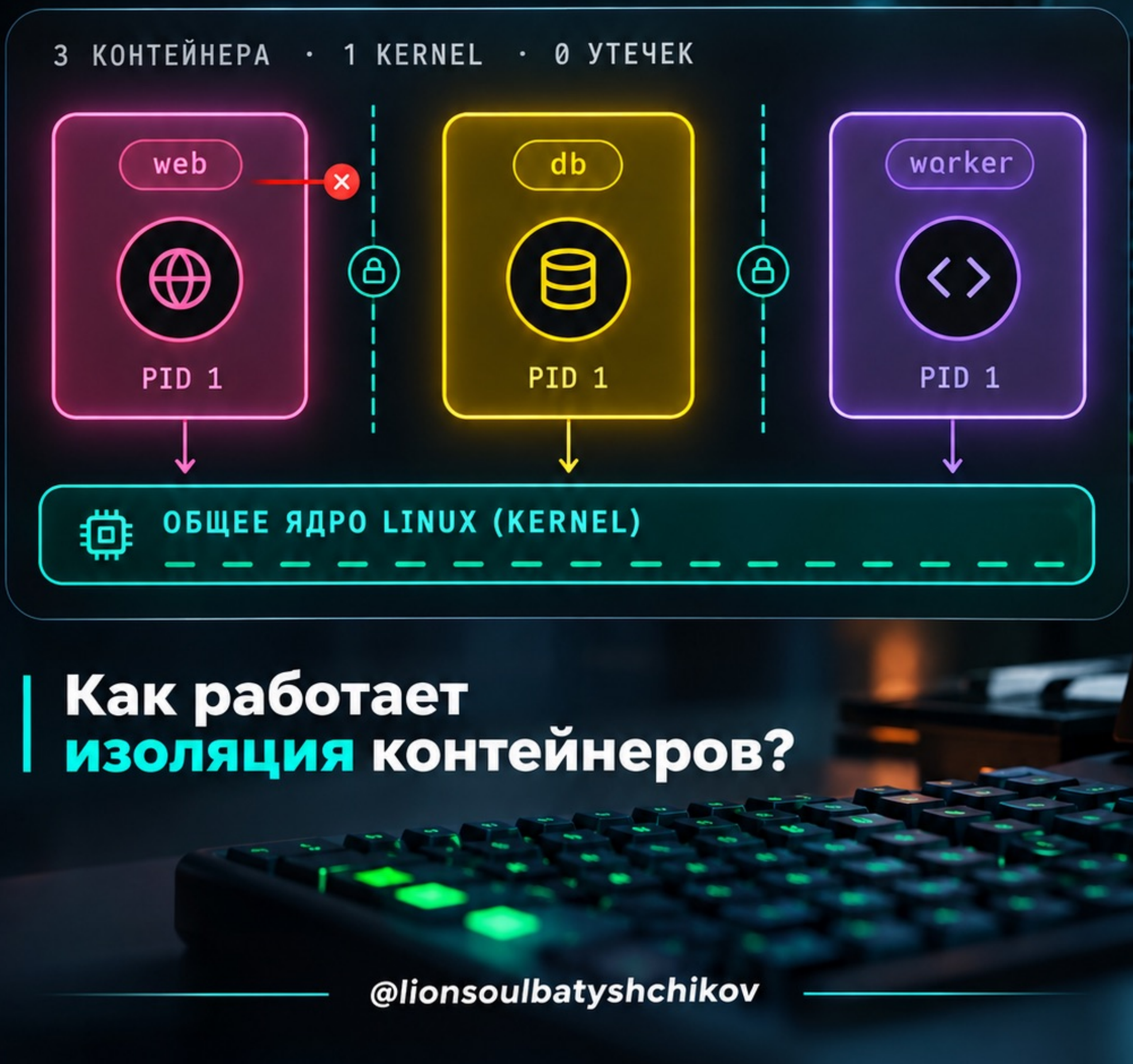
ВИЗУАЛЬНЫЙ ГАЙД ДЛЯ НОВИЧКОВ

Ваши контейнеры делят **одно ядро** Linux. Но как они остаются **изолированными**?

Заметка:

Один Kernel, несколько изолированных пространств. Как это работает?



Как работает **изоляция** контейнеров?

@lionsoulbatyshchikov

01 • ОСНОВЫ

Что такое контейнер?

Контейнер — это не отдельный компьютер, а обычный процесс с изоляцией, которую обеспечивает **Kernel.**

