_harden = 2
```

Важно знать

Переменная `kernel.unprivileged_bpf_disabled` принимает значения `0` и `1`. Значение `0` разрешает любым пользователям взаимодействие с eBPF. Значение `1` запрещает пользователям (кроме `root`) взаимодействовать с eBPF. Переменная `net.core.bpf_jit_harden` принимает значения `0`, `1` и `2`. Значение `0` не активизирует защиту JIT компилятора eBPF. Значение `1` не позволяет пользователям (кроме `root`) использовать JIT компилятор. Значение `2` никому, включая `root` не позволяет использовать JIT компилятор eBPF. Рекомендуется для переменной `net.core.bpf_jit_harden` устанавливать значение `2`.

Такие настройки призваны полностью противодействовать атакам типа JIT spraying[5]. Для установки рекомендуемых значений выполнить:

Листинг 13: Установка политики ограничения eBPF и его JIT компилятора

```
# sysctl -w 'kernel.unprivileged_bpf_disabled=1'
kernel.unprivileged_bpf_disabled = 1
# echo 'kernel.unprivileged_bpf_disabled = 1' >> /etc/sysctl.conf
# sysctl -w 'net.core.bpf_jit_harden=2'
net.core.bpf_jit_harden = 2
# echo 'net.core.bpf_jit_harden = 2' >> /etc/sysctl.conf
```

2.8 Ограничение дисциплины линии

Исторически ядро Linux автоматически загружает любой подходящий драйвер подключаемой к системе дисциплины линии[6]. Например, если к серверу в COM-порт подключить COM терминал, то его драйвер будет загружен при подключении. Такое пове-

[4] https://haxx.in/files/blasty-vs-ebpf.c

[5] Подробнее о семействе атак JIT spraying можно узнать по ссылкам:
http://www.ruscrypto.org/resource/archive/rc2010/files/10_sintsov.pdf;
https://www.youtube.com/watch?v=LVGw6JFPjhc;
http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

[6] https://en.wikipedia.org/wiki/Line_discipline

дение не может гарантировать безопасность. Более того, ранее уже отмечались уязвимости в драйверах дисциплины линии[7].

Важно знать

Рекомендуется ограничить автоматические возможности ядра по подключению драйверов дисциплины линии там, где это не требуется. Для установки ограничения используется специальная переменная ядра `dev.tty.ldisc_autoload`. Эта переменная может принимать значения `0` и `1`. Значение `0` воспрещает автоматическую загрузку драйвера дисциплины линии.

Для проверки текущего значения этой переменной нужно выполнить:

Листинг 14: Проверка текущей политики дисциплины линии

```
# sysctl -a | grep ldisc
dev.tty.ldisc_autoload = 0
```

Если значение не равно нулю, то установить переменную:

Листинг 15: Установка политики ограничения дисциплины линии

```
# sysctl -w 'dev.tty.ldisc_autoload=0'
# echo 'dev.tty.ldisc_autoload = 0' >> /etc/sysctl.conf
```

2.9 Защита алгоритма оптимизации памяти

Алгоритм оптимизации памяти (т.н. «slab allocator») в ядре ОС базируется на принципе дефрагментации участков памяти в ядре (кэш страниц, данные, представленные древовидными структурами – «куча» (heap)), и направлен на повышение эффективности использования памяти (снижение задержек при работе с данными, и т.п.). Основным методом оптимизации является слияние кэшей или повторное использование одних и тех же данных для однотипных объектов. Были зафиксированы некоторые уязвимости[8], использующие эту особенность алгоритма оптимизации памяти. И даже были подготовлены эксплойты, демонстрирующие атаку на этот алгоритм, см. например:

https://syst3mfailure.io/sixpack-slab-out-of-bounds

https://www.openwall.com/lists/oss-security/2022/01/25/14

Следовательно, необходимо противодействовать возможной активности нарушителя, направленной на организацию атак алгоритма оптимизации памяти в ядре ОС. Для

[7] https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html
[8] См. например CVE-2021-42008 и CVE-2022-0185

этого предлагается использовать опцию загрузки ядра `slab_nomerge`, которая сообщает ядру о том, что слияние кэшей выполнять не нужно, и, таким образом, предотвращается возможность повторного получения данных.

Для отключения слияния кэшей нужно внести в файл `/etc/default/grub`, добавив в строку параметров загрузки ядра следующее:

Листинг 16: Конфигурация запрета слияния кэшей

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 slab_nomerge'
```

После чего обновить конфигурацию загрузчика и перезагрузить ОС, так как указано в листинге раздела 2.17.

