

The Kaspersky logo is displayed in a bold, black, lowercase sans-serif font. It is positioned in the upper left quadrant of the page, which features a white, rounded rectangular shape against a teal background with a green-to-teal gradient.

**kaspersky**

# **KasperskyOS Community Edition 1.0**

© 2021 АО «Лаборатория Касперского»

# Содержание

## [Что нового](#)

### [О KasperskyOS Community Edition](#)

[Об этом документе](#)

[Системные требования](#)

[Комплект поставки](#)

[Включенные сторонние библиотеки и приложения](#)

[Ограничения и известные проблемы](#)

### [KasperskyOS: обзор](#)

[Основные понятия KasperskyOS](#)

[Кибериммунитет](#)

[Изоляция и взаимодействие сущностей](#)

[Сущности](#)

[Взаимодействие сущностей \(IPC\)](#)

[Описания интерфейсов сущности \(EDL, CDL, IDL\)](#)

[IPC-транспорт](#)

[Контроль взаимодействий](#)

[Модуль безопасности Kaspersky Security Module](#)

[Язык описания политик PSL](#)

[Контроль доступа к ресурсам](#)

[Надежность TCB](#)

[Микроядерная архитектура](#)

[Надежность доверенных компонентов](#)

[Образ решения на базе KasperskyOS](#)

### [Начало работы](#)

[Установка и удаление](#)

[Настройка среды разработки](#)

[Сборка и запуск примеров](#)

[Сборка примеров](#)

[Запуск примеров на QEMU](#)

[Запуск примеров на Raspberry Pi 4 B](#)

### [Часть 1. Простейшее приложение \(POSIX\)](#)

[Пример hello](#)

[VFS: работа с файлами и сетью](#)

[VFS: обзор](#)

[Сборка сущности VFS](#)

[Сущность Env](#)

[Соединение сущности-клиента с одной и двумя сущностями VFS](#)

[Монтирование файловых систем при запуске VFS](#)

[Ограничения поддержки POSIX](#)

### [Часть 2. Взаимодействие сущностей](#)

[Инструменты IPC-транспорта](#)

[Пример echo](#)

[О примере echo](#)

[Реализация сущности Client в примере echo](#)

[Реализация сущности Server в примере echo](#)

[Файлы описаний в примере echo](#)

[Сборка и запуск примера echo](#)

### [Часть 3. Политика безопасности решения](#)

[Политики безопасности](#)

[Политика безопасности решения \(security.psl\)](#)

[Структура файла security.psl](#)

[Описание глобальных параметров модуля безопасности](#)

[Подключение других PSL-описаний](#)

[Подключение EDL-описаний сущностей](#)

[Создание объектов классов политик](#)

[Связывание событий с политиками](#)

[Объявление и назначение профилей аудита](#)

[Тестирование политики безопасности решения на языке Policy Assertion Language \(PAL\)](#)

[Политики-правила и политики-выражения](#)

[Пример простейшей политики безопасности решения](#)

[Пример ping](#)

[О примере ping](#)

[Реализация сущности Client в примере ping](#)

[Реализация сущности Server в примере ping](#)

[Файлы описаний в примере ping](#)

[Политика безопасности решения в примере ping](#)

[Сборка и запуск примера ping](#)

### [KasperskyOS API](#)

[Описание сущностей, компонентов и интерфейсов \(EDL, CDL, IDL\)](#)

[Модель "сущность-компонент-интерфейс"](#)

[EDL](#)

[CDL](#)

[IDL](#)

[Типы данных IDL](#)

[Работа с ошибками в IDL](#)

[Составные имена сущностей, компонентов и интерфейсов](#)

[Запуск сущностей](#)

[Сущность Einit](#)

[Файл init.yaml](#)

[IPC и транспорт](#)

[Реализация IPC](#)

[Основы IPC в KasperskyOS](#)

[IPC-каналы](#)

[Динамическое создание IPC-каналов](#)

[Общая схема обмена сообщениями](#)

[Локатор сервисов \(Service Locator\)](#)

[Группы каналов](#)

[Транспорт](#)

[Структура IPC-сообщения](#)

[Транспорт NkKosTransport](#)

[Сгенерированные методы и типы](#)

[Классы политик безопасности](#)

[Класс политик Base](#)

[Класс политик Regex](#)

## Класс политик HashSet

Объект класса HashSet

Политика init класса HashSet

Политика fini класса HashSet

Политика add класса HashSet

Политика remove класса HashSet

Политика contains класса HashSet

## Класс политик StaticMap

Объект класса StaticMap

Политика init класса StaticMap

Политика fini класса StaticMap

Политика set класса StaticMap

Политика commit класса StaticMap

Политика rollback класса StaticMap

Политика get класса StaticMap

Политика get\_uncommitted класса StaticMap

## Класс политик Flow

Объект класса Flow

Политика init класса Flow

Политика fini класса Flow

Политика query класса Flow

Политика enter класса Flow

Политика allow класса Flow

## Класс политик Rbac

Основные понятия RBAC

Объект класса Rbac

Полномочия

Роли

Ограничения/правила: создание субъекта

Ограничения/правила: создание объекта

Ограничения/правила: изменение типа объекта

Ограничения/правила: добавление ролей к субъекту

Политика create\_subject класса Rbac

Политика create\_object класса Rbac

Политика check класса Rbac

Политика retype\_object класса Rbac

Политика add\_roles класса Rbac

Политика query\_type класса Rbac

## Инструменты для сборки решения

### Скрипты и компиляторы

Утилиты и скрипты сборки

nk-gen-c

nk-psl-gen-c

einit

makekss

makeimg

Кросс-компиляторы

Подготовка загрузочного образа решения

[Использование шаблона Makefile из состава KasperskyOS Community Edition](#)

[Использование CMake из состава KasperskyOS Community Edition](#)

[Использование собственной системы сборки](#)

[Развертывание загрузочного образа решения на целевых устройствах](#)

[Паттерны безопасности при разработке под KasperskyOS](#)

[Паттерн Distrustful Decomposition](#)

[Пример Secure Logger](#)

[Пример Separate Storage](#)

[Паттерн Defer to Kernel](#)

[Пример Defer to Kernel](#)

[Паттерн Policy Decision Point](#)

[Паттерн Privilege Separation](#)

[Пример Device Access](#)

[Паттерн Information Obscurity](#)

[Пример Secure Login](#)

[Приложения](#)

[Дополнительные примеры](#)

[Пример net\\_with\\_separate\\_vfs](#)

[Пример net2\\_with\\_separate\\_vfs](#)

[Пример embedded\\_vfs](#)

[Пример embed\\_ext2\\_with\\_separate\\_vfs](#)

[Пример multi\\_vfs\\_ntpd](#)

[Пример multi\\_vfs\\_dns\\_client](#)

[Пример multi\\_vfs\\_dhcpd](#)

[Пример mqtt\\_publisher](#)

[Пример mqtt\\_subscriber](#)

[Пример gpio\\_input](#)

[Пример gpio\\_output](#)

[Пример gpio\\_interrupt](#)

[Пример gpio\\_echo](#)

[Лицензирование программы](#)

[Предоставление данных](#)

[Информация о стороннем коде](#)

[Уведомления о товарных знаках](#)

## Что нового

В KasperskyOS Community Edition 1.0 появились следующие возможности и доработки:

- Добавлена поддержка аппаратной платформы Raspberry Pi 4 Model B.
- Добавлена поддержка SD-карты для аппаратной платформы Raspberry Pi 4 Model B.
- Добавлена поддержка Ethernet для аппаратной платформы Raspberry Pi 4 Model B.
- Добавлена поддержка портов ввода-вывода GPIO для аппаратной платформы Raspberry Pi 4 Model B.
- Добавлены сетевые сервисы DHCP, DNS, NTP и примеры работы с ними.
- Добавлена библиотека для работы с протоколом MQTT и примеры ее использования.

# О KasperskyOS Community Edition

KasperskyOS Community Edition (CE) — общедоступная версия KasperskyOS, предназначенная для освоения основных принципов разработки приложений под KasperskyOS. KasperskyOS Community Edition позволит вам увидеть, как концепции, заложенные в KasperskyOS, работают на практике. KasperskyOS Community Edition включает в себя примеры приложений с исходным кодом, подробные пояснения, а также инструкции и инструменты для сборки приложений.

KasperskyOS Community Edition пригодится вам для:

- изучения принципов и приемов разработки "secure by design" на практических примерах;
- изучения KasperskyOS как возможной платформы для реализации своих проектов;
- прототипирования решений (прежде всего, Embedded/IoT) на основе KasperskyOS;
- портирования приложений/компонентов/драйверов на KasperskyOS;
- изучения вопросов безопасности в разработке ПО.

Для получения KasperskyOS Community Edition перейдите по [ссылке](#).

Помимо этой документации, также рекомендуем изучить материалы [раздела сайта](#) KasperskyOS для разработчиков.

## Об этом документе

Руководство разработчика KasperskyOS Community Edition адресовано специалистам, которые осуществляют разработку безопасных решений на базе KasperskyOS.

Руководство предназначено специалистам, обладающим следующими навыками: знание языков C/C++, опыт разработки под POSIX-совместимые системы, знакомство с GNU Binary Utilities (далее также "binutils").

Вы можете применять информацию в этом руководстве для выполнения следующих задач:

- установка и удаление KasperskyOS Community Edition;
- использование KasperskyOS Community Edition.

## Основные понятия

Ниже приведены часто используемые термины, связанные с KasperskyOS Community Edition:

- KasperskyOS – микроядерная операционная система для построения безопасных решений.
- Kaspersky Security System – подсистема, осуществляющая контроль взаимодействия между сущностями в соответствии с заданной политикой безопасности решения.
- *Политика безопасности* – определенное правило проверки допустимости взаимодействия между сущностями. Политики безопасности используются при описании конфигурации безопасности и объединяются в *классы политик*.

- *Политика безопасности решения* – описание политик безопасности, а также правила их применения к событиям.
- *Сущность* – отдельное адресное пространство с одним или более потоками выполнения, ассоциированными с ним.
- *Решение* – программно-аппаратный комплекс, реализованный на базе KasperskyOS.
- *Язык описания интерфейсов* (*Interface Definition Language*, далее также "*IDL*") – декларативный язык, специфицирующий перечень интерфейсов, типов и именованных констант.
- *Язык описания компонентов* (*Component Definition Language*, далее также "*CDL*") – декларативный язык, специфицирующий перечень реализаций интерфейсов, которые включены в компонент, а также список вложенных в компонент.
- *Язык описания сущностей* (*Entity Definition Language*, далее также "*EDL*") – декларативный язык, специфицирующий перечень экземпляров компонентов, которые включены в сущность, а также список реализуемых сущностью интерфейсов.
- *Язык описания политики безопасности решения* (*Policy Specification Language*, далее также *PSL*) – декларативный язык, позволяющий задать правила контроля допустимости следующих событий в KasperskyOS: запуск сущностей, взаимодействие сущностей, обращение сущностей к Kaspersky Security System по интерфейсам безопасности.
- *Init-описание* (*init.yaml*) – вспомогательный формат для статической спецификации сущностей. Позволяет указать сущности, инициализируемые при запуске, и устанавливает возможность взаимодействия между ними.

## Системные требования

Для установки KasperskyOS Community Edition и запуска примеров под QEMU необходимы:

1. **Операционная система:** Debian GNU/Linux® "Buster" версии 10.7 x64.
2. **Процессор:** процессор с архитектурой x86-64 (для большей производительности требуется поддержка аппаратной виртуализации).
3. **Оперативная память:** для комфортной работы с инструментами сборки рекомендуется иметь не менее 4 Гб оперативной памяти.
4. **Дисковое пространство:** не менее 3 Гб свободного пространства в разделе `/opt` (в зависимости от разрабатываемого решения);

Для [запуска примеров на аппаратной платформе](#) Raspberry Pi, необходимо использовать модель Raspberry Pi 4 Model B с любым объемом оперативной памяти.

## Комплект поставки

В комплект поставки KasperskyOS Community Edition входят:

- deb-пакет для установки KasperskyOS Community Edition, содержащий:
  - образ ядра операционной системы KasperskyOS;



- компоненты KasperskyOS Community Edition;
- набор инструментов для разработки решения (компилятор NK, компилятор GCC, отладчик GDB, набор утилит binutils, эмулятор QEMU и сопутствующие инструменты);
- лицензионное соглашение;
- файл с информацией о стороннем коде (Legal Notices).
- Информация о версии (Release Notes);
- Руководство разработчика KasperskyOS Community Edition (онлайн-документация).




Следующие компоненты, входящие в комплект поставки KasperskyOS Community Edition, являются "Runtime компонентами" в соответствии с условиями лицензионного соглашения:

- Образ ядра операционной системы KasperskyOS.

Остальные части комплекта поставки не являются "Runtime компонентами". Условия и возможности использования каждого компонента могут быть дополнительно указаны в разделе ["Информация о стороннем коде"](#).

## Включенные сторонние библиотеки и приложения

Для упрощения процесса разработки приложений в состав KasperskyOS Community Edition также включены следующие сторонние библиотеки и приложения:

- **Boost (v1.71)** – собрание библиотек классов, использующих функциональность языка C++ и предоставляющих удобный кроссплатформенный высокоуровневый интерфейс для лаконичного кодирования различных повседневных подзадач программирования (работа с данными, алгоритмами, файлами, потоками и т. п.)  
Документация: <https://www.boost.org/doc/> 
- **Arm Mbed TLS (v.2.16)** – реализация протоколов TLS и SSL, а также соответствующих криптографических алгоритмов и необходимого кода поддержки.  
Документация: <https://tls.mbed.org/kb> 
- **Civetweb (v1.11)** – простой в использовании, мощный, встраиваемый веб-сервер на C / C++ с дополнительной поддержкой CGI, SSL и Lua.  
Документация: <http://civetweb.github.io/civetweb/UserManual.html> 
- **Eclipse Mosquitto (v1.6.4)** – брокер сообщений, реализующий протокол MQTT.  
Документация: <https://mosquitto.org/documentation/>

Также см. [Информация о стороннем коде](#).

## Ограничения и известные проблемы

Поскольку KasperskyOS Community Edition предназначен для обучения, мы интегрировали в пакет ряд ограничений:

1. Не поддерживается симметричная многопроцессорность (SMP). Используется только одно ядро процессора.
2. Не поддерживается динамическая загрузка библиотек.
3. Максимальное поддерживаемое количество запущенных приложений (сущностей): 32.
4. При завершении работы сущности любым способом (например, return из основного потока исполнения) выделенные сущностью ресурсы не освобождаются, а сама сущность переводится в "спящее" состояние. Сущности не могут быть запущены повторно.
5. Не поддерживается запуск двух и более сущностей с одинаковым EDL-описанием.
6. Система останавливается, если не осталось работающих сущностей или если один из потоков драйверной сущности завершился (штатным или нештатным образом).

## Основные понятия KasperskyOS

### Решение на базе KasperskyOS

Информационная система, работающая под управлением ядра KasperskyOS, называется *решением на базе KasperskyOS*. Процессы в KasperskyOS называются *сущностями*. Ядро гарантирует, что сущности изолированы и могут взаимодействовать только через ядро (с помощью системных вызовов). Каждая сущность в решении имеет *статическое описание*, определяющее интерфейсы, доступные другим сущностям. Для описания интерфейсов используются специально разработанные языки – EDL, CDL и IDL.

### Кибериммунный подход

Для разработки безопасных решений на базе KasperskyOS используется *кибериммунный подход*. Этот подход основан на выборе способа разбиения системы на сущности и задании определенных правил их взаимодействия (т.н. *политика безопасности решения*). Политика безопасности реализуется *модулем безопасности Kaspersky Security Module*, который входит в решение.

Кибериммунный подход позволяет защитить доверенные компоненты системы и минимизировать ее поверхность атаки. Даже в случае компрометации одного из компонентов такой системы остальные компоненты продолжают выполнять функции безопасности.

[Подробнее о кибериммунном подходе](#)

### Kaspersky Security System

*Технология Kaspersky Security System* позволяет разрабатывать и реализовывать разнообразные политики безопасности. При этом можно комбинировать несколько моделей безопасности, добавлять свои модели и гибко настраивать правила взаимодействия сущностей. Для формального описания политики безопасности решения используется специально разработанный язык PSL. На основе PSL-описания генерируется модуль безопасности Kaspersky Security Module для использования в конкретном решении.

### KasperskyOS Community Edition

KasperskyOS Community Edition содержит инструменты для разработки безопасных решений на базе KasperskyOS, включая:

- образ ядра операционной системы KasperskyOS;
- *компилятор NK*, предназначенный для генерации модуля безопасности Kaspersky Security Module и вспомогательного транспортного кода;
- прочие инструменты для разработки решения (компилятор GCC, отладчик GDB, набор утилит binutils, эмулятор QEMU и сопутствующие инструменты);
- набор библиотек, обеспечивающих частичную совместимость со стандартом POSIX;

- компоненты KasperskyOS Community Edition;
- документацию;
- примеры простейших решений на базе KasperskyOS.

## Кибериммунитет

Концепция кибериммунитета основана на следующих понятиях:

- цели и предположения безопасности;
- понятия модели MILS (домен безопасности, ядро разделения и монитор обращений);
- доверенная вычислительная база (TCB).

Ниже рассмотрены эти понятия, после чего даны определения кибериммунной системы и кибериммунного подхода.

### Цели и предположения безопасности

Безопасность информационной системы не является универсальным абстрактным понятием. Является ли данная система безопасной, зависит от выбранных целей и предположений безопасности.

*Цели безопасности* – это требования, предъявляемые к информационной системе, выполнение которых обеспечивает безопасное функционирование информационной системы в любых возможных сценариях ее использования с учетом предположений безопасности. Пример цели безопасности: соблюдение конфиденциальности данных при использовании канала связи.

*Предположения безопасности* – дополнительные ограничения, накладываемые на условия эксплуатации системы, при которых система выполняет цели безопасности. Пример предположения безопасности: у злоумышленника отсутствует физический доступ к аппаратной платформе.

### Понятия модели MILS

В рамках *модели MILS (Multiple Independent Levels of Security)* безопасная информационная система состоит из изолированных *доменов безопасности* и *ядра разделения*, контролирующего взаимодействия доменов друг с другом. Ядро разделения обеспечивает изоляцию доменов и управление информационными потоками между ними.

Каждая попытка взаимодействия между доменами безопасности проверяется на соответствие определенным правилам, которые задаются *политикой безопасности системы*. Если взаимодействие запрещено текущей политикой, оно не пропускается (блокируется). Политику безопасности в архитектуре MILS реализует отдельный компонент – *монитор обращений*. Для каждого взаимодействия доменов безопасности монитор обращений возвращает *решение* (булево значение), соответствует ли это взаимодействие политике безопасности. Ядро разделения вызывает монитор каждый раз, когда один домен обращается к другому.

### Доверенная вычислительная база (TCB)

*Доверенная вычислительная база (Trusted Computing Base, TCB)* – совокупность всего программного кода, уязвимость в котором приводит к невозможности выполнения информационной системой заданных целей безопасности. В модели MILS ядро разделения и монитор обращений составляют основу доверенной вычислительной базы.