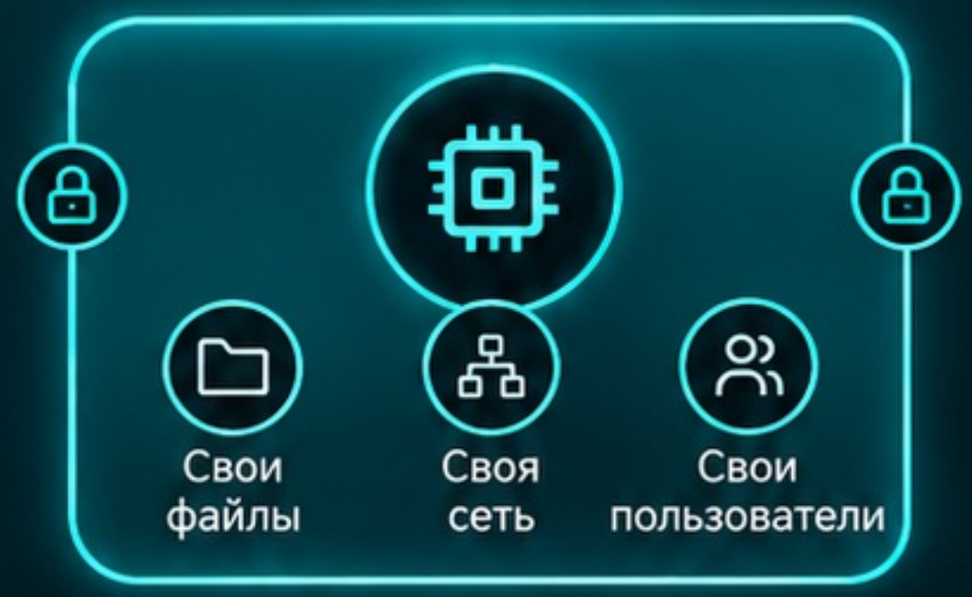


ОБЫЧНЫЙ ПРОЦЕСС



Видит и может взаимодействовать со всей системой

КОНТЕЙНЕР



Видит только своё изолированное окружение

Контейнер =
приложение + его зависимости, упакованные в единый **Image**, который запускается как процесс с изоляцией и ограничениями от **Kernel**.



Упаковано (Packaged)
Приложение и зависимости идут вместе в одном **Image.**

Изолировано (Isolated)
Свой процесс, файловая система, сеть и пользователи.

Ограничено (Limited)
Контроль над ресурсами: CPU, память, количество процессов.

02 • СРАВНЕНИЕ

Контейнеры vs Виртуальные машины (VM)

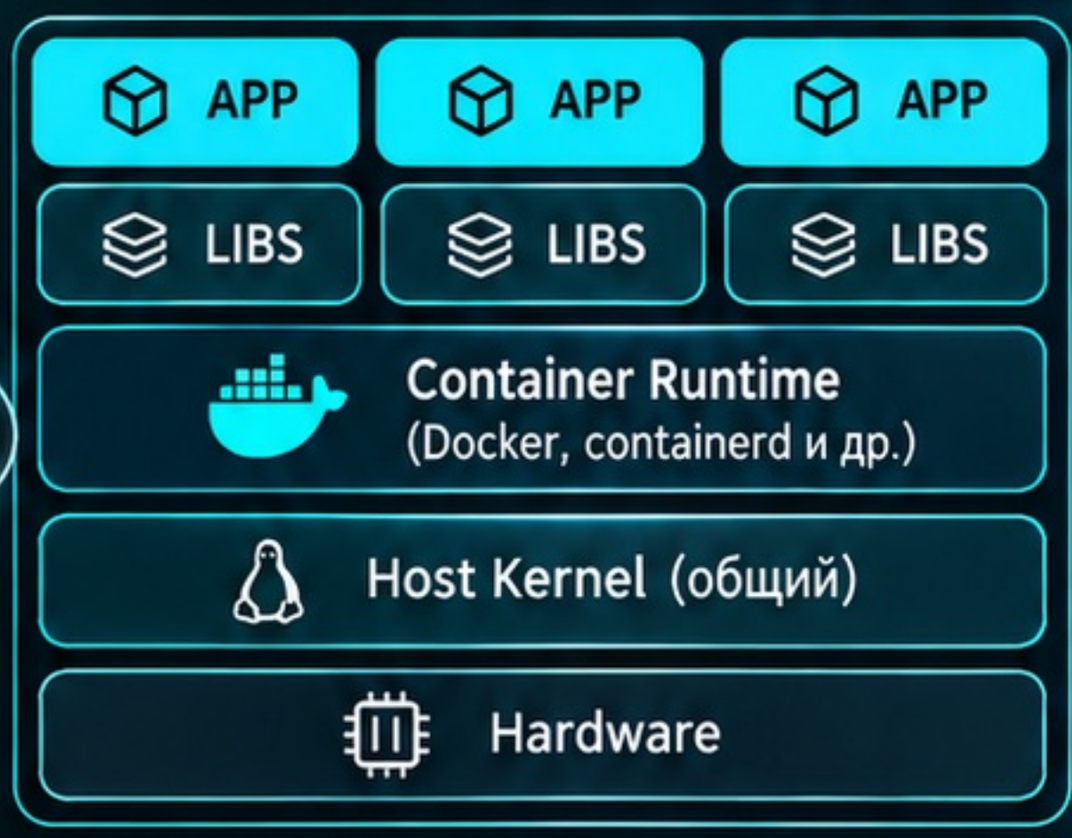
Виртуальная машина (VM)



Каждая VM загружает свой собственный **Kernel**

VS

Контейнер



Все контейнеры используют один общий **Kernel**

Параметр	VM	Контейнер
Запуск (**Boot**)	Минуты	Миллисекунды
Размер (**Size**)	Гигабайты	Мегабайты
Kernel	Собственный	Общий (shared)
Изоляция (**Isolation**)	Сильнее	Хорошая, но не абсолютная

Контейнеры жертвуют частью изоляции ради **высокой скорости** и эффективности.
Задача — понимать эти компромиссы и правильно их **защищать**.

03 • ГЛАВНЫЙ СЕКРЕТ

Это всё тот же процесс

Запустите веб-сервер в контейнере и посмотрите на него с двух сторон — изнутри и на хосте.



Внутри контейнера

nginx container

```
root@web:~$ hostname
web
root@web:~$ ps -e
  PID   CMD
    1   nginx
    7   nginx worker
    8   nginx worker
root@web:~$ ls /
app  bin  etc  lib  usr
```

На хосте (Host)

root@lab

```
root@lab:~$ hostname
lab
root@lab:~$ pgrep -a nginx
48213 nginx
48220 nginx worker
48221 nginx worker
root@lab:~$ grep NSpid \
    /proc/48213/status
NSpid:    48213    1
```

PID 1 внутри = PID 48213 на хосте

*Один и тот же процесс. Два взгляда. Один **Kernel**.*

Попробуйте сами

```
root@lab:~$ grep NSpid /proc/<pid>/status
```

В одной строке вы увидите два PID: PID процесса на хосте и тот, который он видит внутри контейнера.