Важно знать

Ядро ОС Linux имеет возможность осуществлять очистку памяти. Причем можно указать, в какое время она выполняется – либо перед выделением страницы она будет очищена, либо будет очищена после освобождения, но возможно задать и оба этих варианта.

Это значительно усложняет возможности эксплуатации уязвимостей, основанных на раскрытии информации из страниц памяти, при повторном использовании информации из распределяемых или высвобождаемых страниц, поскольку содержимое этих информационных ресурсов будет обнулено. Подробнее о механизмах очистки памяти, и о потенциально возможных техниках эксплуатации уязвимостей, основанных на доступе к страницам памяти, можно прочесть по ссылке.

Для включения очистки страниц памяти при распределении страниц – нужно внести в файл `/etc/default/grub`, добавив в строку параметров загрузки ядра следующее:

Листинг 17: Конфигурация очистки страниц памяти при их выделении

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 init_on_alloc=1'
```

Для включения очистки страниц памяти при высвобождении страниц – нужно внести в файл `/etc/default/grub`, добавив в строку параметров загрузки ядра следующее:

Листинг 18: Конфигурация очистки страниц памяти при их высвобождении

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 init_on_free=1'
```

Для включения очистки страниц памяти при распределении и при высвобождении страниц – нужно внести в файл `/etc/default/grub`, добавив в строку параметров загрузки ядра следующее:

Листинг 19: Очистка страниц памяти при выделении и высвобождении

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 init_on_alloc=1 init_on_free=1'
```

Нужно понимать, что поскольку ядро ОС при указанных выше настройках будет обязано производить дополнительные операции по принудительной очистке страниц памяти, все операции со страницами виртуальной памяти будут выполняться дольше. Следовательно, это деградирует производительность при любой обработке данных в операционной системе в целом.

Важно знать

Однако, возможен и обратный эффект, при котором усиливается и безопасность, и немного вырастет производительность. Рекомендуется активизировать рандомизацию распределения страниц кэш-памяти, при которой, с одной стороны – будет улучшено среднее использование страниц кэш-памяти с прямым отображением. А с другой стороны – это усложнит возможности провести атаку на страницы кэш-памяти, так как их адресация будет менее предсказуемой....

Подробнее о рандомизации распределения страниц кэш-памяти можно прочесть по ссылке:

https://git.kernel.org/...commit/?id=e900a918b0984ec8f2eb150b8477a47b75d17692.

Для включения динамической рандомизации распределения страниц кэш-памяти, нужно внести в файл `/etc/default/grub`, добавив в строку параметров загрузки ядра следующее:

Листинг 20: Рандомизация распределения страниц кэш-памяти

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 page_alloc.shuffle=1'
```

В ОС есть и другие эффективные способы противодействовать атакам на механизмы памяти ядра. Информация о них приведена в разделе 2.4 настоящего руководства.

2.10 Защита прямого доступа к памяти (DMA)

В том случае, если нарушитель получит физический доступ к аппаратуре, на которой функционирует ОС, то существует опасность внедрения устройства, использующего наличие высокоскоростных портов расширения, которые разрешают прямой доступ к па-

мяти (DMA). Таким образом, устройство[9] нарушителя может получить прямой доступ к памяти ОС. Для противодействия такого рода атакам[10] рекомендуется задействовать механизмы IOMMU, и в этом случае решение о предоставлении доступа к памяти возьмет на себя аппаратный контроллер процессора, который не предоставляет всю память устройствам, работающим под управлением ОС или процессам в пользовательском пространстве ОС.