Надежность доверенной вычислительной базы играет ключевую роль в обеспечении безопасности информационной системы.

## Кибериммунная система

Информационная система является *кибериммунной* (или *обладает кибериммунитетом*), если она разделена на изолированные домены безопасности, все взаимодействия между которыми независимо контролируются, и предоставляет:

- описание своих целей и предположений безопасности;
- гарантии надежности всей доверенной вычислительной базы, включая среду исполнения и средства контроля взаимодействий;
- гарантии выполнения целей безопасности во всех возможных сценариях использования системы с учетом описанных предположений, кроме компрометации доверенной вычислительной базы решения.

## Кибериммунный подход

*Кибериммунный подход* – это способ построения кибериммунных систем.

Кибериммунный подход основан на:

- разбиении системы на изолированные домены безопасности;
- независимом контроле всех взаимодействий между доменами безопасности на соответствие заданной политике безопасности;
- обеспечении надежности доверенной кодовой базы.

Конкретный способ разбиения системы на домены безопасности и выбор политики безопасности зависят от целей и предположений безопасности системы, степени доверенности и целостности отдельных компонентов, а также других факторов.

## Преимущества кибериммунного подхода

Кибериммунный подход позволяет:

- свести свойства безопасности системы в целом к свойствам безопасности отдельных ее компонентов;
- предоставить гарантии выполнения целей безопасности системы даже при компрометации любого ее недоверенного компонента;
- снизить требования к одному или нескольким компонентам системы относительно требований к системе в целом;
- минимизировать ущерб для системы в целом при компрометации любого ее компонента;

- упростить процедуру сертификации системы.

## Изоляция и взаимодействие сущностей

Кибериммунная система состоит из изолированных частей (доменов безопасности в терминах MILS), которые могут взаимодействовать друг с другом только через ядро разделения, т.е. контролируемым образом. В KasperskyOS реализацией доменов безопасности являются *сущности*.

## Сущности

Каждый процесс в KasperskyOS – это субъект в политике безопасности решения. При запуске процесса ядро KasperskyOS ассоциирует с ним контекст, необходимый для его исполнения, а модуль Kaspersky Security Module – контекст безопасности, необходимый для контроля его взаимодействий с другими процессами.

Чтобы подчеркнуть связь каждого процесса с политикой безопасности, процессы в KasperskyOS называются *сущностями*.

С точки зрения ядра KasperskyOS, сущность – это процесс, имеющий отдельное адресное пространство и один или несколько потоков исполнения. Ядро гарантирует изоляцию адресных пространств сущностей. Сущность может реализовывать интерфейсы, а другие сущности – вызывать методы этих интерфейсов через ядро.

С точки зрения модуля безопасности Kaspersky Security Module, сущность является субъектом, с которым могут взаимодействовать другие субъекты (сущности). Возможные виды взаимодействий задаются описанием интерфейсов сущности, которые должны соответствовать реализации. Описания интерфейсов позволяют модулю безопасности проверять каждое взаимодействие сущностей на соответствие политике безопасности решения.

## Дополнительная информация по сущностям

Для модуля безопасности Kaspersky Security Module ядро является таким же субъектом, как и сущности. Сущности могут вызывать методы ядра, и эти взаимодействия контролируются аналогично вызовам методов других сущностей. Поэтому далее мы будем говорить, что ядро является *отдельной сущностью* с точки зрения Kaspersky Security Module.

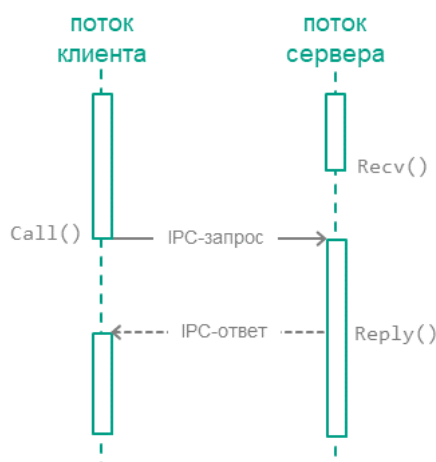
## Взаимодействие сущностей (IPC)

В KasperskyOS есть только один способ взаимодействия сущностей – с помощью синхронного обмена IPC-сообщениями: *запросом* и *ответом*. В каждом взаимодействии выделяются две роли: *клиент* (сущность, инициирующая взаимодействие) и *сервер* (сущность, обрабатывающая обращение). При этом сущность, являющаяся клиентом в одном взаимодействии, может выступать как сервер в другом.

Клиент и сервер используют три системных вызова: `Call()`, `Recv()` и `Reply()`:

1. Клиент направляет серверу сообщение-запрос. Для этого один из потоков исполнения клиента выполняет вызов `Call()` и блокируется до получения ответа от сервера или ядра (например, в случае ошибки).

- Серверный поток, выполнивший вызов `Recv()`, находится в ожидании сообщений. При получении запроса этот поток разблокируется, обрабатывает запрос и отправляет сообщение-ответ с помощью вызова `Reply()`.
- При получении сообщения-ответа (или ошибки) клиентский поток разблокируется и продолжает исполнение.

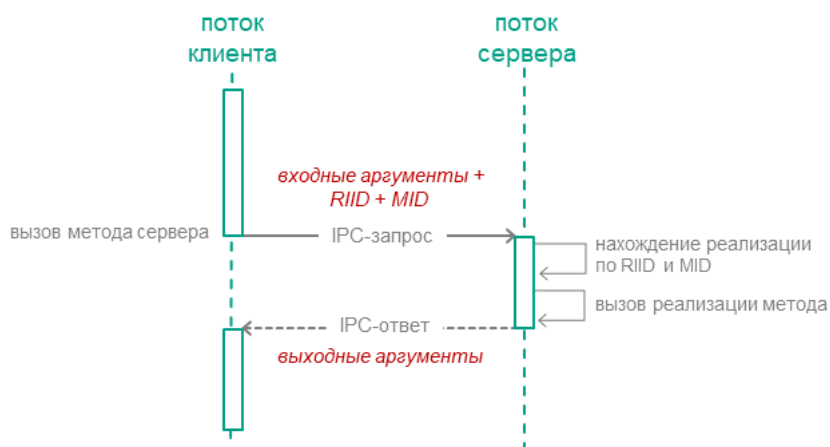


Таким образом, ядро KasperskyOS является ядром разделения в терминах [модели MILS](#), поскольку все взаимодействия сущностей проходят через него.

## Обмен сообщениями как вызов метода

Отправка IPC-запроса сущности-серверу представляет собой обращение к одному из интерфейсов, реализуемых сервером. IPC-запрос содержит входные аргументы вызываемого метода, а также идентификатор реализации интерфейса и идентификатор вызываемого метода. Получив запрос, сущность-сервер использует эти идентификаторы, чтобы найти реализацию метода. Сервер вызывает реализацию метода, передав в нее входные аргументы из IPC-запроса. Обработав запрос, сущность-сервер отправляет клиенту ответ, содержащий выходные аргументы метода.

Модуль безопасности Kaspersky Security Module может анализировать все компоненты IPC-сообщения, чтобы вынести решение, соответствует ли это сообщение политике безопасности системы.



## IPC-каналы

Чтобы две сущности могли обмениваться сообщениями, между ними должен быть установлен *IPC-канал* (далее также *канал* или *соединение*). Канал задает роли сущностей, т.е. имеет "клиентскую" и "серверную" стороны. При этом одна сущность может иметь несколько каналов, в которых она является клиентом, и несколько каналов, где она – сервер.

В KasperskyOS предусмотрено два способа создания IPC-каналов:

1. Статический способ предполагает создание канала в момент запуска сущностей (при старте решения). Статическое создание каналов выполняется инициализирующей сущностью Einit.
2. Динамический способ позволяет уже запущенным сущностям установить канал между собой.

## Описания интерфейсов сущности (EDL, CDL, IDL)

Для контроля взаимодействий между сущностями необходимо, чтобы структура пересылаемых IPC-сообщений была прозрачна для модуля безопасности. В KasperskyOS это достигается с помощью статического декларирования интерфейсов сущностей. Для этого используются специальные языки: Entity Definition Language (EDL), Component Definition Language (CDL) и Interface Definition Language (IDL). Если IPC-сообщение не соответствует описанию интерфейса, оно будет отклонено модулем безопасности.

Описание интерфейсов сущности определяет допустимые структуры IPC-сообщений. Так задается явная связь между реализацией каждого метода и тем, как вызов этого метода представлен для модуля безопасности. Почти все инструменты сборки решения явно или косвенно используют описания интерфейсов сущностей.

### Виды статических описаний

Описание интерфейсов сущности строится по модели "сущность-компонент-интерфейс":

- IDL-описание декларирует интерфейс, а также пользовательские типы и константы (опционально). В сумме все IDL-описания решения охватывают все реализуемые в решении интерфейсы.
- В CDL-описании перечислены интерфейсы, реализуемые *компонентом*. Компоненты дают возможность группировать реализации интерфейсов. Компоненты могут включать в себя другие компоненты.
- В EDL-описании сущности декларируются экземпляры компонентов, входящие в эту сущность. В частности, сущность может не включать в себя ни один компонент.

### Пример

Ниже приведены статические описания решения, состоящего из сущности `Client`, не реализующей ни одного интерфейса, и сущности `Server`, реализующей интерфейс `FileOps`.

`Client.edl`

```
// Статическое описание состоит только из имени сущности
entity Client
```



#### Server.edl

```
// Сущность Server содержит экземпляр компонента Operations
entity Server
components {
    OpsComp: Operations
}
```

#### Operations.cdl

```
// Компонент Operations реализует интерфейс FileOps
component Operations
interfaces {
    FileOpsImpl: FileOps
}
```

#### FileOps.idl

```
package FileOps
// Объявление пользовательского типа String
typedef array <UInt8, 256> String;
// Интерфейс FileOps содержит единственный метод Open с входным аргументом name и
// выходным аргументом h
interface {
    Open(in String name, out UInt32 h);
}
```

См. подробнее о [синтаксисе статических описаний](#).

## IPC-транспорт

Чтобы реализовать взаимодействие сущностей, необходим *транспортный код*, отвечающий за корректное создание IPC-сообщений, их упаковку, отправку, распаковку и диспетчеризацию. Разработчику решения под KasperskyOS нет необходимости самостоятельно писать транспортный код. Вместо этого можно использовать специальные инструменты и библиотеки, поставляемые в составе KasperskyOS Community Edition.

### Транспортный код для разрабатываемых компонентов

Разработчик новых компонентов для KasperskyOS может сгенерировать транспортный код на основе [статических описаний](#) этого компонента. Для этого в составе KasperskyOS Community Edition поставляется компилятор НК. Компилятор НК позволяет генерировать транспортные методы и типы для использования как на клиентской стороне, так и на серверной.

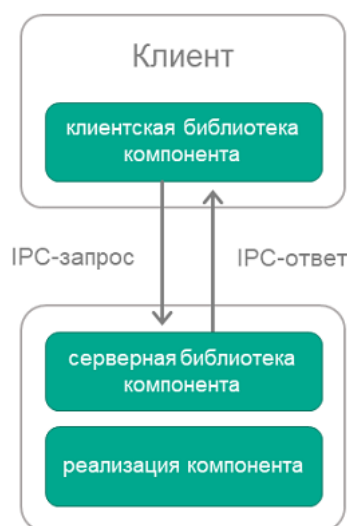
### Транспортный код для поставляемых компонентов

Функциональность большинства компонентов, поставляемых в составе KasperskyOS Community Edition, может быть использована в решении как локально, т.е. путем статической компоновки с разрабатываемой сущностью, так и через IPC.

Для вынесения компонента в отдельную серверную сущность и использования его по IPC поставляются следующие *транспортные библиотеки*.

- *Клиентская библиотека компонента* преобразует локальные вызовы в IPC-запросы к сущности драйвера.
- *Серверная библиотека компонента* принимает IPC-запросы к сущности драйвера и преобразует их в локальные вызовы.

Чтобы использовать компонент по IPC, достаточно его реализацию скомпоновать с серверной библиотекой, а клиентскую сущность скомпоновать с клиентской библиотекой.



Интерфейс клиентской библиотеки не отличается от интерфейса самого компонента. Таким образом, для перехода на использование компонента по IPC (вместо статической компоновки) нет необходимости вносить изменения в код клиентской сущности.

Подробнее см. ["IPC и транспорт"](#).

## Контроль взаимодействий

В [кибериммунной системе](#) каждая попытка взаимодействия между изолированными частями системы (доменами безопасности в терминах MILS) проверяется на соответствие определенным правилам, которые задаются политикой безопасности системы. Если взаимодействие запрещено текущей политикой, оно не пропускается. Ниже мы рассмотрим, как данный принцип реализован в KasperskyOS.

## Модуль безопасности Kaspersky Security Module

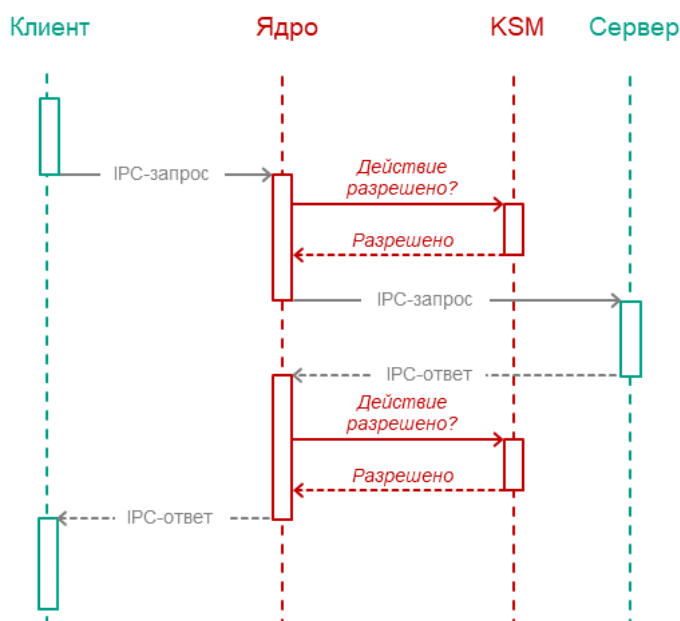
Технология Kaspersky Security System позволяет под каждое разрабатываемое решение сгенерировать код модуля безопасности Kaspersky Security Module, реализующего заданную политику безопасности. Модуль безопасности контролирует все взаимодействия между сущностями, т.е. является монитором безопасности в терминах MILS.

### Контроль IPC-сообщений

Ядро обращается к модулю безопасности Kaspersky Security Module каждый раз, когда одна сущность отправляет сообщение другой сущности. Модуль безопасности проверяет, что структура сообщения соответствует описанию вызываемого метода. Если это так, модуль безопасности вызывает все связанные с этим сообщением правила и выносит решение: "разрешено" или "запрещено". Ядро применяет полученное решение, т.е. доставляет сообщение, только если это разрешено.

Таким образом, код, вычисляющий решения (Policy Decision Point) отделен от кода, применяющего их (Policy Enforcement Point).

IPC-ответы подлежат контролю, как и IPC-запросы. Это может использоваться, например, чтобы гарантировать целостность ответов.



Ядро доставляет IPC-сообщение, только если Kaspersky Security Module разрешает доставку

## Контроль запуска сущностей

Ядро также обращается к модулю безопасности каждый раз, когда происходит запуск сущностей. Обычно это используется, чтобы инициализировать контекст безопасности сущности.

## Интерфейс безопасности

Сущности могут напрямую обращаться к Kaspersky Security Module, с помощью *интерфейса безопасности*, чтобы сообщить о своем состоянии или о контексте какого-либо события. При этом модуль безопасности применяет все правила, связанные с вызываемым методом интерфейса безопасности. Далее сущность может сама использовать полученное от модуля безопасности решение. Интерфейс безопасности используется, например, для реализации [сущностей, управляющих доступом к ресурсам](#).

## Язык описания политик PSL

Важнейшая часть технологии Kaspersky Security System – это язык PSL (Policy Specification Language). Он позволяет формально, близко к терминам самой задачи, описать политику безопасности решения. На основе полученного psf-описания генерируется код модуля безопасности Kaspersky Security Module под конкретное решение. Для этого используется компилятор NK, поставляемый в составе KasperskyOS Community Edition. Таким образом, psf-описание решения является связующим звеном между неформальным описанием политики и ее реализацией.

Основная часть psf-описания представляет собой связывание событий (отправка и получение IPC-сообщения, запуск сущности или обращение по интерфейсу безопасности) с наборами *политик* (т.е. правил). Событие будет разрешено политикой безопасности, только если все связанные с ним политики вернут решение "разрешено". Политики могут использовать и изменять контексты безопасности субъекта и объекта взаимодействия. Для связывания можно использовать различные атрибуты событий, например, вызываемый метод, реализация интерфейса или экземпляр компонента, к которому выполняется запрос.

Язык PSL позволяет использовать различные структуры данных и комбинировать несколько моделей безопасности.

Подробнее см. ["Структура файла security.psf"](#).

## Контроль доступа к ресурсам

### Виды ресурсов

Все ресурсы в KasperskyOS разделяются на системные и пользовательские:

- *Системные ресурсы* управляются ядром. К ним относятся, например, регионы памяти, прерывания или IPC-каналы.
- *Пользовательские ресурсы* управляются специальными сущностями (поставщиками ресурсов). Примеры пользовательских ресурсов: файлы, устройства ввода-вывода, сетевые интерфейсы.

Возможности Kaspersky Security Module и языка PSL позволяют контролировать использование сущностями как системных, так и пользовательских ресурсов.

### Дескрипторы

Как системные, так и пользовательские ресурсы идентифицируются при помощи *дескрипторов* (handles). Дескрипторы имеют тип `Handle`. Получая дескриптор, сущность получает доступ к соответствующему ресурсу, который этот дескриптор идентифицирует.

Не все ресурсы в KasperskyOS идентифицируются с помощью дескрипторов. Например, идентификатор потока (TID) не является дескриптором.

Дескрипторы в KasperskyOS локальны по отношению к сущности. Это значит, что каждая сущность получает дескрипторы из своего пространства дескрипторов независимо от других сущностей. Таким образом, разные сущности могут иметь одинаковые дескрипторы, которые идентифицируют разные ресурсы, а также разные дескрипторы, которые идентифицируют один и тот же ресурс.

### Доступ к системным ресурсам

С точки зрения модуля безопасности, ядро является отдельной сущностью, реализующей описанные на языке IDL интерфейсы. Обращения к интерфейсам ядра так же контролируются модулем безопасности. Поэтому доступ сущностей к системным ресурсам можно ограничивать и гибко контролировать через единую политику безопасности решения, используя все возможности [языка PSL](#).