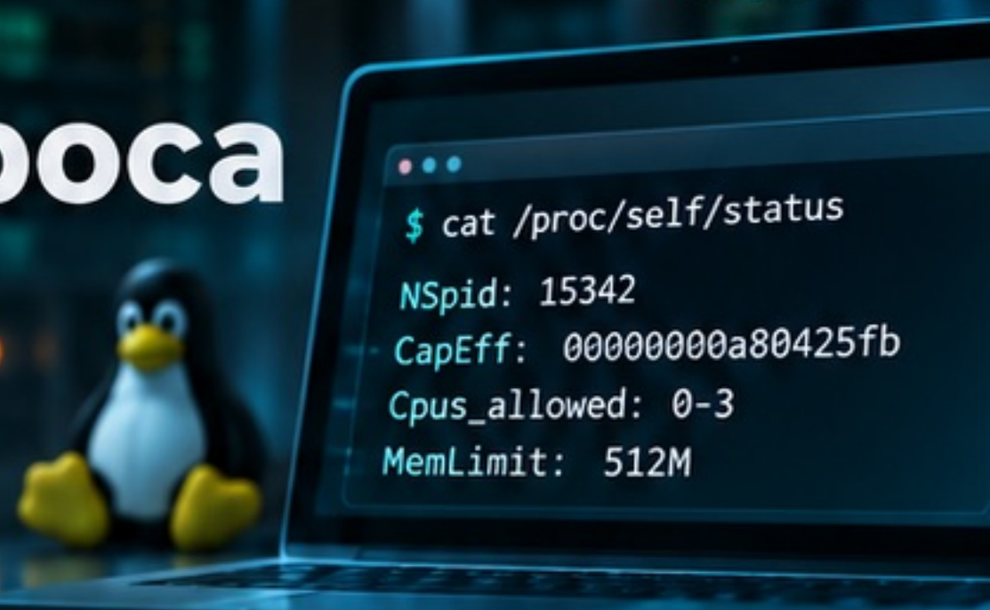
Важно:

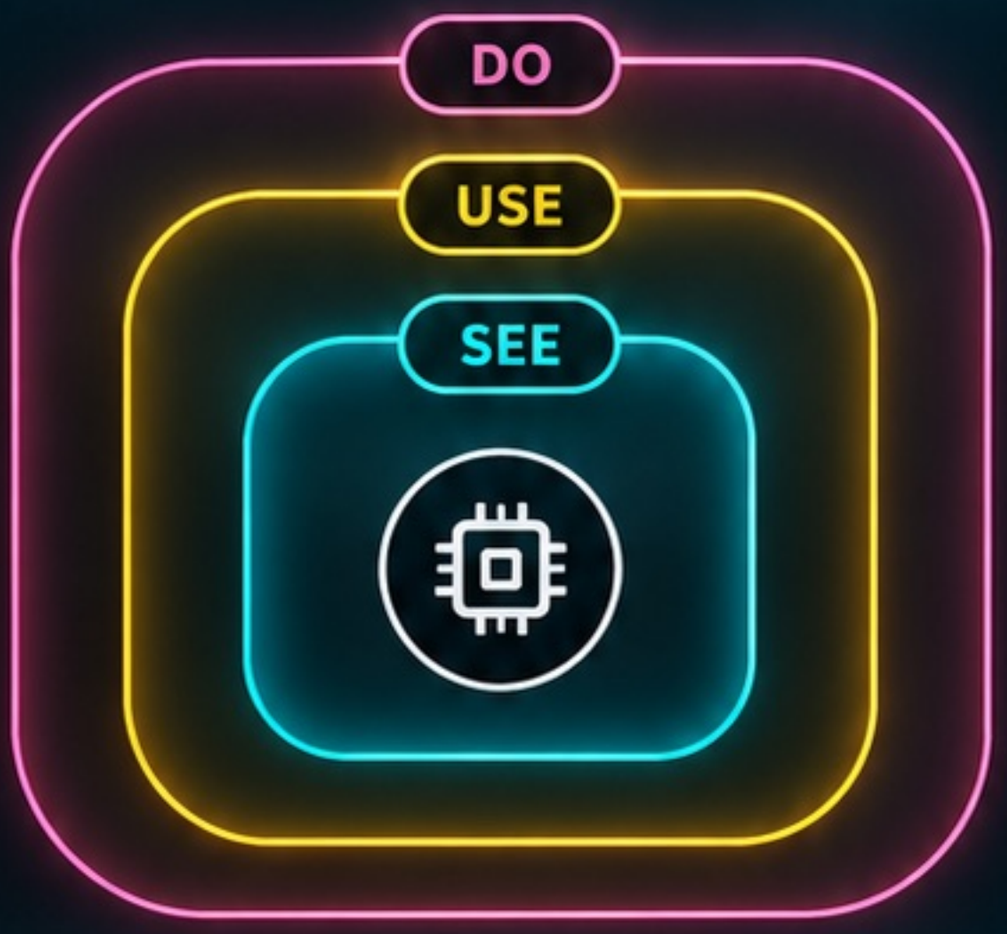
Kernel сохраняет оба идентификатора процесса. Контейнер видит только свой внутренний PID.