Для активизации IOMMU выполнить редактирование файла `/etc/default/grub`, и указать в строке загрузки ядра следующие опции:

Листинг 21: Активизация IOMMU

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 iommu=force iommu.strict=1 iommu.passthrough=0'
```

Маханизмы IOMMU напрямую зависят от процессора, поэтому подробнее от опциях IOMMU можно прочесть по ссылке: https://docs.kernel.org/x86/iommu.html.

2.11 Отключение vsyscall

В составе ядра ОС для обратной совместимости сохраняется устаревший механизм обработки и ускорения выполнения для некоторых системных вызовов – `vsyscall`. В современных условиях этот механизм позднее был заменен на механизм vDSO.

Поскольку механизм `vsyscalls` размещал данные в памяти по фиксированным адресам, то это делало его уязвимым к осуществлению типовых «ROP» атак (т.н. атаки возвра́тно-ориенти́рованного программи́рования).

Рекомендуется отключить этот механизм ядра ОС. Для этого в файле конфигурации загрузчика `/etc/default/grub` в строке параметров загрузки ядра нужно задать:

Листинг 22: Отключение устаревшего механизма vsyscall

```
GRUB_CMDLINE_LINUX_DEFAULT='splash smem=1 vsyscall=none'
```

2.12 Настройка изоляции процессов

В страницах памяти виртуального адресного пространства, выделяемых процессам, содержится информация, необходимая им для выполнения и обработки. Поэтому в страницах памяти могут храниться ключи шифрования, защищаемые данные, хеши паролей пользователей, идентификаторы пользователей и файлов, содержимое файлов и т.п.

[9] К устройствам, требующим прямой доступ к памяти (DMA), например, можно отнести устройства FireWire (IEEE 1394), Thunderbolt, USB 4.0 и др. Ограничивать интерфейсы FireWire и Thunderbolt настоятельно рекомендуется, если нет необходимости их использовать.

[10] https://en.wikipedia.org/wiki/DMA_attack

Злоумышленник может получить доступ к данным, хранящимся в страницах памяти, если механизм изоляции процессов настроен неправильно. Поэтому требуется обеспечить невозможность или существенно затруднить злоумышленнику доступ к чужому или предыдущему содержанию страниц памяти. Для этого в составе ядра содержится функция поддержки случайного выделения страниц памяти. Правильная настройка ASLR обеспечивает изоляцию памяти для процессов.

Важно знать

Применение ASLR (совместно с очисткой страниц памяти, которая описана в разделе 2.9) существенным образом затрудняет для злоумышленника возможность эксплуатации уязвимостей, связанных с повторным получением доступа к страницам памяти, или с несанкционированного воздействием на память соседних процессов.

Для настройки ASLR используется переменная ядра `kernel.randomize_va_space`. Эта переменная ядра может принимать разные значения:

- Значение `0`, определяет, что случайного выделения адресного пространства не происходит, и распределение страниц памяти происходит статично;
- Значение `1` определяет консервативную рандомизацию. Однако, данные адресации общих библиотек, стека, `mmap()`, vDSO и «кучи» рандомизированы;
- Значение `2` определяет полную рандомизацию. В дополнение к элементам, перечисленным ранее, управляемая память `brk()` также рандомизирована.

Рекомендуется использовать полную рандомизацию адресного пространства, следовательно, значение переменной `kernel.randomize_va_space` должно быть установлено в `2`.

Для проверки текущего значения переменной ядра функции изоляции процессов необходимо выполнить следующую команду:

Листинг 23: Проверка текущей политики изоляции процессов

```
# sysctl -a | grep kernel.randomize_va_space
kernel.randomize_va_space = 2
```

В ответ система должна сообщить текущее значение параметра ядра в отношении изоляции процессов. В том случае, если выведенное на экран значение изоляции отлично от `2`, требуется произвести настройку ASLR. Для активизации поддержки функции изоляции процессов ASLR и ее настройки на максимальную рандомизацию, нужно выполнить:

Листинг 24: Настройка рандомизации виртуальной памяти

```
# echo "kernel.randomize_va_space = 2" >> /etc/sysctl.conf
# sysctl -w kernel.randomize_va_space=2
```

Увеличить энтропию рандомизации можно с помощью переменных ядра ОС `vm.mmap_rnd_bits=32` и `vm.mmap_rnd_compat_bits=16`[11]. Указанные значения являются зависимыми от архитектуры используемого процессора, но в общем случае хорошо подходят для архитектуры x86. Данные параметры можно внести в конфигурационный файл `/etc/sysctl.conf`:

Листинг 25: Настройка увеличения энтропии при рандомизации