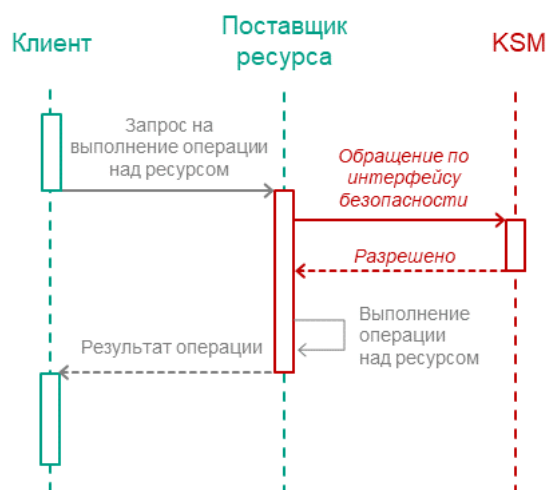
## Интерфейс безопасности и поставщики ресурсов

Ключевую роль при работе с пользовательскими ресурсами играют *поставщики ресурсов* – сущности, вводящие новые виды ресурсов в систему. Доступ к пользовательскому ресурсу возможен только путем обращения к поставщику этого ресурса. При этом у модуля безопасности нет данных о внутреннем контексте ресурса. Поэтому, информации, содержащейся в IPC-запросах к поставщику ресурса, может быть недостаточно, чтобы вынести решение о возможности доступа к ресурсу. Для решения этой проблемы используется *интерфейс безопасности* (security-интерфейс).

Интерфейс безопасности – это механизм, позволяющий сущностям напрямую обращаться к Kaspersky Security Module для вызова правил и вычисления решения.

Поставщики ресурсов используют интерфейс безопасности, чтобы предоставить модулю безопасности дополнительный контекст, связанный с операцией над ресурсом (например, права на папку, которые есть у клиента). Модуль безопасности вычисляет решение и возвращает его поставщику ресурса. И уже сам поставщик (а не ядро) должен применить это решение, т.е. не выполнять операцию с ресурсом, если она запрещена модулем безопасности.

Таким образом, интерфейс безопасности позволяет отделить политику безопасности от реализации сущности даже тогда, когда эта сущность управляет доступом к пользовательскому ресурсу. Поставщик ресурса предоставляет механизм контроля, а политика по-прежнему определяется модулем безопасности.



Поставщик ресурса напрямую обращается к модулю безопасности и затем применяет полученное решение

## Надежность TCB

[Кибериммунная система](#) должна предоставлять гарантии надежности всей доверенной вычислительной базы (TCB). Такие гарантии могут быть предоставлены, только если доверенная вычислительная база достаточно компактна. Ниже мы рассмотрим, как это требование реализуется в решениях на базе KasperskyOS.

## Микроядерная архитектура

Основой доверенной вычислительной базы любого решения является ядро. Ядро KasperskyOS предоставляет всего [три системных вызова](#) и выполняет только небольшое число наиболее важных функций, включая изоляцию и взаимодействие сущностей, планирование и управление памятью. Благодаря этому ядро компактно и имеет малую поверхность атаки, что минимизирует число потенциальных уязвимостей.

В то же время драйверы устройств и поставщики ресурсов (например, реализации файловых систем) представляют собой пользовательские приложения. Потенциальные ошибки в них не могут повлиять на стабильность работы ядра. Более того, в решении на базе KasperskyOS драйвер устройства или поставщик ресурса потенциально может быть недоверенным. Это уменьшает доверенную вычислительную базу решения и увеличивает ее надежность.

Сочетание микроядерной архитектуры и [модуля безопасности](#) позволяет контролировать все взаимодействия между драйвером (или поставщиком ресурса) и другими сущностями, а также все взаимодействия с ядром на соответствие заданной политике безопасности решения.

Вызов служб ядра KasperskyOS (например, для создания потока или выделения памяти) выполняется с помощью того же механизма IPC и того же системного вызова `Call()`, что и вызов методов другой сущности. С этой точки зрения ядро KasperskyOS предстает отдельной сущностью, которая реализует интерфейсы, описанные на языке IDL.

## Надежность доверенных компонентов

Доверенная вычислительная база решения может включать в себя, помимо микроядра KasperskyOS и модуля безопасности, разнообразные доверенные компоненты. В зависимости от целей и предположений безопасности, к доверенным компонентам могут относиться, например, драйверы устройств или поставщики ресурсов. Архитектура и набор инструментов KasperskyOS позволяют повысить надежность доверенных компонентов.

### Вынос доверенного компонента в отдельную сущность

Разработчик решения может повысить надежность TCB, уменьшив объем доверенных компонентов. Для этого их следует отделить от остального (недоверенного) кода, т.е. вынести в отдельные сущности. В составе KasperskyOS Community Edition поставляются [транспортные библиотеки и инструменты генерации транспортного кода](#), позволяющие почти любой компонент реализовать как отдельную сущность, все взаимодействия с которой контролируются.

### Создание дубликатов компонента

Ещё один способ повысить надежность TCB – это ограничение влияния недоверенных компонентов на доверенные путем разделения их информационных потоков. Для этого один и тот же компонент может быть независимо использован в составе нескольких сущностей. Например, за реализации файловых систем и сетевого стека в KasperskyOS отвечает компонент VFS. Если включить экземпляры VFS в разные сущности, каждая из них будет работать с собственной реализацией файловой системы и/или сетевого стека. Таким образом достигается разделение информационных потоков доверенных и недоверенных сущностей и, соответственно, увеличение надежности TCB.

Способ разделения пользовательского кода на доверенный и недоверенный зависит от целей и предположений безопасности конкретного решения.

## Образ решения на базе KasperskyOS

Загружаемый образ конечного решения на базе KasperskyOS содержит:

- образ ядра KasperskyOS;
- [модуль безопасности](#);
- [сущность Einit](#), которая запускает остальные сущности;
- все сущности, входящие в решение (включая драйверы, служебные сущности и сущности, реализующие бизнес-логику).

Все сущности расположены в образе файловой системы ROMFS.

## Сборка образа решения

Образ решения собирается при помощи [специального скрипта](#), который поставляется в составе KasperskyOS Community Edition.

Образ ядра KasperskyOS, а также служебные сущности и сущности-драйверы поставляются в составе KasperskyOS Community Edition. Модуль безопасности и сущность [Einit](#) собираются под каждое конкретное решение.

## Запуск решения

Запуск решения происходит следующим образом:

1. Загрузчик загружает ядро KasperskyOS.
2. Ядро находит и загружает модуль безопасности.
3. Ядро запускает сущность [Einit](#).
4. Сущность [Einit](#) запускает все остальные сущности, входящие в решение.

## Начало работы

Этот раздел содержит информацию, необходимую для начала работы с KasperskyOS Community Edition.

## Установка и удаление

### Установка

KasperskyOS Community Edition поставляется в виде deb-пакета. Для установки KasperskyOS Community Edition мы рекомендуем использовать установщик пакетов `gdebi`.

Для развертывания пакета с помощью `gdebi` запустите с root-правами команду:

```
$ gdebi <путь-к-deb-пакету>
```

Пакет будет установлен в директорию `/opt/KasperskyOS-Community-Edition-<version>`.

Для удобства работы вы можете добавить путь к бинарным файлам инструментов KasperskyOS Community Edition в переменную `PATH`, это позволит работать с утилитами через терминал из любой директории:

```
$ export PATH=$PATH:/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin
```

### Удаление

Для удаления KasperskyOS Community Edition выполните с root-правами команду:

```
$ sudo apt-get remove --purge kasperskyos-community-edition
```

При этом будут удалены все установленные файлы в директории `/opt/KasperskyOS-Community-Edition-<version>`.

## Настройка среды разработки

В этом разделе содержится краткое руководство по настройке среды разработки и добавлению заголовочных файлов, поставляемых в KasperskyOS Community Edition, в проект разработки.

### Настройка редактора кода

Для упрощения процесса разработки решений на базе KasperskyOS перед началом работы рекомендуется:

- Установить в редакторе кода расширения и плагины для используемых языков программирования (C и/или C++).




- Добавить заголовочные файлы, поставляемые в KasperskyOS Community Edition, в проект разработки. Заголовочные файлы расположены в следующей директории: `/opt/KasperskyOS-Community-Edition-<version>/sysroot-arm-kos/include`.

## Пример настройки Visual Studio Code

Например, работа с исходным кодом при разработке под KasperskyOS может проводиться в Visual Studio Code.

Для более удобной навигации по коду проекта, включая системный API, необходимо выполнить следующие действия:

1. Создайте новый проект или откройте существующий проект в Visual Studio Code.
2. Убедитесь, что расширение [C/C++ for Visual Studio Code](#)  установлено.
3. В меню **View** выберите пункт **Command Palette**.
4. Выберите пункт **C/C++: Edit Configurations (UI)**.
5. В поле **Include path** добавьте путь `/opt/KasperskyOS-Community-Edition-<version>/sysroot-arm-kos/include`.
6. Закройте окно **C/C++ Configurations**.

## Сборка и запуск примеров

### Сборка примеров

Сборка примеров осуществляется с помощью системы сборки **CMake**, входящей в состав KasperskyOS Community Edition.

Код примеров и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples
```

Сборку примеров нужно выполнять в домашней директории, поэтому директорию с примером, который требуется собрать, нужно скопировать из `/opt/KasperskyOS-Community-Edition-<version>/examples` в домашнюю директорию.

### Сборка примеров для запуска на QEMU

Чтобы выполнить сборку примера, перейдите в директорию с примером и выполните команду:

```
$ sudo ./cross-build.sh
```

В результате работы скрипта `cross-build.sh` создается образ решения на базе KasperskyOS, который включает пример. Файл образа решения `kos-qemu-image` сохраняется в директории `<название примера>/build/einit`.

## Сборка примеров для запуска на Raspberry Pi 4 B

Чтобы выполнить сборку примера:

1. Перейдите в директорию с примером.
2. Откройте файл скрипта `cross-build.sh` в текстовом редакторе.
3. В последней строке скрипта замените команду `make sim` на команду `make kos-image`.
4. Сохраните файл скрипта, а затем выполните команду:

```
$ sudo ./cross-build.sh
```

В результате работы скрипта `cross-build.sh` создается образ решения на базе KasperskyOS, который включает пример. Файл образа решения `kos-image` сохраняется в директории `<название примера>/build/einit`.

## Запуск примеров на QEMU

### Запуск примеров на QEMU в Linux с графической оболочкой

Запуск примера на QEMU в Linux с графической оболочкой осуществляется скриптом `cross-build.sh`, который также выполняет [сборку примера](#). Чтобы запустить скрипт, перейдите в директорию с примером и выполните команду:

```
$ sudo ./cross-build.sh
```

### Запуск примеров на QEMU в Linux без графической оболочки

Чтобы запустить пример на QEMU в Linux без графической оболочки, перейдите в директорию с примером, [соберите пример](#) и выполните следующие команды:

```
$ cd build/einit
# Перед выполнением следующей команды убедитесь, что путь к
# директории с исполняемым файлом qemu-system-arm сохранен в
# переменной окружения PATH. В случае отсутствия
# добавьте его в переменную PATH.
$ qemu-system-arm -m 2048 -machine vexpress-a15 -nographic -monitor none -serial stdio
-kernel kos-qemu-image
```

# Запуск примеров на Raspberry Pi 4 B

## Коммутация компьютера и Raspberry Pi 4 B

Чтобы видеть вывод примеров на компьютере, выполните следующие действия:

1. Соедините пины преобразователя USB-UART на базе FT232 с соответствующими GPIO-пинами Raspberry Pi 4 B (см. рис. ниже).

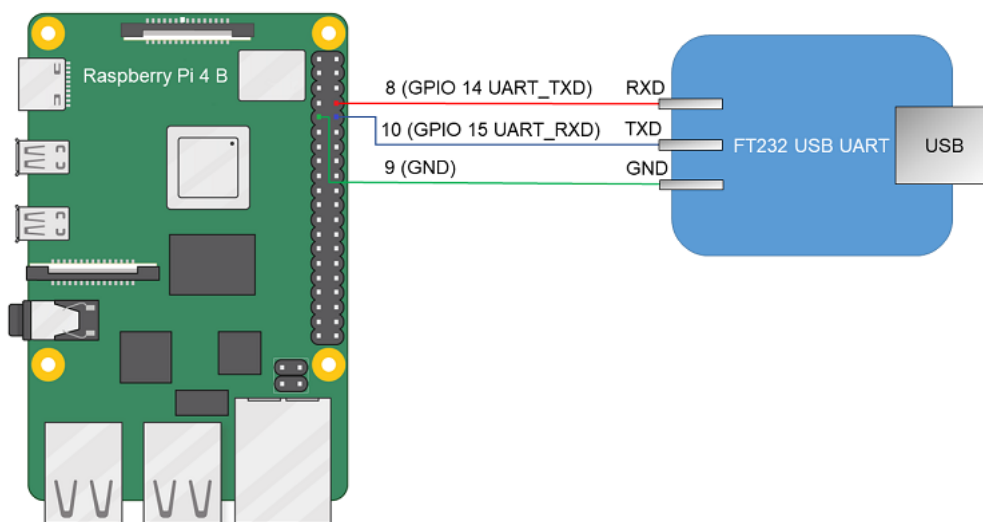


Схема соединения преобразователя USB-UART и Raspberry Pi 4 B

2. Соедините USB-порт компьютера и преобразователь USB-UART.
3. Установите PuTTY или другую аналогичную программу для чтения данных из COM-порта. Настройте параметры следующим образом: `bps = 115200`, `data bits = 8`, `stop bits = 1`, `parity = none`, `flow control = none`.

Чтобы компьютер и Raspberry Pi 4 B могли взаимодействовать через сеть Ethernet, выполните следующие действия:

1. Соедините сетевые карты компьютера и Raspberry Pi 4 B с коммутатором или друг с другом.
2. Выполните настройку сетевой карты компьютера, чтобы ее IP-адрес был в одной подсети с IP-адресом сетевой карты Raspberry Pi 4 B (параметры сетевой карты Raspberry Pi 4 B задаются в файле `dhcpcd.conf`, который находится по пути `<название примера>/resources/...`).

## Подготовка загрузочной SD-карты для Raspberry Pi 4 B

Загрузочную SD-карту для Raspberry Pi 4 B можно можно подготовить автоматически и вручную.

Чтобы подготовить загрузочную SD-карту автоматически, подключите SD-карту к компьютеру и выполните следующие команды:

```
# Следующая команда создает файл образа загрузочного
# носителя (*.img).
$ sudo /opt/KasperskyOS-Community-Edition-<version>/examples/rpi4_prepare_fs_image.sh
```

```
# В следующей команде path_to_img - путь к файлу образа
# загрузочного носителя (этот путь выводится по окончании
# выполнения предыдущей команды), [X] - последний символ
# в имени блочного устройства для SD-карты.
$ sudo dd bs=4M if=path_to_img of=/dev/sd[X] conv=fsync
```

Чтобы подготовить загрузочную SD-карту вручную, выполните следующие действия:

1. Выполните сборку загрузчика U-Boot для платформы ARMv7, который будет автоматически запускать пример. Для этого выполните следующие команды:

```
$ sudo apt install gcc-arm-linux-gnueabi gcc-arm-linux-gnueabihf git bison flex
$ git clone https://github.com/u-boot/u-boot.git u-boot-armv7
$ cd u-boot-armv7 && git checkout tags/v2020.10
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- rpi_4_32b_defconfig
# В меню, которое появится при выполнении следующей команды, включите
# флаг Enable a default value for bootcmd. В поле bootcmd value введите
# значение fatload mmc 0 ${loadaddr} kos-image; bootelf ${loadaddr}.
# Затем выйдите из меню, сохранив параметры.
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- u-boot.bin
```

2. Отформатируйте SD-карту. Для этого подключите SD-карту к компьютеру и выполните следующие команды:

```
$ wget https://downloads.raspberrypi.org/raspbian_lite_latest
$ unzip raspbian_lite_latest
# В следующей команде [X] - последний символ в имени блочного устройства
# для SD-карты.
$ sudo dd bs=4M if=$(ls *raspbian*lite.img) of=/dev/sd[X] conv=fsync
```

3. Скопируйте загрузчик U-Boot на SD-карту, выполнив следующие команды:

```
# В следующих командах путь ~/mnt/fat32 используется для примера. Вы
# можете использовать другой путь.
$ mkdir -p ~/mnt/fat32
# В следующей команде [X] - последний буквенный символ в имени блочного
# устройства для раздела на отформатированной SD-карте.
$ sudo mount /dev/sd[X]1 ~/mnt/fat32/
$ sudo cp u-boot.bin ~/mnt/fat32/u-boot.bin
```

4. Скопируйте конфигурационный файл для загрузчика U-Boot на SD-карту. Для этого перейдите в директорию /opt/KasperskyOS-Community-Edition-<version>/examples и выполните следующие команды:

```
$ sudo cp config.txt ~/mnt/fat32/config.txt
$ sync
$ sudo umount ~/mnt/fat32
```

Запуск примера на Raspberry Pi 4 B

Чтобы запустить пример на Raspberry Pi 4 B, выполните следующие действия:

1. Перейдите в директорию с примером и [соберите пример](#).
2. Скопируйте на загрузочную SD-карту образ решения на базе KasperskyOS. Для этого подключите загрузочную SD-карту к компьютеру и выполните следующие команды:

```
# В следующей команде [X] – последний буквенный символ в имени блочного
# устройства для раздела на загрузочной SD-карте.
# В следующих командах путь ~/mnt/fat32 используется для примера. Вы
# можете использовать другой путь.
$ sudo mount /dev/sd[X]1 ~/mnt/fat32/
$ sudo cp build/einit/kos-image ~/mnt/fat32/kos-image
$ sync
$ sudo umount ~/mnt/fat32
```

3. Подключите загрузочную SD-карту к Raspberry Pi 4 B.
4. Подайте питание на Raspberry Pi 4 B и дождитесь, пока запустится пример.  
О том, что пример запустился, свидетельствует вывод, отображаемый на компьютере.

## Часть 1. Простейшее приложение (POSIX)

В состав KasperskyOS Community Edition входит набор библиотек (`libc`, `libm`, `libpthread`), которые обеспечивают частичную совместимость разрабатываемых приложений с набором стандартов POSIX.

В этой части руководства рассматриваются:

- вывод строки на экран с помощью `fprintf()`;
- использование компонента VFS для работы с сетью и файловыми системами;
- создание инициализирующей сущности `Einit`;
- ограничения поддержки POSIX.

Чтобы сделать изложение более простым, пример в этой части руководства собирается без модуля `ksm.module`. Поэтому при запуске примера выдается предупреждение `WARNING! Booting an insecure kernel!`. Политика безопасности решения, использование политик безопасности и сборка модуля `ksm.module` рассматриваются в [третьей части руководства](#).

### Пример hello

По сложившейся традиции в мире разработки ПО, первое, с чего стоит начать знакомство с любой технологией – это поприветствовать с помощью нее мир. Не будем нарушать эту традицию в KasperskyOS и начнем с примера, выводящего на экран строку `Hello world!`.

KasperskyOS позволяет разрабатывать решения как на языке C, так и на C++.

Код `hello.c` выглядит привычным и простым для разработчика на языке C – он полностью совместим с POSIX:

```
hello.c

#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char *argv[])
{
    fprintf(stderr, "Hello world!\n");

    return EXIT_SUCCESS;
}
```

Чтобы запустить файл `Hello` в KasperskyOS, понадобится несколько дополнительных действий.

Разработка приложений под KasperskyOS имеет следующие особенности:

Во-первых, каждая *сущность* (так в KasperskyOS называются приложения и соответствующие им процессы) должна быть статически описана. Описание содержится в файлах с расширениями edl, cdl и idl, которые используются для сборки решения. Минимально возможное описание сущности – EDL-файл, в котором указано имя сущности. Все сущности, разрабатываемые в первой части руководства, имеют минимальное статическое описание (только EDL-файл с именем сущности). Во-вторых, все запускаемые сущности должны содержаться в загружаемом образе KasperskyOS. Поэтому каждый пример в этом руководстве представляет собой не отдельную сущность (сущности), а готовое *решение* на базе KasperskyOS, включающее в себя образ ядра, инициализирующую сущность и вспомогательные сущности (например, драйверы).

## EDL-описание сущности Hello

Статическое описание сущности `Hello` состоит из единственного файла `Hello.edl`, в котором необходимо прописать имя сущности:

`Hello.edl`

```
/* После ключевого слова "entity" указано имя сущности. */  
entity Hello
```

Имя сущности должно начинаться с заглавной буквы. Имя EDL-файла должно совпадать с именем сущности, которую он описывает.

Во второй части руководства показаны примеры более сложных EDL-описаний, а также появляются CDL- и IDL-описания.

## Создание инициализирующей сущности Einit

При загрузке KasperskyOS ядро запускает сущность с именем `Einit`. Сущность `Einit` запускает все остальные сущности, входящие в решение, то есть служит *инициализирующей сущностью*. В составе пакета инструментов KasperskyOS Community Edition поставляется утилита `einit`, которая позволяет сгенерировать код инициализирующей сущности (`einit.c`) на основе *init-описания*. В приведенном ниже примере файл с *init-описанием* называется `init.yaml`, хотя может иметь любое имя. Подробнее см. "Запуск сущностей".

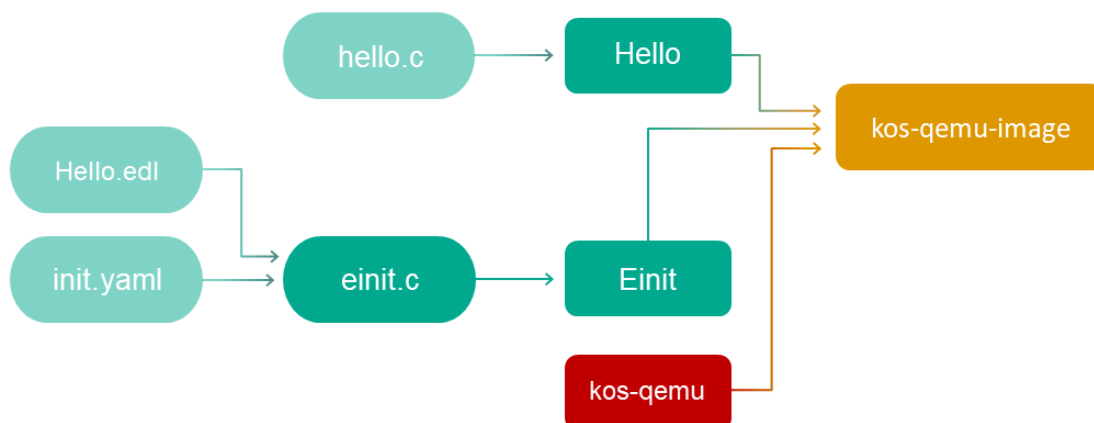
Для того чтобы сущность `Hello` запустилась после загрузки операционной системы, достаточно указать ее имя в файле `init.yaml` и собрать из него сущность `Einit`.

`init.yaml`

```
entities:  
# Запустить сущность "Hello".  
- name: Hello
```

## Схема сборки примера hello

Общая схема сборки примера `hello` выглядит следующим образом:



## Сборка и запуск примера hello

См. ["Сборка и запуск примеров"](#).

## VFS: работа с файлами и сетью

### VFS: обзор

Компонент VFS содержит в себе реализации файловых систем и сетевого стека. POSIX-вызовы для работы с файловыми системами и сетью направляются в компонент VFS, который далее вызывает драйвер блочного устройства, или, соответственно, сетевой драйвер.

Клиент → VFS → Драйвер

В решение можно добавить несколько копий компонента VFS, разделив таким образом информационные потоки разных сущностей. Каждая копия VFS собирается отдельно и может содержать всю функциональность VFS или конкретную ее часть, например:

- одну или несколько файловых систем;
- сетевой стек;
- сетевой стек и сетевой драйвер.

Компонент VFS можно использовать как напрямую (путем статической компоновки), так и через IPC (как отдельную сущность). Использование функциональности VFS по IPC позволяет разработчику решения:



- контролировать вызовы методов для работы с сетью и файловыми системами через [политику безопасности решения](#);
- соединить несколько клиентских сущностей с одной сущностью VFS;
- соединить одну клиентскую сущность [с двумя сущностями VFS](#) для раздельной работы с сетью и файловыми системами.

## Сборка сущности VFS

В составе KasperskyOS Community Edition не поставляется готовый образ сущности, содержащей компонент VFS. Разработчик решения может самостоятельно собрать одну или несколько сущностей, включив в каждую из них именно ту функциональность VFS, которая необходима в решении.

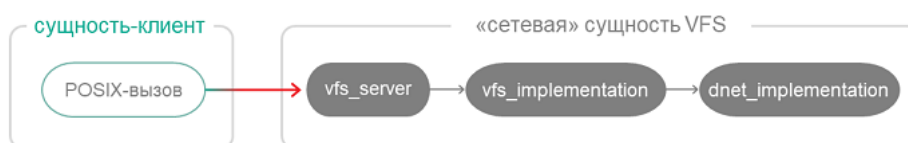
См. также примеры [multi\\_vfs\\_dhcpd](#), [multi\\_vfs\\_dns\\_client](#) и [multi\\_vfs\\_ntpd](#) в составе KasperskyOS Community Edition.

## Сборка сетевой VFS

Для сборки "сетевой" сущности VFS, содержащей сетевой драйвер, необходимо файл с функцией `main()` скомпоновать с библиотеками `vfs_server`, `vfs_net` и `dnet_implementation`:

CMakeLists.txt (фрагмент)

```
target_link_libraries (Net1Vfs ${vfs_SERVER_LIB}
                           ${vfs_NET_LIB}
                           ${dnet_IMPLEMENTATION_LIB})
set_target_properties (Net1Vfs PROPERTIES ${blkdev_ENTITY}_REPLACEMENT "")
```

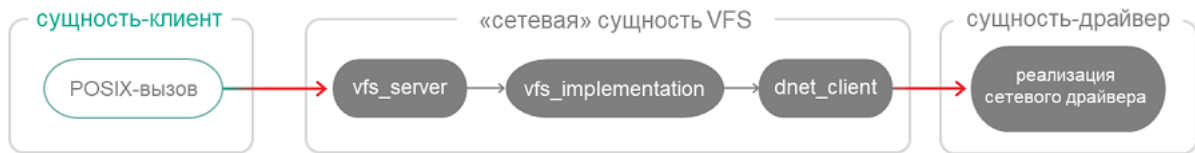


Использование "сетевой" сущности VFS, скомпонованной с сетевым драйвером

Чтобы использовать сетевой драйвер по IPC (как отдельную сущность), необходимо вместо `dnet_implementation` использовать библиотеку `dnet_client`:

CMakeLists.txt (фрагмент)

```
target_link_libraries (Net2Vfs ${vfs_SERVER_LIB}
                           ${vfs_NET_LIB}
                           ${dnet_CLIENT_LIB})
set_target_properties (Net2Vfs PROPERTIES ${blkdev_ENTITY}_REPLACEMENT "")
```



Использование "сетевой" сущности VFS и сетевого драйвера в виде отдельной сущности

Некоторые сетевые функции, а также вывод в `stdout` используют файловые операции. Для корректной работы этих функций требуется при сборке добавить библиотеку `vfs_implementation` вместо `vfs_net`.

## Сборка файловой VFS

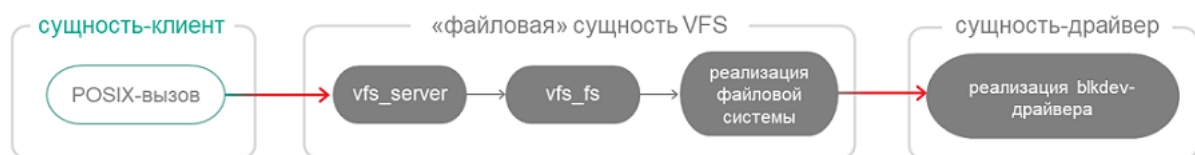
Для сборки "файловой" сущности VFS необходимо файл с функцией `main()` скомпоновать с библиотеками `vfs_server` и `vfs_fs`, а также библиотеками реализации файловых систем:

CMakeLists.txt (фрагмент)

```
target_link_libraries (VfsFs
    ${vfs_SERVER_LIB}
    ${LWEXT4_LIB}
    ${vfs_FS_LIB})
set_target_properties (VfsFs PROPERTIES ${blkdev_ENTITY}_REPLACEMENT
    ${ramdisk_ENTITY})
```

В этом примере сущность VFS подготовлена для соединения с сущностью `ramdisk`-драйвера.

Драйвер блочного устройства не может быть скомпонован с VFS и всегда используется через IPC:



Использование "файловой" сущности VFS и драйвера в виде отдельной сущности

При необходимости можно собрать сущность VFS, содержащую как сетевой стек, так и файловые системы. Для этого необходимо использовать библиотеки `vfs_server`, `vfs_implementation`, `dnet_implementation` (или `dnet_client`), а также библиотеки реализации файловых систем.

## Сущность Env

Служебная сущность `Env` предназначена для передачи запускаемым сущностям аргументов и переменных окружения. При запуске каждая сущность автоматически отправляет запрос сущности `Env` и получает необходимые данные.

Включение сущности Env в решение позволяет [монтировать файловые системы](#) при запуске VFS, соединять одну клиентскую сущность [с двумя сущностями VFS](#) и решать многие другие задачи.

Чтобы использовать сущность Env в своем решении, необходимо:

1. Разработать код сущности Env, используя макросы из `env/env.h`.
2. Собрать образ сущности, скомпоновав ее с библиотекой `env_server`.
3. В init-описании указать, что необходимо запустить сущность Env и соединить с ней выбранные сущности (Env при этом является сервером). Имя канала задается макросом `ENV_SERVICE_NAME`, объявленным в файле `env/env.h`.
4. Включить образ сущности Env в образ решения.

## Код сущности Env

В коде сущности Env используются следующие макросы и функции, объявленные в файле `env/env.h`:

- `ENV_REGISTER_ARGS(name, argarr)` – сущности с именем `name` будут переданы аргументы из массива `argarr`;
- `ENV_REGISTER_VARS(name, envarr)` – сущности с именем `name` будут переданы переменные окружения из массива `envarr`;
- `ENV_REGISTER_PROGRAM_ENVIRONMENT(name, argarr, envarr)` – сущности с именем `name` будут переданы как аргументы, так и переменные окружения;
- `envServerRun()` – инициализировать серверную часть сущности, чтобы она могла отвечать на запросы.

Пример:

`env.c`

```
#include <env/env.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    const char* NetVfsArgs[] = {
        "-l", "devfs /dev devfs 0",
        "-l", "romfs /etc romfs 0"
    };
    ENV_REGISTER_ARGS("VFS", NetVfsArgs);

    envServerRun();
    return EXIT_SUCCESS;
}
```

## Пример init.yaml для использования сущности Env

В следующем примере сущность Client будет соединена с сущностью Env, образ которой находится в папке `env`:

init.yaml

**entities:**

- name: env.Env
- name: Client  
connections:
  - target: env.Env  
id: {var: ENV\_SERVICE\_NAME, include: env/env.h}

## Соединение сущности-клиента с одной и двумя сущностями VFS

Вызовы сетевых и файловых POSIX-функций можно перенаправить в два отдельных компонента VFS, соединив сущность-клиент с двумя разными сущностями VFS. Если такое разделение информационных потоков не требуется (например, если клиент работает только с сетью), достаточно соединить сущность-клиент с одной сущностью VFS.

### Соединение с одной сущностью VFS

Имя IPC-канала между сущностью-клиентом и сущностью VFS должно задаваться макросом `_VFS_CONNECTION_ID`, объявленным в файле `vfs/defs.h`. При этом как "сетевые", так и "файловые" вызовы будут направляться в эту сущность VFS.

Пример:

init.yaml

- name: ClientEntity  
connections:
  - target: VfsEntity  
id: {var: \_VFS\_CONNECTION\_ID, include: vfs/defs.h}
- name: VfsEntity

### Соединение с двумя сущностями VFS

Пусть сущность-клиент соединена с двумя разными сущностями VFS – назовем их "сетевая" VFS и "файловая" VFS.

Чтобы "сетевые" вызовы из сущности-клиента направлялись только в "сетевую" VFS, используется переменная окружения `_VFS_NETWORK_BACKEND`:

- Для "сетевой" сущности VFS: `_VFS_NETWORK_BACKEND=server:<имя IPC-канала до "сетевой" VFS>`
- Для сущности-клиента: `_VFS_NETWORK_BACKEND=client: <имя IPC-канала до "сетевой" VFS>`

Для направления "файловых" вызовов используется аналогичная переменная окружения `_VFS_FILESYSTEM_BACKEND`:

- Для "файловой" сущности VFS: `_VFS_FILESYSTEM_BACKEND=server:<имя IPC-канала до "файловой" VFS>`
- Для сущности-клиента: `_VFS_FILESYSTEM_BACKEND=client: <имя IPC-канала до "файловой" VFS>`

В результате функции для работы с сетью и файлами будут направляться в две разные сущности VFS.

В следующем примере сущность `Client` соединена с двумя сущностями VFS – "сетевой" `VfsFirst` и "файловой" `VfsSecond`:

init.yaml

#### entities:

- name: Env
- name: Client
  - connections:
    - target: Env
      - id: {var: ENV\_SERVICE\_NAME, include: env/env.h}
    - target: VfsFirst
      - id: VFS1
    - target: VfsSecond
      - id: VFS2
- name: VfsFirst
  - connections:
    - target: Env
      - id: {var: ENV\_SERVICE\_NAME, include: env/env.h}
- name: VfsSecond
  - connections:
    - target: Env
      - id: {var: ENV\_SERVICE\_NAME, include: env/env.h}

Код [сущности Env](#):

env.c

```
#include <env/env.h>
#include <stdlib.h>

int main(void)
{
    const char* vfs_first_args[] = { "_VFS_NETWORK_BACKEND=server:VFS1" };
    ENV_REGISTER_VARS("VfsFirst", vfs_first_args);

    const char* vfs_second_args[] = { "_VFS_FILESYSTEM_BACKEND=server:VFS2" };
    ENV_REGISTER_VARS("VfsSecond", vfs_second_args);

    const char* client_envs[] = { "_VFS_NETWORK_BACKEND=client:VFS1" };
    ENV_REGISTER_VARS("Client", client_envs);

    envServerRun();

    return EXIT_SUCCESS;
}
```

## Монтирование файловых систем при запуске VFS

Компонент VFS по умолчанию обеспечивает доступ к:

- Файловой системе RAMFS. По умолчанию RAMFS монтирована в корневой каталог.
- Объектному ROMFS-хранилищу. Хранилище содержит неисполняемые (в т.ч. конфигурационные) файлы, добавленные при сборке в образ решения. По умолчанию файловая система ROMFS не монтирована, но доступ к хранилищу возможен косвенным образом – например, через аргумент `-f`.

Если требуется монтировать другие файловые системы, это можно сделать как после запуска VFS, используя вызов `mount()`, так и непосредственно в момент запуска сущности VFS, передав ей следующие аргументы и переменные окружения:

- `-l <запись в формате fstab>`

Аргумент `-l` позволяет монтировать файловую систему.

- `-f <путь к файлу fstab>`

Аргумент `-f` позволяет передать файл с записями в формате `fstab` для монтирования файловых систем. Файл будет искаться в ROMFS-хранилище. Если переменная `UNMAP_ROMFS` определена, то файл будет искаться на файловой системе, смонтированной с помощью переменной `ROOTFS`.

- `UNMAP_ROMFS`

Если переменная `UNMAP_ROMFS` определена, то ROMFS-хранилище будет удалено. Это позволяет сэкономить память и изменить поведение при использовании аргумента `-f`.

- `ROOTFS = <запись в формате fstab>`

Переменная `ROOTFS` позволяет монтировать файловую систему в корневой каталог. В комбинации с переменной `UNMAP_ROMFS` и аргументом `-f` позволяет искать `fstab`-файл на смонтированной файловой системе, а не в ROMFS-хранилище.

Пример:

env.c

```
#include <env/env.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    /* Для сущности Vfs1 будут монтированы файловые системы devfs и romfs. */
    const char* Vfs1Args[] = {
        "-l", "devfs /dev devfs 0",
        "-l", "romfs /etc romfs 0"
    };

    ENV_REGISTER_ARGS("Vfs1", Vfs1Args);
}
```

```

/* Для сущности Vfs2 будут монтированы файловые системы, заданные через файл
/etc/dhcpd.conf, который расположен в ROMFS-хранилище. */
const char* Vfs2Args[] = { "-f", "/etc/dhcpd.conf" };

ENV_REGISTER_ARGS("Vfs2", Vfs2Args);