04 • КАРТА ИЗОЛЯЦИИ

Три ключевых вопроса про изоляцию

Каждый механизм изоляции в Linux отвечает на один из трёх вопросов о процессе.





DO **Возможности**
Capabilities · seccomp · LSM
Какие привилегированные действия разрешены.

USE **Ресурсы**
cgroups
Сколько CPU, памяти и других ресурсов можно использовать.

SEE **Видимость**
Namespaces
Что процесс «видит»: процессы, ФС, сеть, hostname и другое окружение.

Что он видит? (Namespaces)

Изолируют представление о системе: своя таблица процессов, файловая система, сеть, hostname и т.д.

Чем он может пользоваться? (cgroups)

Ограничивают и контролируют потребление ресурсов: CPU, память, количество процессов и т.д.

Что ему разрешено делать? (Capabilities · seccomp · LSM)

Определяют набор привилегий и фильтруют системные вызовы, разрешая только необходимые операции.

05 · ЧТО ОН ВИДИТ

Namespaces: 8 стен изоляции



Namespace «упаковывает» общий ресурс системы, давая процессу **собственное, изолированное** представление.

Процессы и PID `pid`

Свой PID-неймспейс: у приложения свой PID 1 и собственное дерево процессов.

Файловая система и Mounts `mnt`

Своё дерево монтирований и собственный root (/).

Сеть `net`

Собственные интерфейсы, IP-адреса и порты.

Hostname `uts`

Своё имя узла (hostname) и доменное окружение.

IPC `ipc`

Собственные механизмы межпроцессного взаимодействия (shared memory, очереди и др.).

Пользователи `user`

Собственная маппинг-таблица UID и GID (user namespaces).

Ресурсы `cgroup`

Своё представление о cgroup (ограничения по CPU, памяти и т.д.).

Время `time`

Собственные boot- и монотонные часы.

ПОПРОБУЙТЕ САМИ

```
root@lab:~$ lsns
```

Покажет все namespaces на вашей машине с их владельцами (PID).

PID Namespace

Изолирует дерево процессов: внутри такого окружения ваше приложение — **PID 1**, а процессы хоста для него не существуют.

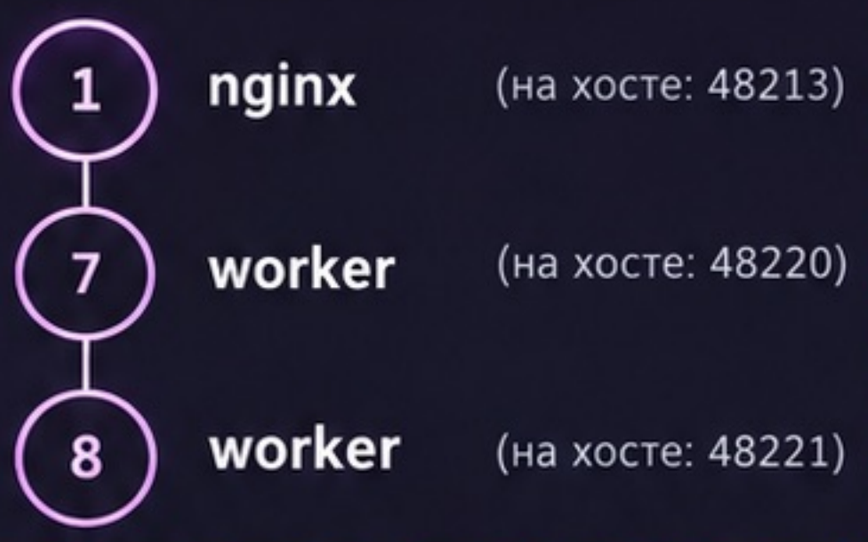


Дерево процессов на хосте



На хосте видны все процессы

PID Namespace
(внутренний взгляд)



Внутри — только эти 3 процесса

Не видит остальные процессы

Команда ps покажет только процессы внутри namespace (например, 3 вместо 300).

Не может с ними взаимодействовать

Вы не можете, например, отправить сигнал (kill –9) процессу на хосте.

Особая роль PID 1

Ваш процесс становится init: он должен корректно обрабатывать дочерние процессы (reap zombies).

ПОПРОБУЙТЕ САМИ

```
root@lab:~$ sudo unshare -pf --mount-proc ps -e
```

Запустит новый PID namespace: в выводе ps будут только ваш shell и сама ps.