```
] # echo "vm.mmap_rnd_bits = 32" >> /etc/sysctl.conf
# sysctl -w vm.mmap_rnd_bits=32
# echo "vm.mmap_rnd_compat_bits = 16" >> /etc/sysctl.conf
# sysctl -w vm.mmap_rnd_compat_bits=16
```

Для противодействия атакам, связанным с разыменовыванием нулевого указателя рекомендуется использовать переменную ядра `vm.mmap_min_addr`. Аргументом эта переменная принимает адрес в виртуальной памяти, который процессу будет резрешен при выполнении системного вызова `mmap()`. Значение должно быть не меньше, чем `4096`. В ОС она по умолчанию уже установлена в безопасное значение, и обычно не требуется её дополнительная настройка. Однако всегда можно проверить её текущее значение:

Листинг 26: Проверка адресации при вызове `mmap()`

```
# sysctl -a | grep vm.mmap_min_addr vm.mmap_min_addr = 65536
```

Иначе, если значение отличается от `4096` в меньшую сторону, то можно установить безопасное значение для этой переменной:

Листинг 27: Настройка безопасной адресации при вызове `mmap()`

```
# sysctl -w 'vm.mmap_min_addr=4096'
# echo "vm.mmap_min_addr = 4096" >> /etc/sysctl.conf
```

2.13 Настройка пользовательских пространств имен

Пользовательские пространства имен `user_namespaces(7)` – механизм ядра ОС Linux, предназначенный для изолированных пространств, и обычно применяющийся в интересах изоляции контейнеров, но не только. Использование этого механизма доступно обычным пользователям.

[11] По умолчанию в ОС используется значение: `vm.mmap_rnd_compat_bits=8`

Важно знать

Проблема разрешения применения `user_namespaces(7)` для обычных пользователей с точки зрения безопасности состоит в том, что такое разрешение открывает доступ для пользователей к коду ядра, который в обычном случае (без этого разрешения) доступен только пользователю **root**. Например, разрешая `user_namespaces(7)`, пользователи получают доступ к подсистеме ядра, отвечающей за обработку протокола L2TP, и, как следствие, смогут создавать свои туннели. Кроме того, это же разрешение открывает пользователям доступ к управлению программным коммутатором OpenVSwitch, в случае его использования, управлению беспроводными сетями (NL80211), к монтированию ФС, и т.п. То есть применение этого механизма серьезно увеличивает площадь атаки на ядро ОС.

Более подробно информацию можно изучить по ссылкам:

https://unix.stackexchange.com/questions/303213/how-to-enable-user-namespaces-in-the-kernel-for-unprivileged-unshare

https://grsecurity.net/10_years_of_linux_security.pdf

https://lwn.net/Articles/652468/

Более того, реализация этого механизма в ОС Linux пока еще далека от идеальной. Например, только с 2020 года, и по настоящее время в механизме `user_namespaces(7)` сообщалось не менее чем о десяти опасных уязвимостей, к некоторым из которых были подготовлены эксплойты. Поэтому рекомендуется отключить `user_namespaces(7)` если rootless контейнеры применять не планируется.

Более подробный список известных уязвимостей в этом механизме приведен по ссылке:

https://security.stackexchange.com/questions/209529/what-does-enabling-kernel-unprivileged-userns-clone-do#209533

Кроме того, механизм `user_namespaces(7)` используется при выполнении приложений,использующих WebKit (WebKit GTK). И такие приложения как браузеры Google© Chrome, Chromium, Konqueror или почтовый клиент Gnome Evolution могут испытывать проблемы с запуском и производительностью, или не запускаться вовсе.

Для отключения `user_namespaces(7)` используется соответствующие переменные ядра ОС – `user.max_user_namespaces` и `kernel.unprivileged_userns_clone`, которые определяют возможность создания пользовательских пространств имен.

Для проверки текущего значения этой переменной нужно выполнить:

Листинг 28: Проверка политики пользовательских пространств имен

```
# sysctl -a | grep user.max_user_namespaces
user.max_user_namespaces = 0
```

Если значение не равно нулю, то для отключения пользовательских пространств имен можно установить переменную:

Листинг 29: Установка политики ограничения `user_namespaces(7)`

```
# sysctl -w 'user.max_user_namespaces=0'
# echo 'user.max_user_namespaces = 0' >> /etc/sysctl.conf
# sysctl -w 'kernel.unprivileged_userns_clone=0'
# echo 'kernel.unprivileged_userns_clone = 0' >> /etc/sysctl.conf
```

2.14 Технология защиты ядра Lockdown

В ядре Linux начиная с версии 5.4 появилась поддержка специальной технологии защиты ядра под названием Lockdown.

Эта технология нужна для ограничения воздействия на выполняющееся ядро даже со стороны `root`. Логика здесь в том, что если злоумышленник все же добился прав `root`, то нужно препятствовать его попыткам загрузить другое ядро или прочесть важные данные из памяти ядра (например, ключи шифрования и т.п.).

Однако стоить помнить, что в таком случае[12] невозможен переход в сон (режим гибернации), а это может быть важно при работе на ноутбуке. А также ограничивается доступ для `root` к довольно большому количеству интерфейсов ядра: `/dev/mem` (`/dev/kmem`), `/dev/port`, `/proc/kcore`, `debugfs`, отладочному режиму `kprobes`, `mmiotrace`, `tracefs`, BPF, некоторым интерфейсам ACPI и MSR-регистрам процессора, блокируется использование системных вызовов `kexec_file()` и `kexec_load()`, не допускаются манипуляции с портами ввода/вывода, в том числе изменение номера прерывания и порта ввода/вывода для последовательного порта, а также блокируются некоторые другие интерфейсы ядра, используемые реже. Одним словом, для разработки такой режим не годится.

Поскольку в ОС ROSA Fresh 12 используется ядро версии не ниже 5.10, то защиту ядра Lockdown можно активизировать. По умолчанию она выключена[13]. У технологии Lockdown два возможных режима работы. Менее строгий, и более строгий.

Первый режим называется `integrity`, и препятствует воздействию на работающее ядро как со стороны пользовательского пространства, так и со стороны `root`. Проще говоря, нельзя будет выполнить загрузку другого ядра с помощью `kexec_load`,

[12] Имеется ввиду режим максимальной защиты `confidentiality`.

[13] Если установить систему с поддержкой UEFI Secure Boot, то Lockdown может включится в режиме `integrity`.

`kexec_file_load`[14], или сбросить дамп ядра с помощью `kdump`. Интерфейсы отладки и возможности со стороны администратора **root** получать данные из ядра в этом режиме сохраняются.

Второй вариант, более строгий, называется `confidentiality`. Помимо того, что нельзя манипулировать с работающим ядром, все отладочные и некоторые другие интерфейсы ядра блокируются даже для администратора **root**, как было описано выше.

Проверить, возможно ли включение технологии Lockdown на текущем ядре, можно просмотрев конфигурационный файл загруженного ядра, например:

Листинг 30: Проверка наличия Lockdown в текущем ядре

```
] # cat /boot/config-5.10.118-generic-2rosa2021.1-x86_64 | grep LOCKDOWN
CONFIG_SECURITY_LOCKDOWN_LSM=y
```

Если поддержки в ядре нет, то есть в ответ возвращается:

`CONFIG_SECURITY_LOCKDOWN_LSM is not set`,

это значит, что включить эту технологию не получится без пересборки ядра с нужными опциями.

Если ядро поддерживает включение Lockdown, то нужно загрузить ядро с соответствующими опциями. Для этого в файле `/etc/default/grub` в строке параметров загрузки ядра нужно задать:

Листинг 31: Включение Lockdown в режим confidentiality

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash lockdown=confidentiality"
```

После чего обновить конфигурацию загрузчика и перезагрузить ОС.