07 · NAMESPACE · MNT

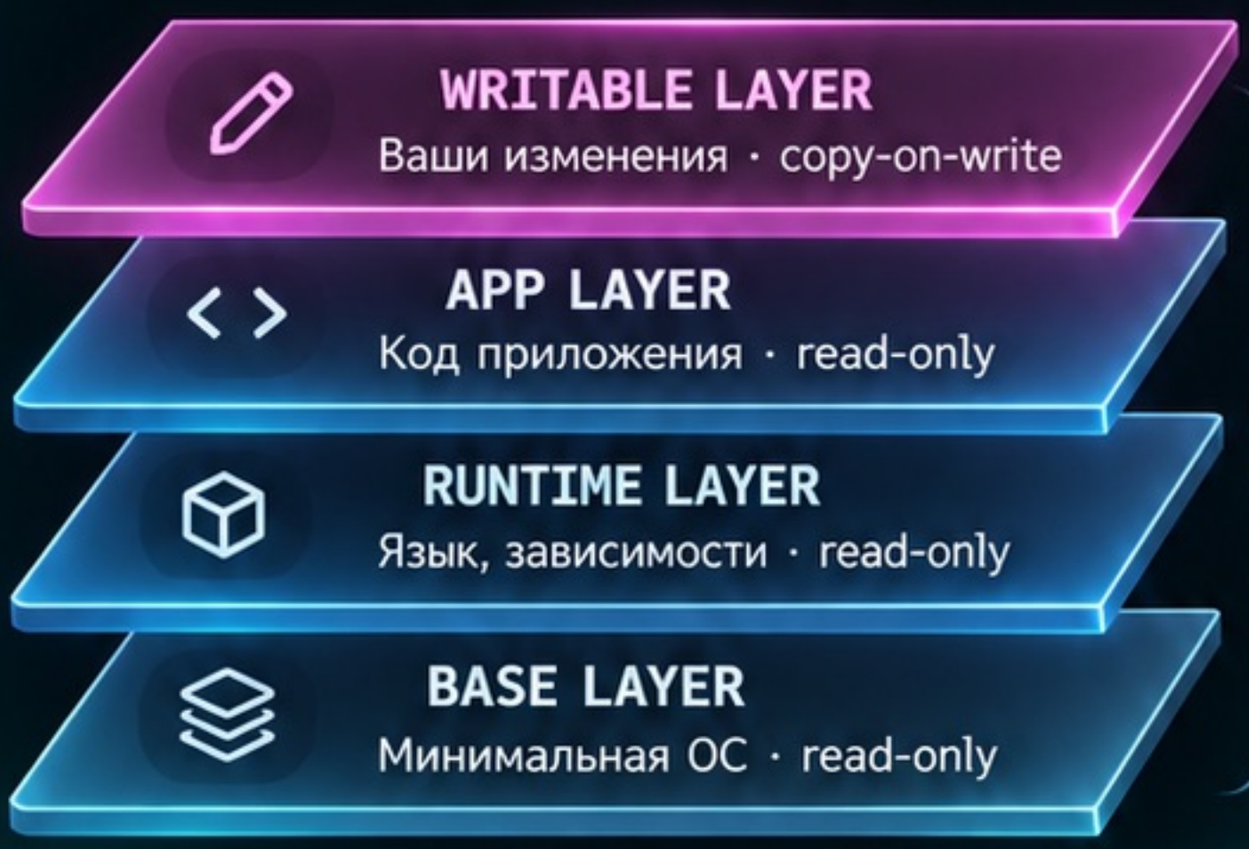
08/20

Mount Namespace и слои (Layers)



У контейнера — своё дерево файловой системы (/), собранное из **read-only** слоёв образа и одного тонкого **writable** слоя для изменений.

Слои образа (image layers)



Что видит процесс

```
/
├── bin
├── etc
├── lib
├── usr
├── app
│   └── server.py
└── tmp
    └── cache.db
```

Единое дерево из 4 слоёв

Общие, а не копии

Read-only слои используются всеми контейнерами из одного образа.

Copy-on-write

Изменяете файл — в writable слое появляется его копия, базовый слой не трогается.

Всё исчезает при удалении

Удаляете контейнер — вместе с ним пропадает и его writable слой (все изменения).

ПОПРОБУЙТЕ САМИ

```
root@lab:~$ findmnt /
```

Запустите внутри контейнера: / — это **overlay mount**, а не настоящий диск.

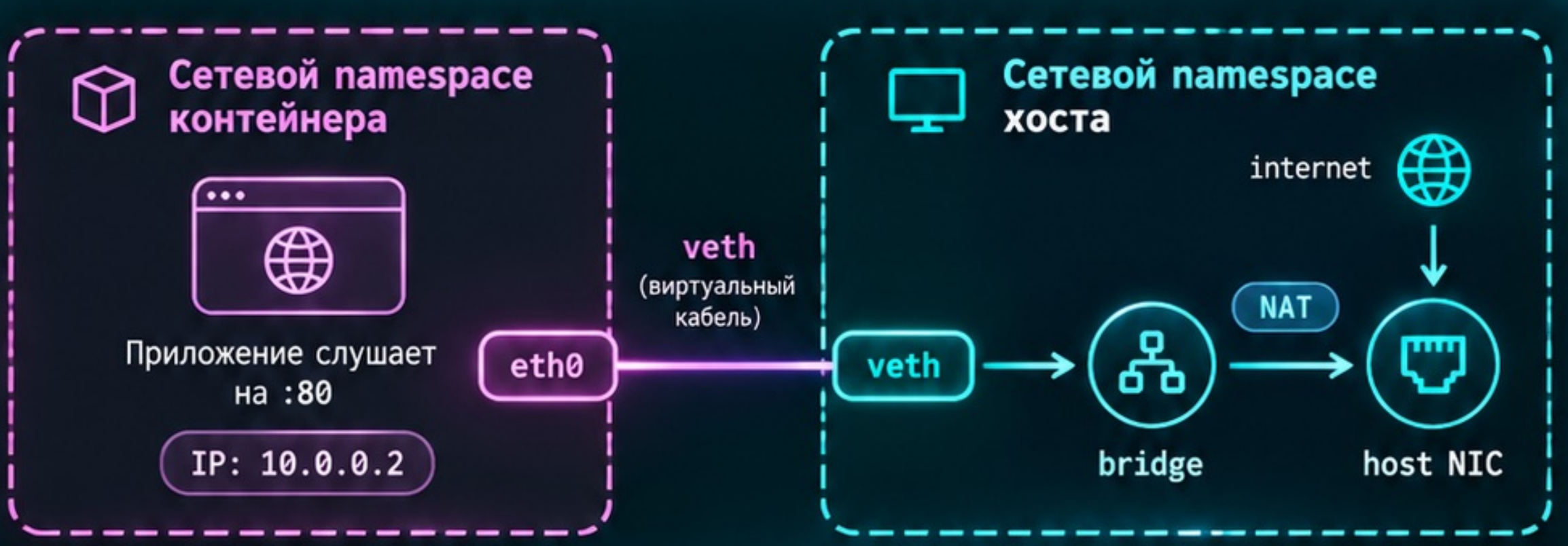
08 • NAMESPACE • NET

09/20

Network Namespace

У каждого контейнера свой сетевой стек: интерфейсы, IP-адрес, таблица маршрутов и порты.





host:8080 → контейнер:80

veth pair

Виртуальная «пара кабелей»: один конец в контейнере, другой — на хосте.

bridge

Виртуальный коммутатор, который объединяет интерфейсы контейнеров на хосте.

Port publish

Правила NAT, которые пробрасывают порт с хоста на порт внутри контейнера.

ПОПРОБУЙТЕ САМИ

```
root@lab:~$ ip link | grep veth
```

Каждый veth на хосте — это один конец «кабеля» контейнера.

09 · NAMESPACE · USER

User Namespace

Изолирует пользователей: внутри контейнера вы можете быть **root (uid 0)**, а на хосте — обычным **непривилегированным** пользователем.



ВНУТРИ КОНТЕЙНЕРА

root

uid 0

Полные привилегии внутри изоляции

СООТВЕТСТВИЕ UID

0	⟶	100000
1	⟶	100001
…	⟶	…
999	⟶	100999

Ядро отображает UID между namespace

НА ХОСТЕ

nobody

uid 100000

Обычный непривилегированный пользователь

Внутри контейнера

```
root@web:~# cat /proc/self/uid_map
         0     100000      65536
```

UID внутри ↑ | UID на хосте ↑ | Сколько ID ↑

Важно:
root внутри контейнера ≠ root на хосте.

⚠ Без **user namespace** **root внутри контейнера = root на хосте.**

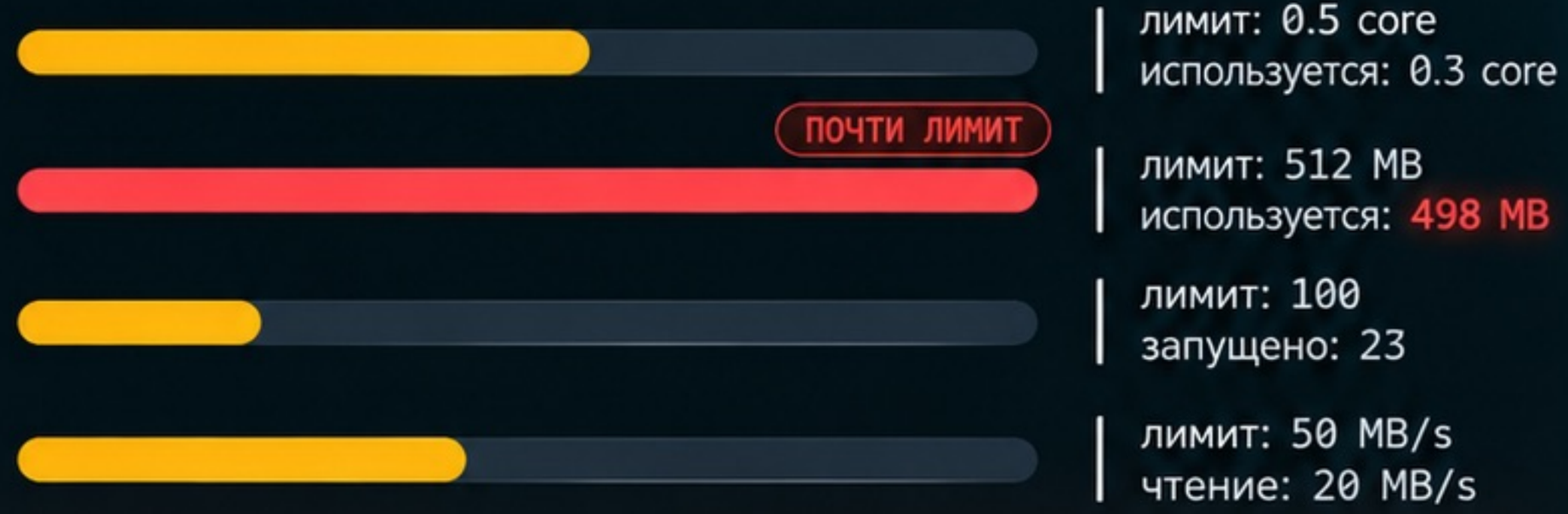
Одного уязвимого процесса достаточно, чтобы получить контроль над хостом.

cgroups: Лимиты ресурсов

Control groups задают «бюджет» на CPU, память и количество процессов. Превысили лимит — ядро вмешивается и ограничивает потребление.



Бюджет одного контейнера



Память > лимита → срабатывает **OOM killer**: процесс завершается, остальная система продолжает работать.

Без «шумных соседей»

Один жадный контейнер не отберёт весь CPU у остальных.

Защита от fork bomb

Лимит **pids.max** останавливает неконтролируемое создание процессов.

Учет потребления

Каждый байт и каждый цикл CPU учитывается на уровне группы.

ПОПРОБУЙТЕ САМИ

```
legion@lab:~$ cat /sys/fs/cgroup/memory.max
```

Внутри контейнера: лимит памяти в байтах или **max**, если он не задан.