Текущий режим работы Lockdown указывается в квадратных скобках, например:

```
# cat /sys/kernel/security/lockdown
none integrity [confidentiality]
```

При включенном Lockdown в режиме конфиденциальности можно убедиться в том, что даже **root** лишился возможности запрашивать данные из интерфейсов ядра, обычно доступных по умолчанию, например:

```
# cat /dev/mem
cat: /dev/mem: Operation not permitted
# cat /dev/port
cat: /dev/port: Operation not permitted
# cat /proc/kcore
cat: /proc/kcore: Operation not permitted
```

[14] Входят в состав пакета **kexec-tools**.

2.15 Ограничения для процессов при обработке ошибок

В случае по умолчанию ядро ОС может делегировать пользовательским процессам обработку различных ошибок в страницах памяти, что в противном случае мог бы сделать только код ядра, выполняющийся в изолированной области. Поведение ядра по умолчанию делает возможным атаки на «кучу» (heap). Подробная информация о таких атаках и концепция эксплойта приведена по ссылке:

https://duasynt.com/blog/linux-kernel-heap-spray

В данном случае, речь идет о типовых атаках типа «use-after-free» (использование после высвобождения). При этом эксплуатируются возможности системного вызова `userfaultfd(2)`[15], который создает файл (и даже от имени непривилегированного пользователя), в который попадает содержимое страниц памяти при ошибках.

Рекомендуется противодействовать получению информации с помощью `userfaultd(2)`. Для этого нужно переключить переменную ядра ОС `vm.unprivileged_userfaultfd` в значение `0`.

Для проверки текущего значения этой переменной нужно выполнить:

Листинг 32: Проверка политики ограничения `userfaultd(2)`

```
# sysctl -a | grep userfaultd
```

Если значение не равно нулю, или вывода нет вовсе, то установить переменную:

Листинг 33: Установка политики ограничения `userfaultd(2)`

```
# sysctl -w vm.unprivileged_userfaultfd=0
# echo 'vm.unprivileged_userfaultfd = 0' >> /etc/sysctl.conf
```

2.16 Отключение служебной ФС ядра debugfs

Служебная файловая система ядра `debugfs` предназначена для упрощения отладки и монтируется по умолчанию в оперативную память при загрузке ядра. Поскольку к ней предоставляются права чтение, нарушитель может использовать данные, предоставляемые `debugfs` для изучения системы. Также эта файловая система предоставляет функции, способствующие отладке приложений. Подробнее информацию о `debugfs`

[15] Требуется учитывать, что системный вызов `userfaultd(2)` используется гипервизором KVM для реализации динамической миграции с последующим копированием. Динамическая миграция с посткопированием — это одна из форм расширения памяти, состоящая в том, что виртуальная машина работает с частью или всей памятью, расположенной на другом узле в облаке. Если у вас используется виртуализация на нескольких узлах вместе с NUMA, то ограничение использования этого системного вызова приведет к невозможности миграции виртуальных машин.

можно получить по ссылке:

https://lkml.org/lkml/2020/7/16/122

2.17 Рекомендуемые опции строки загрузки ядра

Таким образом, если подытожить все опции загрузки, направленные на безопасность и описанные в данном руководстве (в том числе в разделах 2.1 – 2.14), то можно рекомендовать следующее значение в `/etc/default/grub`:

Листинг 34: Использование безопасных опций загрузки ядра

```
GRUB_CMDLINE_LINUX_DEFAULT="spectre_v2=on spec_store_bypass_disable=on tsx=off tsx_async_-
abort=full,nosmt mds=full,nosmt l1tf=full,force nosmt=force kvm.nx_huge_pages=force slab_nomerge
init_on_alloc=1 init_on_free=1 page_alloc.shuffle=1 pti=on vsyscall=none debugfs=off oops=panic
module.sig_enforce=1 lockdown=confidentiality mce=0 quiet loglevel=0 audit=1 audit_backlog_limit=8192
ipv6.disable=1 selinux=1 lsm=selinux,yama,selinux,integrity,landlock intel_iommu=on iommu.strict=1
iommu.passthrough=0 efi=disable_early_pci_dma randomize_kstack_offset=1 quiet splash"
```

Затем нужно не забыть обновить конфигурацию загрузчика командой `update-grub2` от имени `root` и перезагрузить ОС, для вступления изменений в силу:

Листинг 35: Команды обновления параметров загрузчика и перезагрузки

```
# update-grub2
# reboot
```

Список листингов

Список таблиц