Capabilities: root по частям

Linux разбивает полномочия root на **40+ capabilities.** Контейнер получает лишь небольшой, обычно безопасный набор.



root включает все 41 capability

Типичный набор по умолчанию

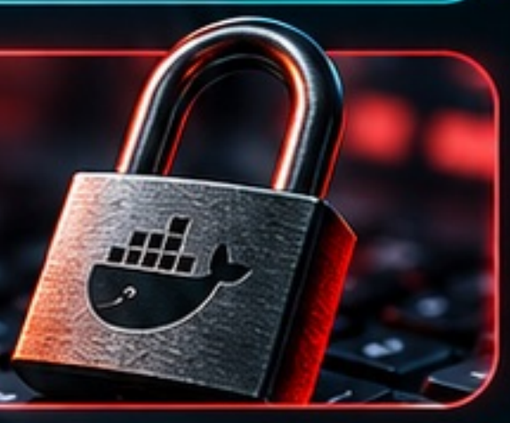
ОСТАЮТСЯ (14) — относительно безопасные

- CHOWN
- DAC_OVERRIDE
- FOWNER
- FSETID
- KILL
- SETGID
- SETUID
- SETPCAP
- NET_BIND_SERVICE
- SYS_CHROOT
- MKNOD
- AUDIT_WRITE
- NET_RAW

УДАЛЯЮТСЯ (27) — критичные и рискованные

- SYS_ADMIN
- SYS_MODULE
- SYS_PTRACE
- NET_ADMIN
- SYS_TIME
- SYS_BOOT
- SYS_RAWIO
- DAC_READ_SEARCH
- SYS_NICE
- SYS_RESOURCE
- MAC_ADMIN
- MAC_OVERRIDE
- BPF
- PERFMON
- SYS_PACCT
- LINUX_IMMUTABLE
- NET_BROADCAST
- IPC_LOCK
- IPC_OWNER
- LEASE
- AUDIT_CONTROL
- WAKE_ALARM
- BLOCK_SUSPEND
- SYSLOG
- AUDIT_READ
- CHECKPOINT_RESTORE
- SYS_TTY_CONFIG

Только **SYS_ADMIN** максимально приближает процесс к полноценному root. Его выдача контейнеру фактически сводит изоляцию на нет.



ПОПРОБУЙТЕ САМИ

```
legion@lab:~$ grep CapEff /proc/1/status
```

Покажет битовую маску capabilities для PID 1. Декодируйте её с помощью **capsh --decode**.

```
$ capsh --decode=0x00000000a80425fb
= cap_chown,cap_dac_override,...
```

seccomp: Фильтр системных вызовов

Каждое действие процесса — это **system call.** seccomp — фильтр, который проверяет их по правилам и решает: разрешить или заблокировать.





Allow-list

Профиль по умолчанию разрешает только те **system calls**, которые нужны обычным приложениям.

Blocked = EPERM

Запрещённый вызов завершается сразу с кодом **EPERM**. Ядро его не выполняет.

Сокращает поверхность атаки

Меньше доступных **system calls** — меньше потенциально достижимых уязвимостей в ядре.

ПОПРОБУЙТЕ САМИ

```
legion@lab:~$ grep Seccomp /proc/1/status
```

Seccomp: 2 означает режим фильтрации — для процесса активен профиль.

13 • МЕХАНИЗМЫ ЗАЩИТЫ

Mandatory Access Control

AppArmor и SELinux добавляют финальный уровень контроля: профиль определяет, к каким путям и действиям есть доступ — даже для **root**.



ДВА ПОСЛЕДОВАТЕЛЬНЫХ КОНТРОЛЯ

процесс → **DAC** права на файлы *Кто ты?* → **MAC** AppArmor / SELinux *Это вообще разрешено?* → разрешено

✓	`write /app/data.db`	разрешено
✗	`write /proc/sys/kernel/`	заблокировано профилем, даже для root
✗	`mount -t proc`	заблокировано профилем

Профили

Правила доступа: какие пути и операции разрешены контейнеру.

Метки (Labels)

SELinux маркирует файлы и процессы, чтобы A не мог читать данные B.

Подчиняет root

Даже uid 0 внутри контейнера не обходит профиль MAC.

Важно:



***DAC** отвечает на вопрос **«кто ты?»**.*
***MAC** — **«разрешено ли это вообще?»**.*
*Сначала срабатывает **DAC**, затем **MAC**.*

14 • КАК ВСЁ РАБОТАЕТ

Как запускается контейнер



Все «стены» возводятся за **миллисекунды** — в такой последовательности, ещё до старта вашего приложения.

1

Распаковка образа

Слои **(read-only)** объединяются в единое представление файловой системы.

SEE

2

clone() и новые namespaces

Создаются изолированные **PID**, **MNT**, **NET**, **UTS**, **IPC** и **USER** — собственный приватный мир.

SEE

3

Подключение к cgroup

К процессу применяются лимиты на CPU, память и количество процессов (PID).

USE

4

pivot_root

Объединённый **view** становится корнем **/**, а файловая система хоста скрывается.

SEE

5

Сброс capabilities

Привилегии root урезаются до минимального набора (по умолчанию).

DO

6

Загрузка seccomp + LSM profile

Включается фильтр **syscalls** (seccomp) и политики доступа (AppArmor / SELinux, LSM).

DO

7

Запуск вашего приложения

Процесс стартует как **PID 1** внутри своей изолированной среды.

RUN

Как ломается изоляция



*Ядро Linux редко подводит — проблемы обычно возникают из-за **удобства**: когда ради скорости и простоты отключают ключевые ограничения.*

Privileged mode

✕ Практически отключает все механизмы защиты из этого гайда.

✓ Не используйте в проде. Вместо этого добавляйте только необходимые **capabilities**.

Runtime socket внутри

✕ Монтирование **control socket** от **runtime** даёт контейнеру контроль над всем хостом.

✓ Не пробрасывайте его в контейнер. Если нужно — используйте **proxy** с минимальным **allow-list**.

Общие namespaces с хостом

✕ host **PID** или host **NET** стирают границы, ради которых вообще используются **namespaces**.

✓ Допускайте только для агентов мониторинга и строго по назначению.

Избыточные capabilities

✕ **SYS_ADMIN** или **SYS_PTRACE** возвращают большую часть полномочий **root**.

✓ Сбрасывайте всё по умолчанию и добавляйте только то, что действительно нужно.

Старое ядро (kernel)

✕ Общее ядро — единая поверхность атак: уязвимость в ядре затронет все контейнеры.

✓ Держите ядро и хост в актуальном состоянии и используйте **sandboxed runtime** для не доверенного кода.

Важно:

*Всё это — **конфигурация**, а не эксплойт. Правильные настройки предотвращают инциденты быстрее и дешевле.*

16 • РЕКОМЕНДАЦИИ

17/20

Hardening Checklist

Десять практик, которые действительно укрепляют изоляцию контейнеров. У большинства это просто **одна настройка** в конфигурации запуска.



Запускайте без root

Используйте обычного **USER** в образе, не оставляйте **uid 0**.

Включайте user namespaces

Rootless-режим: root остаётся внутри контейнера, снаружи — обычный пользователь.

Сбрасывайте capabilities

Удаляйте все и возвращайте только необходимые для приложения.

Держите seccomp + LSM

Оставляйте включёнными default профили (seccomp, AppArmor / SELinux) — базовая защита.

Делайте корневую ФС read-only

Монтируйте writable только там, где это действительно нужно.

Задавайте лимиты на memory, CPU, PID

Ограничивайте ресурсы, чтобы контейнер не мог занять весь хост.

Без privileged и host namespaces

Не используйте privileged и не шарьте host namespaces. Нужна интеграция с хостом — лучше отдельная **VM**.

Не пробрасывайте runtime socket

Не монтируйте Docker/CRI сокет внутрь контейнера — это фактически root на хосте.

Используйте минимальные образы

Меньше пакетов — меньше точек атаки. Регулярно пересобирайте образы.

Обновляйте kernel хоста

Единое ядро для всех контейнеров. Устраняйте уязвимости своевременно.

@lionsoulbatyshchikov

17 • БОЛЬШАЯ КАРТИНА

Спектр изоляции

Контейнеры — это компромисс в середине спектра. Двигаемся вправо — получаем более сильную **изоляцию**. Влево — выигрыш в скорости и плотности.



СКОРОСТЬ • ПЛОТНОСТЬ ← → ИЗОЛЯЦИЯ • БЕЗОПАСНОСТЬ

ВЫ ЗДЕСЬ

- **Обычный процесс** (plain process)
- **Контейнер (container)**
- **Изолированный контейнер** (sandboxed container)
- **Виртуальная машина** (virtual machine)
- **Отдельное железо** (separate hardware)

УРОВЕНЬ	ЧТО РАЗДЕЛЯЕТСЯ	ИЗОЛЯЦИЯ	НАКЛАДНЫЕ РАСХОДЫ
Plain process	Ничего: общий userspace, ресурсы и kernel.	■□□□□	■□□□□
Container	Namespaces, cgroups, фильтры. Общий kernel с хостом.	■■■□□	■■□□□
Sandboxed container	Доп. изоляция на уровне userspace (напр. gVisor, Kata, Firecracker).	■■■■□	■■■□□
Virtual machine	Свой kernel под управлением гипервизора (hypervisor).	■■■■■	■■■■□
Separate hardware	Полное разделение: нет общего состояния.	■■■■■	■■■■■

Вывод: Контейнеры дают хороший баланс изоляции, скорости и утилизации ресурсов.

18 · ШПАРГАЛКА

19/20

Cheat Sheet

Ключевые механизмы изоляции контейнеров: что они делают, чем управляют и как быстро проверить их состояние.

№	МЕХАНИЗМ	ЧЕМ УПРАВЛЯЕТ	КАК ПРОВЕРИТЬ
1	**Namespaces**	Изоляция **PID, MNT, NET, UTS, IPC, USER, CGROUP, TIME.**	`lsns`
2	**cgroups**	Лимиты и учет ресурсов: **CPU, memory, PIDs, I/O.**	`cat /sys/fs/cgroup/memory.max`
3	**Capabilities**	Набор привилегий, которыми располагает процесс (в т.ч. **root**).	`grep Cap /proc/1/status`
4	**seccomp**	Фильтрация allowed **syscalls** на уровне ядра.	`grep Seccomp /proc/1/status`
5	**LSM profile**	Политики доступа (**AppArmor / SELinux**): разрешённые пути и действия.	`cat /proc/1/attr/current`
6	**Image layers**	Слои образа: **read-only** слои и один **writable** (верхний).	`findmnt /`
7	**User namespace**	Маппинг **uid**: root внутри контейнера соответствует обычному **uid** снаружи.	`cat /proc/self/uid_map`

Важно:



*Контейнер — это процесс, который видит «**свою**» (ограниченную) картину ядра. В этом и заключается суть **изоляции**.*

20 • ИТОГ

20/20

Было полезно?

Короткий гид по изоляции контейнеров в **Linux**: ключевые механизмы, практики и команды в одном месте.



Сохраните
чтобы не искать, когда понадобится

Поделитесь
с коллегой, который разбирается в контейнерах




```
$
```

- [x] Namespaces
- [x] cgroups
- [x] Capabilities
- [x] seccomp
- [x] LSM

@lionsoulbatyshchikov