/* Для сущности Vfs3 в корневой каталог будет монтирована файловая система ext2,
на которой будет найден файл /etc/fstab для монтирования дополнительных файловых
систем. ROMFS-хранилище будет удалено. */
const char* Vfs3Args[] = { "-f", "/etc/fstab" };

const char* Vfs3Envs[] = {
    "ROOTFS=ramdisk0,0 / ext2 0",
    "UNMAP_ROMFS=1"
};
ENV_REGISTER_PROGRAM_ENVIRONMENT("Vfs3", Vfs3Args, Vfs3Envs);

envServerRun();

return EXIT_SUCCESS;
}

```

См. также примеры [net with separate vfs](#), [net2 with separate vfs](#), [multi vfs dhcpd](#), [multi vfs dns client](#) и [multi vfs ntpd](#) в составе KasperskyOS Community Edition.

## Ограничения поддержки POSIX

В KasperskyOS ограниченно реализован интерфейс POSIX с ориентацией на стандарт POSIX.1-2008 (без поддержки XSI). Прежде всего ограничения связаны с обеспечением безопасности.

Ограничения затрагивают:

- взаимодействие между процессами;
- взаимодействие между потоками выполнения (threads) посредством сигналов;
- стандартный ввод-вывод;
- асинхронный ввод-вывод;
- использование робастных мьютексов;
- работу с терминалом;
- использование оболочки;
- манипуляции с дескрипторами файлов.

Ограничения представлены:

- нереализованными интерфейсами;

- интерфейсами, которые реализованы с отклонениями от стандарта POSIX.1-2008;
- интерфейсами-заглушками, которые не выполняют никаких действий, кроме присвоения переменной `errno` значения `ENOSYS` и возвращения значения `-1`.

В KasperskyOS сигналы не могут прервать системные вызовы `Call()`, `Recv()`, `Reply()`, которые обеспечивают работу библиотек, реализующих интерфейс POSIX. Ядро KasperskyOS не посылает сигналы.

## Ограничения взаимодействия между процессами

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
<code>fork()</code>	Создать новый (дочерний) процесс	Заглушка	<code>unistd.h</code>
<code>pthread_atfork()</code>	Зарегистрировать обработчики, которые вызываются перед и после создания дочернего процесса	Не реализован	<code>pthread.h</code>
<code>wait()</code>	Ожидать остановки или завершения дочернего процесса	Заглушка	<code>sys/wait.h</code>
<code>waitid()</code>	Ожидать изменения состояния дочернего процесса	Не реализован	<code>sys/wait.h</code>
<code>waitpid()</code>	Ожидать остановки или завершения дочернего процесса	Заглушка	<code>sys/wait.h</code>
<code>execl()</code>	Запустить исполняемый файл	Заглушка	<code>unistd.h</code>
<code>execle()</code>	Запустить исполняемый файл	Заглушка	<code>unistd.h</code>
<code>execlp()</code>	Запустить исполняемый файл	Заглушка	<code>unistd.h</code>
<code>execv()</code>	Запустить исполняемый	Заглушка	<code>unistd.h</code>



	файл		
execve()	Запустить исполняемый файл	Заглушка	unistd.h
execvp()	Запустить исполняемый файл	Заглушка	unistd.h
fexecve()	Запустить исполняемый файл	Заглушка	unistd.h
setpgid()	Перевести процесс в другую группу или создать группу	Заглушка	unistd.h
setsid()	Создать сессию	Не реализован	unistd.h
getpgrp()	Получить идентификатор группы для вызывающего процесса	Не реализован	unistd.h
getpgid()	Получить идентификатор группы	Заглушка	unistd.h
getppid()	Получить идентификатор родительского процесса	Не реализован	unistd.h
getsid()	Получить идентификатор сессии	Заглушка	unistd.h
times()	Получить значения времени для процесса и его потомков	Заглушка	sys/times.h
kill()	Послать сигнал процессу или группе процессов	Можно посылать только сигнал SIGTERM. Параметр pid игнорируется.	signal.h
pause()	Ожидать сигнала	Не реализован	unistd.h
sigpending()	Проверить наличие полученных заблокированных сигналов	Не реализован	signal.h
sigprocmask()	Получить и изменить набор заблокированных сигналов	Заглушка	signal.h
sigsuspend()	Ожидать сигнала	Заглушка	signal.h
sigwait()	Ожидать сигнала	Заглушка	signal.h

	из заданного набора сигналов		
<code>sigqueue()</code>	Послать сигнал процессу	Не реализован	<code>signal.h</code>
<code>sigtimedwait()</code>	Ожидать сигнала из заданного набора сигналов	Не реализован	<code>signal.h</code>
<code>sigwaitinfo()</code>	Ожидать сигнала из заданного набора сигналов	Не реализован	<code>signal.h</code>
<code>sem_init()</code>	Создать неименованный семафор	Нельзя создать неименованный семафор для синхронизации между процессами. Если передать функции ненулевое значение через параметр <code>pshared</code> , то она только вернет значение <code>-1</code> и присвоит переменной <code>errno</code> значение <code>ENOTSUP</code> .	<code>semaphore.h</code>
<code>sem_open()</code>	Создать/открыть именованный семафор	Нельзя открыть именованный семафор, который был создан другим процессом. Именованные семафоры (как и неименованные) являются локальными, то есть они доступны только тому процессу, который их создал.	<code>semaphore.h</code>
<code>pthread_mutexattr_setpshared()</code>	Задать атрибут мьютекса, который разрешает использования мьютекса несколькими процессами	Нельзя задать атрибут мьютекса, который разрешает использование мьютекса несколькими процессами. Если передать функции значение <code>PTHREAD_PROCESS_SHARED</code> через параметр <code>pshared</code> , то она только вернет значение <code>ENOSYS</code> .	<code>pthread.h</code>
<code>pthread_barrierattr_setpshared()</code>	Задать атрибут барьера, который разрешает использование барьера несколькими процессами	Нельзя задать атрибут барьера, который разрешает использование барьера несколькими процессами. Если передать функции значение <code>PTHREAD_PROCESS_SHARED</code> через параметр <code>pshared</code> , то она только вернет значение <code>ENOSYS</code> .	<code>pthread.h</code>
<code>pthread_condattr_setpshared()</code>	Задать атрибут условной переменной, который разрешает использование условной переменной несколькими процессами	Нельзя задать атрибут условной переменной, который разрешает использование условной переменной несколькими процессами. Если передать функции значение <code>PTHREAD_PROCESS_SHARED</code> через параметр <code>pshared</code> , то она только вернет значение <code>ENOSYS</code> .	<code>pthread.h</code>
<code>pthread_rwlockattr_setpshared()</code>	Задать атрибут объекта блокировки чтения-записи, который разрешает использование объекта	Нельзя задать атрибут объекта блокировки чтения-записи, который разрешает использование объекта блокировки чтения-записи несколькими процессами. Если передать функции значение <code>PTHREAD_PROCESS_SHARED</code> через параметр <code>pshared</code> , то она только вернет значение <code>ENOSYS</code> .	<code>pthread.h</code>

	блокировки чтения-записи несколькими процессами		
pthread_spin_init()	Создать спин-блокировку	Нельзя создать спин-блокировку для синхронизации между процессами. Если передать функции значение PTHREAD_PROCESS_SHARED через параметр pshared, то это значение будет проигнорировано.	pthread.h
shm_open()	Создать или открыть объект разделяемой памяти	Не реализован	sys/mman.h
mmap()	Отобразить в память	Нельзя выполнить отображение в память для взаимодействия между процессами. Если передать функции значение MAP_SHARED через параметр flags, то это значение будет проигнорировано. Кроме того, через параметр prot нельзя передавать сочетания флагов PROT_WRITE   PROT_EXEC и PROT_READ   PROT_WRITE   PROT_EXEC. В этом случае функция только возвращает значение MAP_FAILED и присваивает переменной errno значение ENOMEM.	sys/mman.h
mprotect()	Задать права доступа к памяти	По умолчанию функция работает как заглушка. Чтобы использовать функцию, требуется задать специальные параметры ядра KasperskyOS.	sys/mman.h
pipe()	Создать неименованный канал	Нельзя использовать неименованный канал для передачи данных между процессами. Неименованные каналы являются локальными, то есть они доступны только тому процессу, который их создал.	unistd.h
mkfifo()	Создать специальный файл FIFO (именованный канал)	Заглушка	sys/stat.h
mkfifoat()	Создать специальный файл FIFO (именованный канал)	Не реализован	sys/stat.h

## Ограничения взаимодействия между потоками выполнения посредством сигналов

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
pthread_kill()	Послать сигнал потоку	Нельзя послать сигнал потоку выполнения. Если передать функции номер сигнала	signal.h

	выполнения	через параметр <code>sig</code> , то она только вернет значение <code>ENOSYS</code> .	
<code>pthread_sigmask()</code>	Получить и изменить набор заблокированных сигналов	Заглушка	<code>signal.h</code>
<code>siglongjmp()</code>	Восстановить состояние потока управления и маску сигналов	Не реализован	<code>setjmp.h</code>
<code>sigsetjmp()</code>	Сохранить состояние потока управления и маску сигналов	Не реализован	<code>setjmp.h</code>

## Ограничения стандартного ввода-вывода

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
<code>dprintf()</code>	Выполнить форматированный вывод в файл	Не реализован	<code>stdio.h</code>
<code>fmemopen()</code>	Использовать память как поток данных (stream)	Не реализован	<code>stdio.h</code>
<code>open_memstream()</code>	Использовать динамически выделенную память как поток данных (stream)	Не реализован	<code>stdio.h</code>
<code>vdprintf()</code>	Выполнить форматированный вывод в файл	Не реализован	<code>stdio.h</code>

## Ограничения асинхронного ввода-вывода

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
<code>aio_cancel()</code>	Отменить запросы ввода-вывода, которые ожидают обработки	Не реализован	<code>aio.h</code>
<code>aio_error()</code>	Получить ошибку операции асинхронного ввода-вывода	Не реализован	<code>aio.h</code>
<code>aio_fsync()</code>	Запросить выполнение операций ввода-вывода	Не реализован	<code>aio.h</code>
<code>aio_read()</code>	Запросить чтение из файла	Не реализован	<code>aio.h</code>
<code>aio_return()</code>	Получить статус операции асинхронного ввода-вывода	Не реализован	<code>aio.h</code>
<code>aio_suspend()</code>	Ожидать выполнения операций асинхронного ввода-вывода	Не реализован	<code>aio.h</code>
<code>aio_write()</code>	Запросить запись в файл	Не	<code>aio.h</code>

		реализован	
lio_listio()	Запросить выполнение набора операций ввода-вывода	Не реализован	aio.h

## Ограничения использования робастных мьютексов

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
pthread_mutex_consistent()	Вернуть робастный мьютекс в консистентное состояние	Не реализован	pthread.h
pthread_mutexattr_getrobust()	Получить атрибут робастности мьютекса	Не реализован	pthread.h
pthread_mutexattr_setrobust()	Задать атрибут робастности мьютекса	Не реализован	pthread.h

## Ограничения работы с терминалом

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
ctermid()	Получить путь к файлу управляющего терминала	Функция только возвращает или передает через параметр s пустую строку.	stdio.h
tcsetattr()	Задать параметры терминала	Скорость ввода, скорость вывода и другие параметры, специфичные для аппаратных терминалов, игнорируются.	termios.h
tcdrain()	Ожидать завершения вывода	Функция только возвращает значение -1.	termios.h
tcflow()	Приостановить или возобновить прием или передачу данных	Приостановка вывода и запуск приостановленного вывода не поддерживаются.	termios.h
tcflush()	Очистить очередь ввода или очередь вывода, или обе эти очереди	Функция только возвращает значение -1.	termios.h
tcsendbreak()	Разорвать соединение с терминалом на заданное время	Функция только возвращает значение -1.	termios.h
ttyname()	Получить путь к файлу терминала	Функция только возвращает нулевой указатель.	unistd.h
ttyname_r()	Получить путь к файлу терминала	Функция только возвращает значение ошибки.	unistd.h
tcgetpgrp()	Получить идентификатор группы процессов, использующих терминал	Функция только возвращает значение -1.	unistd.h

tcsetpgrp()	Задать идентификатор группы процессов, использующих терминал	Функция только возвращает значение -1.	unistd.h
tcgetsid()	Получить идентификатор группы процессов для лидера сессии, связанной с терминалом	Функция только возвращает значение -1.	termios.h

## Ограничения работы с оболочкой

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
popen()	Создать дочерний процесс для выполнения команды и канал с этим процессом	Функция только присваивает переменной errno значение ENOSYS и возвращает значение NULL.	stdio.h
pclose()	Закрыть канал с дочерним процессом, созданным функцией popen(), и ожидать завершения этого дочернего процесса	Функцию нельзя использовать, так как ее входным параметром является дескриптор потока данных, возвращаемый функцией popen(), которая не может вернуть ничего, кроме значения NULL.	stdio.h
system()	Создать дочерний процесс для выполнения команды	Заглушка	stdlib.h
wordexp()	Раскрыть строку как в оболочке	Не реализован	wordexp.h
wordfree()	Освободить память, выделенную для результатов вызова интерфейса wordexp()	Не реализован	wordexp.h

## Ограничения манипуляций с дескрипторами файлов

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
dup()	Сделать копию дескриптора открытого файла	Поддерживаются дескрипторы обычных файлов, стандартных потоков ввода-вывода, сокетов и каналов. Не гарантируется, что будет получен наименьший свободный дескриптор.	fcntl.h
dup2()	Сделать копию дескриптора открытого файла	Поддерживаются дескрипторы обычных файлов, стандартных потоков ввода-вывода, сокетов и каналов. Через параметр fildes2 нужно передавать дескриптор открытого файла.	fcntl.h

## Часть 2. Взаимодействие сущностей

В предыдущей части руководства показано, как построить взаимодействие с сущностями, поставляемыми в составе KasperskyOS Community Edition. Для этого достаточно добавить несколько строк в файл `init.yaml` и подключить клиентскую библиотеку сущности (`vfs_remote`).

Но как самостоятельно создать серверную сущность (то есть приложение, предоставляющее функциональность другим, клиентским, сущностям)? Для этого потребуется использовать IPC-транспорт, вспомогательные утилиты и библиотеки, поставляемые в составе KasperskyOS Community Edition.

В этой части руководства рассматриваются:

- механизм взаимодействия сущностей в KasperskyOS;
- утилиты и библиотеки, реализующие транспорт;
- пошаговые действия для обмена IPC-сообщениями.

Чтобы сделать изложение более простым, примеры в этой части руководства собираются без модуля `ksm.module`. Поэтому при запуске примеров выдается предупреждение `WARNING! Booting an insecure kernel!`. Политика безопасности решения, использование политик безопасности и сборка модуля `ksm.module` рассматриваются в [третьей части руководства](#).

## Инструменты IPC-транспорта

Чтобы организовать взаимодействие сущностей, не требуется "с нуля" реализовывать корректную упаковку и распаковку сообщений. Кроме того, не обязательно писать отдельный код для создания IPC-каналов.

Для решения этих и других проблем организации IPC-транспорта в KasperskyOS есть специальный набор инструментов:

- Транспорт `NkKosTransport`
- EDL-, CDL- и IDL-описания
- Компилятор NK
- Init-описание и утилита `einit`
- Локатор сервисов (Service Locator)

Совместное использование этих инструментов показано в примере [echo](#).

### Транспорт `NkKosTransport`

Транспорт `NkKosTransport` является удобной надстройкой над системными вызовами `Call`, `Recv` и `Reply`. Он позволяет работать отдельно с фиксированной частью и ареной сообщений.

Для вызова транспорта используются функции `nk_transport_call()`, `nk_transport_recv()` и `nk_transport_reply()`.

Пример:

```
/* Функция nk_transport_recv () выполняет системный вызов Recv.
 * Полученный от клиента запрос помещается в req (фиксированная часть ответа) и
 * req_arena (арена ответа). */
nk_transport_recv(&transport.base, (struct nk_message *)&req, &req_arena);
```

Структура `NkKosTransport` и методы работы с ней объявлены в файле `transport-kos.h`:

```
#include <coresrv/nk/transport-kos.h>
```

Подробнее о фиксированной части и арене см. "[Структура IPC-сообщения](#)".

Подробнее об использовании `NkKosTransport` см. "[Транспорт NkKosTransport](#)".

## EDL-, CDL- и IDL-описания

Для описания интерфейсов, которые реализуют серверные сущности, используются языки EDL, CDL и IDL.

Подробнее см. "[Описание сущностей, компонентов и интерфейсов \(EDL, CDL, IDL\)](#)".

Файлы описаний (\*.edl, \*.cdl и \*.idl) при сборке обрабатываются компилятором NK. В результате создаются файлы \*.edl.h, \*.cdl.h, и \*.idl.h, содержащие транспортные методы и типы, используемые как на клиенте, так и на сервере.

## Компилятор NK

На основе EDL-, CDL- и IDL-описаний [компилятор NK](#) (nk-gen-c) генерирует набор транспортных методов и типов. Транспортные методы и типы нужны для формирования, отправки, приема и обработки IPC-сообщений

Важнейшие транспортные методы:

- **Интерфейсные методы.** При вызове на клиенте интерфейсного метода серверу отправляется IPC-запрос для вызова соответствующего метода.
- **Dispatch-методы (диспетчеры).** При получении запроса сервер вызывает диспетчер, который в свою очередь вызывает реализацию соответствующего метода.

Важнейшие транспортные типы:

- **Типы, определяющие структуру фиксированной части сообщения.** Передаются в интерфейсные методы, диспетчеры и функции транспорта (`nk_transport_recv()`, `nk_transport_reply()`).
- **Типы прокси-объектов.** Прокси-объект используется как аргумент интерфейсного метода.

Подробнее см. "[Сгенерированные методы и типы](#)".



## Init-описание и утилита einit

Утилита `einit` позволяет автоматизировать создание кода инициализирующей сущности `Einit`. Эта сущность первой запускается при загрузке KasperskyOS и запускает остальные сущности, а также создает каналы (соединения) между ними.

Чтобы сущность `Einit` при запуске создала соединение между сущностями `A` и `B`, в файле `init.yaml` нужно указать:

```
init.yaml

# Запустить B
- name: B
# Запустить A
- name: A
  connections:
    # Создать соединение с серверной сущностью B.
    - target: B
    # Имя нового соединения: some_connection_to_B
    id: some_connection_to_B
```

Подробнее см. "[Запуск сущностей](#)".

## Локатор сервисов (Service Locator)

Локатор сервисов – библиотека, содержащая следующие функции:

- `ServiceLocatorConnect()` – позволяет узнать клиентский IPC-дескриптор канала с указанным именем.
- `ServiceLocatorRegister()` – позволяет узнать серверный IPC-дескриптор канала с указанным именем.
- `ServiceLocatorGetRiid()` – позволяет узнать RIID (порядковый номер реализации интерфейса) по имени реализации интерфейса.

Значения IPC-дескриптора и RIID используются при инициализации транспорта `NkKosTransport`.

Чтобы использовать локатор сервисов, нужно в коде сущности подключить файл `sl_api.h`:

```
#include <coresrv/sl/sl_api.h>
```

Подробнее см. "[Локатор сервисов \(Service Locator\)](#)".

## Пример echo

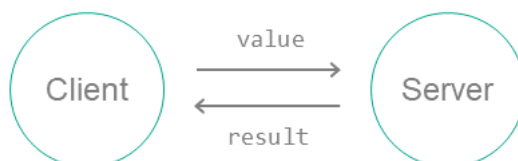
Пример `echo` демонстрирует использование IPC-транспорта.

Показана работа с основными инструментами, позволяющими реализовать взаимодействие между сущностями.

## О примере echo

Пример echo описывает простейший случай взаимодействия двух сущностей:

1. Сущность `Client` передает сущности `Server` число (`value`).
2. Сущность `Server` изменяет это число и передает новое число (`result`) сущности `Client`.
3. Сущность `Client` выводит число `result` на экран.

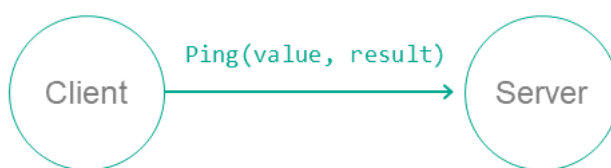


Чтобы организовать такое взаимодействие сущностей, потребуется:

1. Соединить сущности `Client` и `Server`, используя `init`-описание.
2. Реализовать на сервере интерфейс с единственным методом `Ping`, который имеет один входной аргумент – исходное число (`value`) и один выходной аргумент – измененное число (`result`).

Описание метода `Ping` на языке IDL:

```
Ping(in UInt32 value, out UInt32 result);
```



3. Создать файлы статических описаний на языках EDL, CDL и IDL. С помощью компилятора NK сгенерировать файлы, содержащие транспортные методы и типы (прокси-объект, диспетчеры и т.д.).
4. В коде сущности `Client` инициализировать все необходимые объекты (транспорт, прокси-объект, структуру запроса и др.) и вызвать интерфейсный метод.
5. В коде сущности `Server` подготовить все необходимые объекты (транспорт, диспетчер компонента и диспетчер сущности и др.), принять запрос от клиента, обработать его и отправить ответ.

Пример echo состоит из следующих исходных файлов:

- `client.c` – реализация сущности `Client`;
- `server.c` – реализация сущности `Server`;

- `Server.edl`, `Client.edl`, `Ping.cdl`, `Ping.idl` – статические описания;
- `init.yaml` – init-описание.

## Реализация сущности `Client` в примере `echo`

В коде сущности `Client` используются транспортные типы и методы, которые будут сгенерированы во время сборки решения компилятором NK на основе IDL-описания интерфейса `Ping`.

Однако чтобы получить необходимые для реализации сущности описания типов и сигнатуры методов, вы можете воспользоваться компилятором NK непосредственно после создания EDL-описания сущности, CDL-описаний компонентов и IDL-описаний используемых интерфейсов взаимодействия. В результате необходимые типы и сигнатуры методов будут объявлены в сгенерированных файлах `*.h`.

В реализации сущности `Client` необходимо:

1. Получить клиентский IPC-дескриптор соединения (канала), используя функцию локатора сервисов `ServiceLocatorConnect()`.

На вход нужно передать имя IPC-соединения `server_connection`, заданное ранее в файле `init.yaml`.

2. Инициализировать транспорт `NkKosTransport`, передав полученный IPC-дескриптор в функцию `NkKosTransport_Init()`.

3. Получить идентификатор необходимого интерфейса (RIID), используя функцию локатора сервисов `ServiceLocatorGetRiid()`.

На вход нужно передать полное имя реализации интерфейса `Ping`. Оно состоит из имен экземпляра компонента (`pingComp`) и интерфейса (`pingImpl`), заданных ранее в `Server.edl` и `Ping.cdl`.

4. Инициализировать прокси-объект, передав транспорт и идентификатор интерфейса в функцию `Ping_proxy_init()`.

5. Подготовить структуры запроса и ответа.

6. Вызвать интерфейсный метод `Ping_Ping()`, передав прокси-объект, а также указатели на запрос и ответ.

`client.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

/* Файлы, необходимые для инициализации транспорта. */
#include <coresrv/nk/transport-kos.h>
#include <coresrv/sl/sl_api.h>

/* Описание интерфейса сервера, по которому обращается клиентская сущность. */
#include <Ping.idl.h>

#include <assert.h>

#define EXAMPLE_VALUE_TO_SEND 777

/* Точка входа в клиентскую сущность. */
int main(int argc, const char *argv[])
```

```

{
    NkKosTransport transport;
    struct Ping_proxy proxy;
    int i;

    fprintf(stderr, "Hello I'm client\n");

    /* Получаем клиентский IPC-дескриптор соединения с именем
     * "server_connection". */
    Handle handle = ServiceLocatorConnect("server_connection");
    assert(handle != INVALID_HANDLE);

    /* Инициализируем IPC-транспорт для взаимодействия с серверной сущностью. */
    NkKosTransport_Init(&transport, handle, NK_NULL, 0);

    /* Получаем идентификатор интерфейса pingComp.pingImpl. Здесь
     * pingComp - имя экземпляра компонента Ping,
     * pingImpl - имя реализации интерфейса Ping. */
    nk_iid_t riid = ServiceLocatorGetRiid(handle, "pingComp.pingImpl");
    assert(riid != INVALID_RIID);

    /* Инициализируем прокси-объект, указав транспорт (&transport)
     * и идентификатор интерфейса сервера (riid). Каждый метод
     * прокси-объекта будет реализован как отправка запроса серверу. */
    Ping_proxy_init(&proxy, &transport.base, riid);

    /* Структуры запроса и ответа */
    struct Ping_Ping_req req;
    struct Ping_Ping_res res;

    /* Тестовый цикл. */
    req.value = EXAMPLE_VALUE_TO_SEND;
    for (i = 0; i < 10; ++i)
    {
        /* Вызываем интерфейсный метод Ping_Ping, передав прокси-объект,
         * а также указатели на запрос и ответ. При этом серверу будет
         * отправлен запрос для вызова метода Ping интерфейса
         * pingComp.pingImpl с аргументом value. Поскольку у метода Ping
         * нет аргументов типа sequence, arena не используется - вместо нее
         * передается NULL.
         * Поток блокируется до момента получения ответа от сервера. */
        if (Ping_Ping(&proxy.base, &req, NULL, &res, NULL) == rcOk)
        {
            /* Выводим значение result, содержащееся в ответе
             * (result - выходной аргумент метода Ping). */
            fprintf(stderr, "result = %d\n", (int) res.result);
            /* Помещаем полученное значение result в аргумент value
             * для повторной отправки серверу в следующей итерации. */
            req.value = res.result;
        }
        else
            fprintf(stderr, "Failed to call Ping.Ping()\n");
    }

    return EXIT_SUCCESS;
}

```

## Реализация сущности Server в примере echo

В коде сущности `Server` используются транспортные типы и методы, которые будут сгенерированы во время сборки решения компилятором NK на основе EDL-описания сущности `Server`.

Однако чтобы получить необходимые для реализации сущности типы и сигнатуры методов, вы можете воспользоваться компилятором NK непосредственно после создания EDL-описания сущности, CDL-описаний компонентов и IDL-описаний используемых интерфейсов взаимодействия. В результате необходимые типы и сигнатуры методов будут объявлены в сгенерированных файлах `*.h`.

В реализации сущности `server` необходимо:

1. Реализовать метод `Ping()`.

Сигнатура реализации метода `Ping()` должна в точности совпадать с сигнатурой интерфейсного метода `Ping_Ping()`, который объявлен в файле `Ping.idl.h`.

2. Получить серверный IPC-дескриптор соединения (канала), используя функцию локатора сервисов `ServiceLocatorRegister()`.

На вход нужно передать имя IPC-соединения `server_connection`, заданное ранее в файле `init.yaml`.

3. Инициализировать транспорт `NkKosTransport`, передав полученный IPC-дескриптор в функцию `NkKosTransport_Init()`.

4. Подготовить структуры запроса и ответа.

5. Инициализировать `dispatch`-метод (диспетчер) компонента `Ping`, используя функцию `Ping_component_init()`.

6. Инициализировать `dispatch`-метод (диспетчер) сущности `Server`, используя функцию `Server_entity_init()`.

7. Получить запрос, вызвав `nk_transport_recv()`.

8. Обработать полученный запрос, вызвав диспетчер `Server_entity_dispatch()`.

Диспетчер вызовет необходимую реализацию метода на основе полученного от клиента идентификатора интерфейса (RIID).

9. Отправить ответ сущности `Client`, вызвав `nk_transport_reply()`.

`server.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* Файлы, необходимые для инициализации транспорта. */
#include <coresrv/nk/transport-kos.h>
#include <coresrv/sl/sl_api.h>

/* Описания сущности-сервера на языке EDL. */
#include <Server.edl.h>

#include <assert.h>
```

```

/* Тип объекта реализующего интерфейс. */
typedef struct IPingImpl {
    struct Ping base;          // базовый интерфейс объекта
    int step;                  // дополнительные параметры
} IPingImpl;

/* Реализация метода Ping. */
static nk_err_t Ping_impl(struct Ping *self,
                           const Ping_req *req,
                           const struct nk_arena* req_arena,
                           Ping_res* res,
                           struct nk_arena* res_arena)
{
    IPingImpl *impl = (IPingImpl *)self;
    /* Значение value, пришедшее в запросе от клиента, инкрементируем на
     * величину шага step и помещаем в аргумент result, который будет
     * отправлен клиенту в составе ответа от сервера. */
    res->Ping.result = req->Ping.value + impl->step;
    return NK_EOK;
}

/* Конструктор объекта IPing.
 * step - шаг, то есть число, на которое будет увеличиваться входящее значение. */
static struct Ping *CreateIPingImpl(int step)
{
    /* Таблица реализаций методов интерфейса IPing. */
    static const struct Ping_ops ops = {
        .Ping = Ping_impl
    };

    /* Объект, реализующий интерфейс. */
    static struct IPingImpl impl = {
        .base = {&ops}
    };

    impl.step = step;

    return &impl.base;
}

/* Точка входа в сервер. */
int main(void)
{
    NkKosTransport transport;
    ServiceId iid;

    /* Получаем серверный IPC-дескриптор соединения "server_connection". */
    Handle handle = ServiceLocatorRegister("server_connection", NULL, 0, &iid);
    assert(handle != INVALID_HANDLE);

    /* Инициализируем транспорт до клиента. */
    NkKosTransport_Init(&transport, handle, NK_NULL, 0);

    /* Подготавливаем структуры запроса к сущности server: фиксированную
     * часть и арену. Поскольку ни у одного из методов сущности server нет
     * аргументов типа sequence, используются только фиксированные части
     * запроса и ответа. Арены фактически не используются. Однако в серверные

```

```

    * методы транспорта (nk_transport_recv, nk_transport_reply) и
    * dispatch-метод server_entity_dispatch необходимо передать валидные
    * аргументы запроса и ответа. */
Server_entity_req req;
char req_buffer[Server_entity_req_arena_size];
struct nk_arena req_arena = NK_ARENA_INITIALIZER(req_buffer, req_buffer +
sizeof(req_buffer));

/* Подготавливаем структуры ответа: фиксированную часть и арену. */
Server_entity_res res;
char res_buffer[Server_entity_res_arena_size];
struct nk_arena res_arena = NK_ARENA_INITIALIZER(res_buffer, res_buffer +
sizeof(res_buffer));

/* Инициализируем диспетчер компонента ping. 3 - величина шага,
* то есть число, на которое будет увеличиваться входящее значение. */
Ping_component component;
Ping_component_init(&component, CreateIPingImpl(3));

/* Инициализируем диспетчер сущности server. */
Server_entity entity;
Server_entity_init(&entity, &component);

fprintf(stderr, "Hello I'm server\n");

/* Реализация цикла обработки запросов. */
do
{
    /* Сбрасываем буферы с запросом и ответом. */
    nk_req_reset(&req);
    nk_arena_reset(&req_arena);
    nk_arena_reset(&res_arena);

    /* Ожидаем поступление запроса к сущности-серверу. */
    if (nk_transport_recv(&transport.base, &req.base_, &req_arena) != NK_EOK) {
        fprintf(stderr, "nk_transport_recv error\n");
    } else {
        /* Обрабатываем полученный запрос, вызывая реализацию Ping_impl
        * запрошенного интерфейсного метода Ping. */
        Server_entity_dispatch(&entity, &req.base_, &req_arena, &res.base_,
&res_arena);
    }

    /* Отправка ответа. */
    if (nk_transport_reply(&transport.base, &res.base_, &res_arena) != NK_EOK) {
        fprintf(stderr, "nk_transport_reply error\n");
    }
}
while (true);

return EXIT_SUCCESS;
}

```

Файлы описаний в примере echo

## Описание сущности client

Сущность `Client` не реализует ни одного интерфейса, поэтому в ее EDL-описании достаточно объявить имя сущности:

### Client.edl

```
/* Описание сущности Client. */  
entity Client
```

## Описание сущности server

Описание сущности `Server` должно содержать информацию о том, что он реализует интерфейс `Ping`. С помощью [статических описаний](#) нужно "поместить" реализацию интерфейса `Ping` в новый компонент (например, `Ping`), а экземпляр этого компонента – в сущность `Server`.

Сущность `Server` содержит экземпляр компонента `Ping`:

### Server.edl

```
/* Описание сущности Server. */  
entity Server  
/* pingComp - именованный экземпляр компонента Ping. */  
components {  
    pingComp: Ping  
}
```

Компонент `Ping` содержит реализацию интерфейса `Ping`:

### Ping.cdl

```
/* Описание компонента Ping. */  
component Ping  
/* pingImpl - реализация интерфейса Ping. */  
interfaces {  
    pingImpl: Ping  
}
```

Пакет `Ping` содержит объявление интерфейса `Ping`:

### Ping.idl

```
/* Описание пакета Ping. */  
package Ping  
interface {  
    Ping(in UInt32 value, out UInt32 result);  
}
```

## Init-описание



Чтобы сущность `Client` могла вызвать метод сущности `Server`, между ними требуется создать соединение (IPC-канал).

Для этого в `init`-описании укажите, что необходимо запустить сущности `Client` и `Server`, а также соединить их:

`init.yaml`

```
entities:

- name: Client
  connections:
    - target: Server
      id: server_connection

- name: Server
```

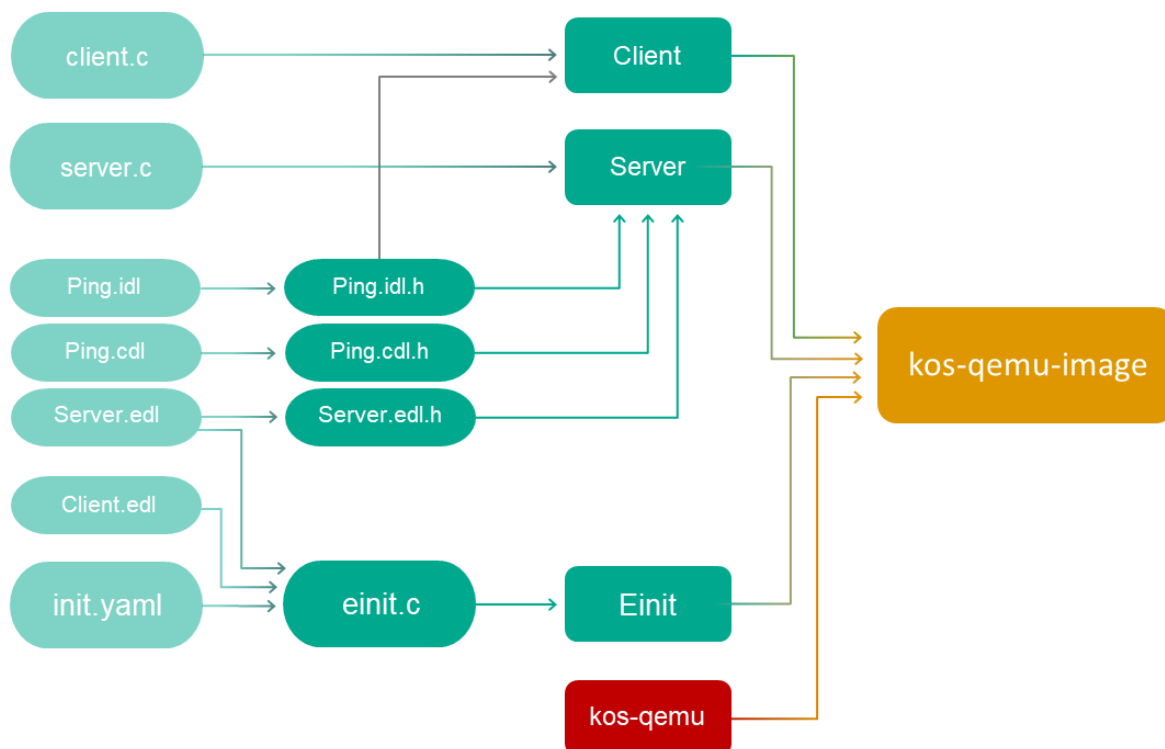
Поскольку сущность `Server` указана как `- target`, она будет выступать в роли серверной сущности, т.е. принимать запросы сущности `Client` и отвечать на них.

Созданный IPC-канал имеет имя `server_connection`. С помощью [локатора сервисов](#) можно получить дескриптор этого канала.

## Сборка и запуск примера echo

См. "[Сборка и запуск примеров](#)".

Схема сборки примера echo выглядит следующим образом:



## Часть 3. Политика безопасности решения

В предыдущих частях руководства показано, как реализовать взаимодействие между сущностями. При этом для простоты все решения собирались без модуля безопасности (`ksm.module`), которым представлена подсистема Kaspersky Security System.

Между тем, именно подсистема Kaspersky Security System контролирует обращения сущностей друг к другу и другие события. Это значит, что решение можно разделить на сущности, специфицировать правила их взаимодействия и, как следствие, повысить безопасность решения.

В этой части руководства на конкретных примерах рассматриваются:

- назначение и особенности Kaspersky Security System;
- синтаксис языка описания политики безопасности решения (PSL-описание);
- сборка модуля безопасности `ksm.module` на основе PSL-описания;
- использование различных классов политик для контроля взаимодействия сущностей;
- использование аудита для добавления событий в журнал;
- использование интерфейса безопасности.

## Политики безопасности

### Политики и классы политик

*Политика безопасности* или просто *политика* – это метод, определяющий правило проверки допустимости события. Политики делятся на два типа:

- *Политики-правила* – это методы, возвращающие решение "разрешено" или "запрещено". Политики-правила могут изменять внутреннее состояние модуля безопасности.
- *Политики-выражения* – это методы, возвращающие значение, которое может использоваться как аргумент других политик. Если вернуть значение не удалось, политики-выражения могут также вернуть "запрещено".

| Подробнее см. "[Политики-правила и политики-выражения](#)".

Политики безопасности выносят решение на основе данных о событии (например, имя запускаемой сущности или фактические аргументы вызываемого метода), а также могут учитывать состояние *объекта класса* (см. ниже), методами которого являются. Вызовы политик могут быть вложены друг в друга.

*Класс политик безопасности* – набор семантически связанных политик, описывающих определенную модель безопасности.

В составе KasperskyOS Community Edition поставляются следующие классы политик:

- `Base` – базовый класс, реализующий политики `grant`, `assert` и `deny`,

- `Regex` – класс, реализующий валидацию текстовых данных по статическим шаблонам (регулярным выражениям),
- `HashSet` – класс, реализующий механизмы работы со структурами данных типа "hash-таблица",
- `StaticMap` – класс, реализующий механизмы работы со структурами данных типа "словарь со статическими ключами",
- `Flow` – реализация конечного автомата,
- `RBAC` – реализация модели управления доступом на основе ролей (Role Based Access Control).

Подробнее см. "[Классы политик безопасности](#)".

## Объект класса

Каждая политика является методом заранее созданного *объекта класса*. Объект класса имеет внутреннее состояние, которое политики могут учитывать при вынесении решения "разрешено" или "запрещено".

В зависимости от описываемой модели безопасности, объект класса может:

- иметь глобальное (в контексте монитора безопасности) внутреннее состояние;
- позволять привязывать данные, содержащиеся в его внутреннем состоянии, к конкретным пользовательским ресурсам, взаимодействие с которыми нужно контролировать (например файлам или портам).

Вызов политики представляет собой вызов метода объекта класса в конкретных условиях и с конкретными аргументами. Некоторые *политики-правила* могут изменять состояние объекта класса, методами которого они являются.

## Связь с событиями и вызов политик

Когда сущность инициирует событие (отправка запроса или ответа, запуск другой сущности или вызов интерфейса безопасности), подсистема Kaspersky Security System вызывает все политики, *связанные* с этим конкретным событием. Если все политики вернули решение "разрешено", то Kaspersky Security System возвращает решение "разрешено". Если хотя бы одна политика вернула решение "запрещено", Kaspersky Security System возвращает решение "запрещено".

Если с событием не связана ни одна политика безопасности, Kaspersky Security System возвращает решение "запрещено". Таким образом, в KasperskyOS все, что не разрешено явно, запрещено (*принцип Default Deny*).

Связывание событий с политиками задается статически в специальном файле [security.psl](#) (т.н. *политика безопасности решения*).

## Политика безопасности решения (security.psl)

Файл `security.psl` называется *описанием политики безопасности решения* или просто *политикой безопасности решения*. В этом файле описаны правила взаимодействия сущностей, правила их запуска и обращения по интерфейсу безопасности, а также используемые профили аудита.

В приведенных примерах файл описания политики безопасности решения называется `security.psl`, хотя может иметь любое имя вида `*.psl`.

На основе `security.psl` собирается модуль безопасности (`ksm.module`). Для этого используется [специальный скрипт `makekss`](#), поставляемый в составе KasperskyOS Community Edition. Помимо файла `security.psl`, для сборки модуля требуются файлы [статических описаний](#) всех сущностей, компонентов и интерфейсов в решении. Собранный модуль имеет имя `ksm.module`.

Чтобы модуль безопасности был включен в образ решения, имя модуля нужно указать в параметрах [скрипта `makeimg`](#).

Если образ решения собран без модуля безопасности, то взаимодействия контролироваться не будут (будут разрешены любые взаимодействия).

## Структура файла `security.psl`

Описание политики безопасности решения состоит из следующих разделов:

- описание глобальных параметров модуля безопасности;
- подключение других PSL-описаний (в том числе используемых классов политик);
- подключение EDL-описаний сущностей;
- создание объектов классов политик;
- связывание событий с политиками;
- [опционально] объявление профилей аудита;
- [опционально] описание тестов политики безопасности решения на языке Policy Assertion Language (PAL).

## Общий синтаксис

Разделы описания политики безопасности решения могут располагаться в файле `security.psl` любом порядке.

Файл состоит из деклараций, которые могут быть:

- Линейными: `<decl> <body ...>`
- Блочными:

```
show <args ...> {  
    <body ...>  
}
```

Отступы являются частью синтаксиса и используются для разграничения деклараций. Формальных знаков для окончания декларации нет и она продолжается до тех пор, пока количество отступов в строке не станет равным количеству отступов в первой строке декларации.

Поддерживаются однострочные и многострочные комментарии:

```
/* Это комментарий  
   И это тоже */  
// Ещё один комментарий
```

## Описание глобальных параметров модуля безопасности

Глобальные параметры модуля безопасности являются общими для всех событий, описываемых в конфигурации безопасности решения. Глобальными параметрами монитора сообщений являются:

- Execute-интерфейс, который используется для запуска сущностей. Этот интерфейс объявляется декларацией `execute`:

```
execute: kl.core.Execute
```

Во всех примерах этого руководства для запуска сущностей используется интерфейс `Execute`, объявленный в пакете `kl.core` (`Execute.idl`). Этот интерфейс содержит единственный вариант запуска – метод `main` без параметров.

- [Опционально] Используемый по умолчанию профиль аудита и стартовое значение рантайм-уровня аудита (см. [Объявление и назначение профилей аудита](#)).

```
audit default = empty 0
```

## Подключение других PSL-описаний

PSL-описание можно разбивать на модули и включать одни модули в другие. Это позволяет, например, выносить общие части описаний в модули и переиспользовать их для написания похожих решений под разные архитектуры.

Содержимое подключаемого файла сливается с текущим файлом на этапе обработки компилятором `nk-psl-gen-c`.

Для подключения модуля используется декларация `use`:

```
use <путь.к.модюлю._>
```

Символ подчеркивания `"_"` используется для того, чтобы загрузить все определения из указанного файла (прямой аналог `#include` в C++).

## Подключение описаний классов политик безопасности

Чтобы использовать политики класса, необходимо подключить PSL-файл, содержащий определение этого класса.

Файлы классов находятся в директории `/opt/KasperskyOS-Community-Edition-<version>/toolchain/include/nk/`

Примеры:

```
/* Подключение базовых политик безопасности: grant и deny. */
use nk.base._

/* Подключение политик класса "Flow". */
use nk.flow._
```

## Подключение EDL-описаний сущностей

Описания всех включенных в решение сущностей должны быть подключены с помощью декларации `use EDL`:

```
/* Подключение сущностей "kl.core.Core" и "Einit". */
use EDL kl.core.Core
use EDL Einit

/* Подключение остальных сущностей, входящих в решение. */
use EDL Client
use EDL Server
```

## Создание объектов классов политик

Чтобы использовать политики класса, нужно сначала создать *объект* этого класса с помощью блочной декларации `policy object`:

```
policy object <имя объекта> : <Имя Класса> { ... }
```

При создании объекта класса необходимо задать содержащиеся в нем типы и указать его конфигурацию.

Например, для объекта класса `Flow` (реализация конечного автомата):

- тип `States` описывает все возможные состояния конечного автомата;
- конфигурация описывает начальное состояние автомата и переходы между состояниями.

```
/* Создание объекта класса "Flow". Имя нового объекта: request_state. */
policy object request_state : Flow {
    type States = "ping_next" | "pong_next"
    config = {
        states      : ["ping_next", "pong_next"],
```

```

    initial      : "ping_next",
    transitions : {
        "not_sent" : ["pong_next"],
        "sent"      : ["ping_next"]
    }
}
}

```

Каждый объект класса является реализацией модели безопасности со своим внутренним состоянием. Это состояние может быть глобальным или быть привязано к конкретному пользовательскому ресурсу. Некоторые политики-правила могут изменять это состояние, а также определять права доступа к ресурсу на основе текущего состояния.

Подробные описания конфигураций для различных классов содержатся в разделе "[Классы политик безопасности](#)".

## Связывание событий с политиками

Важнейшая часть политики безопасности решения – связывание событий с политиками безопасности (далее также "конфигурирование событий").

Для конфигурирования событий используются декларации следующего вида:

```

<вид события> <селекторы события> {<используемый профиль аудита> <список вызываемых политик> [вложенные декларации match]}

```

Здесь:

1. Вид события имеет значение `request`, `response`, `error`, `security` или `execute`.
2. Селекторы можно разделять запятыми. Допустимые селекторы зависят от вида события (см. ниже). Имена сущностей, компонентов, интерфейсов и методов в селекторах указываются так, как они описаны в EDL-, CDL- и IDL-файлах.

Обратите внимание: для Kaspersky Security System ядро представлено отдельной сущностью `kl.core.Core`.

3. Используемый профиль аудита указывается декларацией `audit <имя профиля>`.
4. Вызываемые политики указываются в формате:

```

<имя объекта класса>.<имя политики> <аргумент политики>

```

Политики без указания объекта считаются принадлежащими объекту `base`.

У каждой политики всегда имеется один аргумент, являющийся выражением на языке PSL. PSL-выражения имеют синтаксис, схожий с синтаксисом JSON.

Подробные описания выражений аргументов для каждой из политик различных классов содержатся в разделе "[Классы политик безопасности](#)".

Политики могут получать значения аргументов интерфейсных методов в формате `message.<name>`, где `<name>` – имя аргумента в соответствии с IDL-описанием метода.

Помимо аргументов вызываемого метода, в политику можно передавать значения `src_sid` и `dst_sid`, обозначающие дескрипторы сущностей. На какие сущности указывают значения `src_sid` и `dst_sid`, зависит от типа события (см. ниже).

Список вызываемых политик необязателен, если присутствует хотя бы одна вложенная декларация `match`.

5. Декларации `match` позволяют создавать "вложенные связывания", наследующие селекторы родительской декларации.

Одно и то же событие может быть связано с несколькими декларациями одного типа. При наступлении такого события будут вызваны политики из всех деклараций, подходящих под это событие. Политики вызываются последовательно, в том же порядке, в котором они встречаются в файле описания политики безопасности решения.

Событие, которое не связано ни одной декларацией, всегда запрещено.

## Событие запуска сущности (декларация `execute`)

Событие запуска сущности конфигурируется с помощью декларации `execute`. При этом можно указать селекторы `src` (имя сущности, инициирующей запуск) и `dst` (имя запускаемой сущности), например:

```
/* При запуске сущности "Server" будут вызваны две политики: request_state.allow и request_state.enter.
Первая политика возвращает решение "разрешено" только если объект "request_state" находится в состоянии "ready_to_start".
Вторая политика переводит объект request_state в состояние "not_ready" и возвращает решение "разрешено". */
execute dst=Server {
    request_state.allow {sid: dst_sid, states: ["ready_to_start"]}
    request_state.enter {sid: dst_sid, state: "not_ready"}
}
```

Если хотя бы одна из политик, связанных с событием, вернула решение "запретить", то состояния объектов изменены не будут.

В примере выше, если объект `request_state` находится в состоянии, отличном от `ready_to_start`, то запуск сущности `Server` будет запрещен, а состояние объекта `request_state` не изменится, несмотря на вызов политики `request_state.enter`.

Если селекторы `src` и `dst` не указаны, то декларация `execute` конфигурирует запуск всех сущностей всеми сущностями в решении:

```
/* Конфигурирование запуска всех сущностей (любая сущность может запускать любую другую сущность). */
execute {
    grant ()
}
```

В дальнейших примерах конфигурирования список политик для простоты содержит единственную политику `grant ()`.

## Событие отправки запроса (декларация `request`)

Событие отправки запроса конфигурируется с помощью декларации `request`. Например:



```
/* Конфигурирование всех запросов. */  
request {  
    grant ()  
}
```

Чтобы сконфигурировать запрос от конкретного клиента к конкретному серверу, используются селекторы `src` и `dst`:

```
/* Конфигурирование всех запросов от сущности Client. */  
request src=Client {  
    grant ()  
}  
/* Конфигурирование всех запросов к сущности Server. */  
request dst=Server {  
    grant ()  
}
```

Селекторы `src` и `dst` можно использовать совместно:

```
/* Конфигурирование запросов от сущности Client к сущности Server. */  
request src=Client, dst=Server {  
    grant ()  
}
```

Если необходимо задать вызываемый метод конкретной реализации интерфейса, используется селектор `method` совместно с селектором `endpoint`:

```
/* Конфигурирование запросов для вызова метода Ping реализации интерфейса "pingImpl"  
экземпляра компонента "pingComp". */  
request endpoint=pingComp.pingImpl, method=Ping {  
    grant ()  
}
```

Чтобы сконфигурировать обращения ко всем реализациям конкретного интерфейса, используйте селектор `interface` (только совместно с селектором `method`):

```
/* Конфигурирование запросов для вызова метода Ping любых реализаций интерфейса Ping.  
*/  
request interface=Ping, method=Ping {  
    grant ()  
}
```

## Событие отправки ответа (декларация `response`)

Событие отправки ответа конфигурируется аналогично событию отправки запроса.

Используется декларация `response`. Селектор `dst` обозначает сущность-клиент, селектор `src` – сущность-сервер. Например:

```
/* Конфигурирование ответов от сущности "Server" к сущности "Client" при вызове метода "Ping" реализации интерфейса "pingImpl" экземпляра компонента "pingComp". */
response src=Server, dst=Client, endpoint=pingComp.pingImpl, method=Ping {
    grant ()
}
```

## Событие отправки ошибки (декларация error)

Событие отправки ошибки возникает когда сущность-сервер выставляет флаг ошибки в сообщении-ответе перед его отправкой. Подробнее см. [Работа с ошибками в IDL](#).

Событие отправки ошибки конфигурируется аналогично событию отправки ответа.

Используется декларация `error`. Селектор `dst` обозначает сущность-клиент, селектор `src` – сущность-сервер. Например:

```
/* Конфигурирование отправки ошибок от сущности "Server" к сущности "Client" при вызове метода "Ping" реализации интерфейса "pingImpl" экземпляра компонента "pingComp". */
error src=Server, dst=Client, endpoint=pingComp.pingImpl, method=Ping {
    grant ()
}
```

## Событие обращения по интерфейсу безопасности (декларация security)

Обращение по интерфейсу безопасности конфигурируется с помощью декларации `security`. В селекторе `src` должно быть имя сущности, вызывающей метод, в селекторе `method` – имя метода интерфейса безопасности:

```
/* При вызове сущностью "Sdcard" метода "Register" интерфейса безопасности всегда возвращается решение "разрешено". */
security src=Sdcard, method=Register {
    grant ()
}
```

Если необходимо указать метод интерфейса безопасности, реализованный во вложенном компоненте, используется селектор `method` совместно с селектором `endpoint`:

```
/* При вызове сущностью "Sdcard" метода "Register" интерфейса безопасности "pingSec" экземпляра компонента "pingComp" всегда возвращается решение "разрешено". */
security src=Sdcard, endpoint=pingComp.pingSec, method=Register {
    grant ()
}
```

Вызов метода интерфейса безопасности отличается от других типов конфигурируемых событий тем, что к Kaspersky Security System обращается сущность, а не ядро. Поэтому именно сущность получает решение "разрешить" или "запретить".

## Вложенные связывания (декларация match)

Декларации `match` позволяют создавать "вложенные связывания", наследующие тип события и селекторы родительской декларации.

Вложенные `match`-декларации уточняют родительские декларации, объединяются с ними через конъюнкцию (AND) и не должны противоречить им.

Например:

```
/* Конфигурирование запросов от сущности "Client" к сущности "Server". */
request src=Client, dst=Server {
  /* Конфигурирование запросов от сущности "Client" к конкретным методам сущности "Server". */
  match endpoint=pingComp.pingImpl, method=Ping { grant () }
  match endpoint=pingComp.pingImpl, method=Pong { grant () }
}
```

## Объявление и назначение профилей аудита

Профили аудита определяют, какие политики и в каких случаях добавляют записи в журнал аудита.

Модуль безопасности передает записи журнала аудита через ядро KasperskyOS в сущность `klog`, которая декодирует и обрабатывает их. Подробнее см. пример `klog` в составе KasperskyOS Community Edition.

### Объявление профилей аудита

Профили аудита объявляются с помощью декларации `audit profile`:

```
audit profile <имя профиля> =
{ <рантайм-уровень>:
  { <имя объекта>:
    { kss: [ "granted", "denied" ]
      , <конфигурация объекта ...>
    }
  }
}
```

- Один профиль аудита может содержать несколько рантайм-уровней (в виде численного значения), предоставляющих разную детализацию записей в журнал. Kaspersky Security System предоставляет метод для программного переключения рантайм-уровня аудита без пересборки модуля безопасности (`ksm.module`).
- В рамках каждого рантайм-уровня можно назначать различные конфигурации для разных объектов классов политик.
- Поле `kss` в конфигурации объекта – это соглашение о том, какие сообщения попадут в журнал аудита в соответствии с принятым политикой решением. Список может быть пустым (никакие сообщения не будут

добавлены в журнал) или комбинацией литералов "granted" и "denied".

- Каждое значение конфигурации аудита объекта должно соответствовать своему типу в [определении класса этого объекта](#).

Пример объявления профиля аудита:

```
audit profile trace =
{ 0:
  { base:
    { kss: ["denied"]
    }
  },
  1:
  { session:
    { kss: ["granted", "denied"]
      , omit: ["closed"]
    }
  }
}
```

## Назначение профилей аудита

- При [связывании политик с событиями](#) каждому вызову политики можно назначить профиль аудита.
- Если профиль не указан в секции связывания, то используется профиль из родительской match-секции (при наличии).
- Если match-секции отсутствуют или у верхнеуровневой match-секции профиль не задан, то используется [глобальный](#).

Например:

```
audit default = global 0 // Задание профиля аудита по умолчанию
/* Конфигурирование запросов от сущности "Client" к сущности "Server". */
request src=Client, dst=Server {
  audit parent // На вызовы политик в этой секции назначен профиль аудита parent.
  /* Конфигурирование запросов от сущности "Client" к конкретным методам сущности
  "Server". */
  match endpoint=pingComp.pingImpl, method=Ping {
    audit child
    grant () // На этот вызов назначен профиль аудита child.
  }
  match endpoint=pingComp.pingImpl, method=Pong {
    grant () // Так как на этот вызов не назначен профиль аудита, будет
    использован родительский (parent).
  }
}
response src=Client, dst=Server {
  grant () // Так как на этот вызов не назначен профиль аудита, будет использован
  глобальный (global).
}
```

# Тестирование политики безопасности решения на языке Policy Assertion Language (PAL)

Базовый сценарий использования Policy Assertion Language (PAL) – это создание тестовых сценариев для проверки политик безопасности решения, написанных с использованием PSL.

Язык PAL позволяет формировать и отправлять запросы в модуль безопасности, проверяя полученные решения на соответствие ожидаемым решениям. При этом для запуска тестовых сценариев, написанных на PAL, нет необходимости генерировать код клиент-серверных взаимодействий, так как PAL работает на уровне абстракции модуля безопасности.

## Общий синтаксис

Тестовый сценарий является последовательностью запросов к модулю безопасности, для каждого из которых указано ожидаемое решение.

Тестовые сценарии можно объединять в группы. Группа сценариев может также содержать секции `setup` и `finally`, которые будут выполняться соответственно перед и после выполнения *каждого* сценария в этой группе. Это можно использовать для задания общих стартовых условий или для проверки общих инвариантов.

Для объявления группы тестовых сценариев используется блочная декларация `assert`:

```
assert "имя группы сценариев" {  
  setup {}  
  sequence "имя сценария 1" {}  
  sequence "имя сценария 2" {}  
  ...  
  finally {}  
}
```

## Синтаксис запросов

Для конфигурирования запросов внутри тестовых сценариев используются декларации следующего вида:

```
<ожидание> <заголовок> <операция> <селекторы события> {сообщение}
```

Здесь:

- `<ожидание>` – ожидаемое решение: `grant`, `deny` или `any`.

При ожидании `any` решение модуля безопасности игнорируется, но ошибка при выполнении запроса пометит тестовый сценарий как неудачный.

Если ожидаемое решение не указано явно, ожидается решение `grant`.

- `<заголовок>` – опциональный параметр, содержащий текстовый заголовок описываемого запроса.

- <операция> – один из видов обращения к модулю безопасности: `execute`, `security`, `request` или `response`.
- <селекторы события> – допустимые селекторы события совпадают с селекторами, использующимися при [связывании событий с политиками](#). Например можно указать сущность-получатель сообщения или вызываемый метод.
- {сообщение} – параметр, содержащий значение аргументов вызываемого метода. Значение этого параметра должно соответствовать операции и селекторам запроса. По умолчанию передается пустое количество аргументов {}.

Пример:

```
assert "some tests" {
  sequence "first sequence" {
    // При вызове сущностью Client метода Ping реализации интерфейса
    pingComp.pingImpl сущности Server ожидается решение grant
    // При этом в аргументе value метода Ping передается значение 100
    grant "Client calls Ping" request src=Client dst=Server
    endpoint=pingComp.pingImpl method=Ping {value: 100}
    // Ожидается, что сущности Server запрещено отвечать на запросы сущности
    Client
    deny "Server cannot respond" response src=Server dst=Client
  }
}
```

## Сокращенная форма запросов

Для удобства можно также применять следующие сокращенные формы запросов:

- При выполнении операции `execute`, можно сохранить дескриптор запускаемой сущности в переменную с помощью оператора `<-`.
- Сокращенная форма для операции `security: <сущность> ! <путь.к.методу> {сообщение}`  
При выполнении разворачивается в полную форму: `security src=<сущность> method=<путь.к.методу> {сообщение}`
- Сокращенная форма для операции `request: <клиент> ~> <сервер> : <путь.к.имплементации.метода> {сообщение}`  
При выполнении разворачивается в полную форму: `request src=<клиент> dst=<сервер> endpoint=<путь.к.имплементации> method=<метод> {сообщение}`
- Сокращенная форма для операции `response: <клиент> <~ <сервер> : <путь.к.имплементации.метода> {сообщение}`  
При выполнении разворачивается в полную форму: `response src=<сервер> dst=<клиент> endpoint=<путь.к.имплементации> method=<метод> {сообщение}`

Пример:

```
assert "ping test"{
  setup {
    // переменная s содержит указатель на запущенный экземпляр сущности Server
```

```

s <- execute dst=ping.Server
// переменная s содержит указатель на запущенный экземпляр сущности Client
c <- execute dst=ping.Client
}

sequence "ping ping is denied" {
  // При вызове сущностью Client метода Ping реализации интерфейса
  pingComp.pingImpl сущности Server ожидается решение grant
  // При этом в аргументе value метода Ping передается значение 100
  c ~> s : pingComp.pingImpl.Ping {value: 100 }
  // При повторном вызове ожидается решение deny
  deny c ~> s : pingComp.pingImpl.Ping {value: 100 }
}

sequence "normal" {
  // При последовательном вызове методов Ping и Pong ожидаются решения grant
  c ~> s : pingComp.pingImpl.Ping { value: 100 }
  c ~> s : pingComp.pingImpl.Pong { value: 100 }
}
}

```

## Запуск тестов

Чтобы запустить написанные на PAL тестовые сценарии, необходимо использовать флаг `--tests run` при запуске компилятора [nk-psl-gen-c](#):

```
$ nk-psl-gen-c --tests run <другие параметры> ./security.psl
```

Компилятор `nk-psl-gen-c` сгенерирует код модуля безопасности и код тестовых сценариев, а затем скомпилирует их с помощью `gcc` и запустит.

Чтобы сгенерировать код тестовых сценариев, но не компилировать и запускать их, необходимо использовать флаг `--tests generate`.

## Политики-правила и политики-выражения

В языке PSL используются политики безопасности двух типов: *политики-правила* и *политики-выражения*.

Наиболее часто в описании политики безопасности решения используются политики-правила: они возвращают *решение* "разрешено" или "запрещено". В отличие от них, политики-выражения возвращают некоторое значение – например, текущее состояние объекта класса, которое может использоваться в аргументах других политик, которые вызвали эту политику-выражение.

### Политики-правила

Политика-правило возвращает решение "разрешено" или "запрещено".

Например:

```

/* Запуск сущности "Server" разрешен только если объект "flow_instance" находится в
состоянии "ready_to_start". */
execute dst=Server {
    flow_instance.allow {sid: dst_sid, states: ["ready_to_start"]}
}

```

Некоторые политики-правила могут изменять состояние объекта класса, методами которого они являются. Например, политика `flow_instance.enter`; переводит объект `flow_instance` в состояние, указанное в аргументе политики.

Политика `enter` изменит состояние объекта класса, только если с событием не связаны другие политики, или все связанные политики вернули решение "разрешено". Это правило действует для любых политик-правил, изменяющих состояние объекта.

## Политики-выражения

Политика-выражение возвращает значение, которое может использоваться в аргументах других политик, которые вызывают эту политику-выражение.

Например, это значение может использоваться в декларации множественного выбора `choice`. Декларация `choice` позволяет связать событие с разными политиками-правилами в зависимости от значения, которое вернула политика-выражение.

Примером политики-выражения является политика `query ()`, возвращающая текущее состояние объекта класса `Flow`:

```

/* При отправке запроса сущностью "ResourceDriver" будет проверено состояние объекта
"service_flow".
Если "service_flow" находится в состоянии "started" или "stopped", то отправка запроса
разрешена,
иначе - запрещена. */
request src=ResourceDriver {
    choice (service_flow.query {sid: src_sid}) {
        "started"    : grant ()
        "stopped"    : grant ()
        -            : deny ()
    }
}

```

## Пример простейшей политики безопасности решения

Ниже приведена простейшая политика безопасности решения вида "все разрешено" для решения, состоящего из пользовательских сущностей `Client` и `Server`, а также сущности `Einit` и ядра `KasperskyOS`, представленного сущностью `k1.core.Core`.

Эта политика разрешает:

- все взаимодействия сущностей (отправку любых запросов и ответов);



- запуск всех сущностей;

Использование такой политики безопасности недопустимо в реальных решениях. Более сложная политика безопасности решения показана в примере [ping](#).

#### security.psl

```
execute: kl.core.Execute

use nk.base._

use EDL Einit
use EDL kl.core.Core
use EDL Client
use EDL Server

/* Запуск сущностей разрешен */
execute {
    grant ()
}
/* Отправка и получение запросов, ответов и ошибок разрешены.
   Это означает, что любая сущность может вызывать методы других сущностей и ядра
   (т.е. совершать любые системные вызовы). */

request {
    grant ()
}
response {
    grant ()
}

error {
    grant ()
}
/* Обращения по интерфейсу безопасности игнорируются. */
security {
    grant ()
}
```

## Пример ping

Пример ping демонстрирует использование политики безопасности решения для контроля взаимодействий между сущностями.

## О примере ping

Пример ping включает в себя две сущности: `Client` и `Server`.

Сущность `Server` предоставляет два идентичных метода `Ping` и `Pong`, которые получают число и возвращают измененное число:

```
Ping(in UInt32 value, out UInt32 result);
Pong(in UInt32 value, out UInt32 result);
```

Сущность `Client` вызывает оба этих метода в различной последовательности. Если вызов метода запрещен политикой безопасности решения, выводится сообщение `Failed to call...`

Транспортная часть примера `ping` практически аналогична таковой для примера [echo](#). Единственное отличие состоит в том, что в примере `ping` используется два метода (`Ping` и `Pong`), а не один. Поскольку использование IPC-транспорта подробно рассмотрено в комментариях к примеру `echo`, в примере `ping` оно рассматривается кратко.

В составе примера `ping` содержится политика безопасности решения (`security.psl`) на базе класса [flow](#).

## Состав примера `ping`

Пример `ping` состоит из следующих файлов:

- `client.c`
- `Client.edl`
- `server.c`
- `Server.edl`, `Ping.cd1`, `Ping.idl`
- `init.yaml`
- `security.psl`

## Реализация сущности `Client` в примере `ping`

Сущность `Client` для вызывает методы `Ping` и `Pong` в различной последовательности.

Инициализация транспорта до сервера, использование прокси-объекта и интерфейсных методов, а также назначение файла `Ping.idl.h` рассматриваются более подробно в комментариях к [файлу client.c](#) в примере `echo`.

`client.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
/* Файлы, необходимые для инициализации транспорта. */
#include <coresrv/nk/transport-kos.h>
#include <coresrv/sl/sl_api.h>

/* Описание интерфейса сервера, по которому обращается клиентская сущность. */
#include <Ping.idl.h>
```

```

#include <assert.h>

#define EXAMPLE_VALUE_TO_SEND 777

static struct Ping_proxy proxy;

static uint32_t ping(uint32_t value)
{
    /* Структуры запроса и ответа */
    struct Ping_Ping_req req;
    struct Ping_Ping_res res;

    req.value = value;
    /* Вызываем интерфейсный метод Ping_Ping.
     * Серверу будет отправлен запрос для вызова метода Ping интерфейса
     * pingComp.pingImpl с аргументом value. Вызывающий поток блокируется
     * до момента получения ответа от сервера. */
    if (Ping_Ping(&proxy.base, &req, NULL, &res, NULL) == rcOk)
    {
        fprintf(stderr, "Client: Ping(%d), result = %d\n", (int) value, (int)
res.result);
        value = res.result;
    }
    else
        fprintf(stderr, "Client: Ping(%d), failed\n", (int) value);

    return value;
}

static uint32_t pong(uint32_t value)
{
    /* Структуры запроса и ответа */
    struct Ping_Pong_req req;
    struct Ping_Pong_res res;

    req.value = value;
    /* Вызываем интерфейсный метод Ping_Pong.
     * Серверу будет отправлен запрос для вызова метода Pong интерфейса
     * ping_comp.ping_impl с аргументом value. Вызывающий поток блокируется
     * до момента получения ответа от сервера. */
    if (Ping_Pong(&proxy.base, &req, NULL, &res, NULL) == rcOk)
    {
        fprintf(stderr, "Client: Pong(%d), result = %d\n", (int) value, (int)
res.result);
        value = res.result;
    }
    else
        fprintf(stderr, "Client: Pong(%d), failed\n", (int) value);

    return value;
}

/* Точка входа в клиентскую сущность. */
int main(int argc, const char *argv[])
{
    NkKosTransport transport;
    uint32_t value;
    int i;

```

```

fprintf(stderr, "Hello I'm client\n");

/* Получаем клиентский IPC-дескриптор соединения с именем
 * "server_connection". */
Handle handle = ServiceLocatorConnect("server_connection");
assert(handle != INVALID_HANDLE);

/* Инициализируем IPC-транспорт для взаимодействия с серверной сущностью. */
NkKosTransport_Init(&transport, handle, NK_NULL, 0);

/* Получаем идентификатор интерфейса pingComp.pingImpl. */
nk_iid_t riid = ServiceLocatorGetRiid(handle, "pingComp.pingImpl");
assert(riid != INVALID_RIID);

/* Инициализируем прокси-объект, указав транспорт (&transport)
 * и идентификатор интерфейса сервера (riid). Каждый метод
 * прокси-объекта будет реализован как отправка запроса серверу. */
Ping_proxy_init(&proxy, &transport.base, riid);

/* Тестовый цикл. */
value = EXAMPLE_VALUE_TO_SEND;

for (i = 0; i < 5; ++i)
{
    value = ping(value);
    value = pong(value);
}

value = ping(value);
value = ping(value);

value = pong(value);
value = pong(value);

return EXIT_SUCCESS;
}

```

## Реализация сущности Server в примере ping

Инициализация транспорта до клиента, подготовка структур запроса и ответа, а также назначение файла `Server.edl.h` рассматриваются более подробно в комментариях к [файлу server.c](#) в примере echo.

server.c

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* Файлы, необходимые для инициализации транспорта. */
#include <coresrv/nk/transport-kos.h>
#include <coresrv/sl/sl_api.h>

/* Описания сущности-сервера на языках EDL, CDL, IDL. */

```

```

#include <Server.edl.h>

#include <assert.h>

#define INCREMENT_STEP 3

/* Тип объекта реализующего интерфейс. */
typedef struct PingImpl {
    struct Ping base;      // базовый интерфейс объекта
    int step;              // дополнительные параметры
} PingImpl;

/* Реализация метода Ping. */
static nk_err_t Ping_impl(struct Ping *self, const struct Ping_Ping_req *req, const
struct nk_arena* req_arena, struct Ping_Ping_res* res, struct nk_arena* res_arena)
{
    PingImpl *impl = (PingImpl *)self;
    /* Значение value, пришедшее в запросе от клиента, инкрементируем на
    * величину шага step и помещаем в аргумент result, который будет
    * отправлен клиенту в составе ответа от сервера. */
    res->result = req->value + impl->step;
    return NK_EOK;
}

/* Реализация метода Pong. */
static nk_err_t Pong_impl(struct Ping *self, const struct Ping_Pong_req *req, const
struct nk_arena* req_arena, struct Ping_Pong_res* res, struct nk_arena* res_arena)
{
    PingImpl *impl = (PingImpl *)self;
    /* Значение value, пришедшее в запросе от клиента, инкрементируем на
    * величину шага step и помещаем в аргумент result, который будет
    * отправлен клиенту в составе ответа от сервера. */
    res->result = req->value + impl->step;
    return NK_EOK;
}

/* Конструктор объекта Ping.
 * step - шаг, то есть число, на которое будет увеличиваться входящее значение. */
static struct Ping *CreatePingImpl(int step)
{
    /* Таблица реализаций методов интерфейса Ping. */
    static const struct Ping_ops ops = {
        .Ping = Ping_impl,
        .Pong = Pong_impl
    };

    /* Объект, реализующий интерфейс. */
    static struct PingImpl impl = {
        .base = {&ops}
    };

    impl.step = step;

    return &impl.base;
}

/* Точка входа в сервер. */
int main(void)
{
    NkKosTransport transport;

```

```

ServiceId iid;

/* Регистрируем соединение и получаем дескриптор для него. */
Handle handle = ServiceLocatorRegister("server_connection", NULL, 0, &iid);
assert(handle != INVALID_HANDLE);

/* Инициализируем транспорт до клиента. */
NkKosTransport_Init(&transport, handle, NK_NULL, 0);

/* Подготавливаем структуры запроса: фиксированную часть и арену. */
union Server_entity_req req;
char req_buffer[Server_entity_req_arena_size];
struct nk_arena req_arena = NK_ARENA_INITIALIZER(req_buffer, req_buffer +
sizeof(req_buffer));

/* Подготавливаем структуры ответа: фиксированную часть и арену. */
union Server_entity_res res;
char res_buffer[Server_entity_res_arena_size];
struct nk_arena res_arena = NK_ARENA_INITIALIZER(res_buffer, res_buffer +
sizeof(res_buffer));

/* Инициализируем диспетчер компонента ping. INCREMENT_STEP - значение "шага",
 * которое будет прибавляться к входному аргументу value. */
struct Ping_component component;
Ping_component_init(&component, CreatePingImpl(INCREMENT_STEP));

/* Инициализируем диспетчер сущности Server. */
struct Server_entity entity;
Server_entity_init(&entity, &component);

fprintf(stderr, "Hello I'm server\n");

/* Реализация цикла обработки запросов. */
do
{
    /* Сбрасываем буферы с запросом и ответом. */
    nk_req_reset(&req);
    nk_arena_reset(&req_arena);
    nk_arena_reset(&res_arena);

    /* Ожидаем запрос от клиента. */
    if (nk_transport_rcv(&transport.base, &req.base_, &req_arena) != NK_EOK)
        fprintf(stderr, "nk_transport_rcv error\n");
    else
        /* Обрабатываем полученный запрос, вызывая реализацию Ping_impl
         * запрошенного интерфейсного метода Ping. */
        Server_entity_dispatch(&entity, &req.base_, &req_arena, &res.base_,
&res_arena);

    /* Отправляем ответ. */
    if (nk_transport_reply(&transport.base, &res.base_, &res_arena) != NK_EOK)
        fprintf(stderr, "nk_transport_reply error\n");
}
while (true);

return EXIT_SUCCESS;
}

```

## Файлы описаний в примере ping

### Описание сущности Client

Сущность `Client` не реализует ни одного интерфейса, поэтому в EDL-описании достаточно объявить имя сущности.

#### Client.edl

```
/* Описание сущности Client. */
entity Client
```

### Описание сущности Server

Сущность `Server` реализует интерфейс `Ping`, содержащий два метода – `Ping` и `Pong`. Как и в примере [echo](#), требуется объявить отдельный компонент (например `Ping`), содержащий реализацию интерфейса `Ping`.

В EDL-описании сущности `server` необходимо указать, что она содержит экземпляр компонента `Ping`:

#### Server.edl

```
/* Описание сущности Server. */
entity Server
/* pingComp - именованный экземпляр компонента Ping. */
components {
    pingComp: Ping
}
```

В CDL-описании компонента `Ping` необходимо указать, что он содержит реализацию интерфейса `Ping`:

#### Ping.cdl

```
/* Описание компонента Ping. */
component Ping

/* pingImpl - реализация интерфейса Ping. */
interfaces {
    pingImpl: Ping
}
```

В пакете `Ping` необходимо объявить интерфейс `Ping`, содержащий два метода – `Ping` и `Pong`:

#### Ping.idl

```
/* Описание пакета Ping. */
package Ping
interface {
    Ping(in UInt32 value, out UInt32 result);
    Pong(in UInt32 value, out UInt32 result);
}
```

## Init-описание

Чтобы сущность `Client` могла вызвать метод сущности `Server`, между ними требуется создать [IPC-канал](#).

init.yaml

```
entities:
- name: Client
  connections:
    - target: Server
      id: server_connection
- name: Server
```

Канал имеет имя `server_connection`. С помощью [локатора сервисов](#) можно получить дескриптор этого канала.

## Политика безопасности решения в примере ping

Политика безопасности решения в этом примере разрешает запуск всех сущностей и позволяет любой сущности обращаться к сущностям `Core` и `Server`. При этом вызовы методов сущности `Server` контролируются с помощью политик безопасности класса `flow` (модель конечного автомата).

Конечный автомат, описанный в конфигурации объекта `request_state`, имеет два состояния: `ping_next` и `pong_next`. Исходное состояние – `ping_next`. Разрешены только переходы из `ping_next` в `pong_next` и обратно.

При вызове методов `Ping` и `Pong` проверяется текущее состояние объекта `request_state`. В состоянии `ping_next` разрешен только вызов `Ping`, при этом состояние изменится на `pong_next`. Аналогично, в состоянии `pong_next` разрешен только вызов `Pong`, при этом состояние изменится на `ping_next`.

Таким образом, методы `Ping` и `Pong` разрешено вызывать только по очереди.

security.psl

```
/* Политика безопасности решения для демонстрации использования класса "flow" в
примере "ping". */

/* Импорт файла с объявлением псевдонимов базовых политик
и файла с объявлением класса политик "flow" (finite-state machine). */
use nk.base._
use nk.flow._

/* Создание объекта класса "flow". Имя нового объекта: request_state. */
policy object request_state : Flow {
  type States = "ping_next" | "pong_next"
  config = {
    states      : ["ping_next" , "pong_next"],
    initial     : "ping_next",
    transitions : {
```



```

        "ping_next" : ["pong_next"],
        "pong_next" : ["ping_next"]
    }
}

/* Запуск сущностей разрешен. */
execute {
    grant ()
}

/* Сообщения типа request разрешены. */
request {
    grant ()
}

/* Сообщения типа response разрешены. */
response {
    grant ()
}

/* Объявление сущностей. */
use EDL kl.core.Core
use EDL Client
use EDL Server
use EDL Einit

/* При запуске сущности "Server" перевести request_state в начальное состояние. */
execute dst=Server {
    request_state.init {sid: dst_sid}
}

/* При вызове метода Ping проверить, что объект "request_state" находится в состоянии
"ping_next".
Если это так, разрешить вызов метода Ping и перевести объект "request_state" в
состояние "pong_next". */
request dst=Server, endpoint=pingComp.pingImpl, method=Ping {
    request_state.allow {sid: dst_sid, states: ["ping_next"]}
    request_state.enter {sid: dst_sid, state: "pong_next"}
}

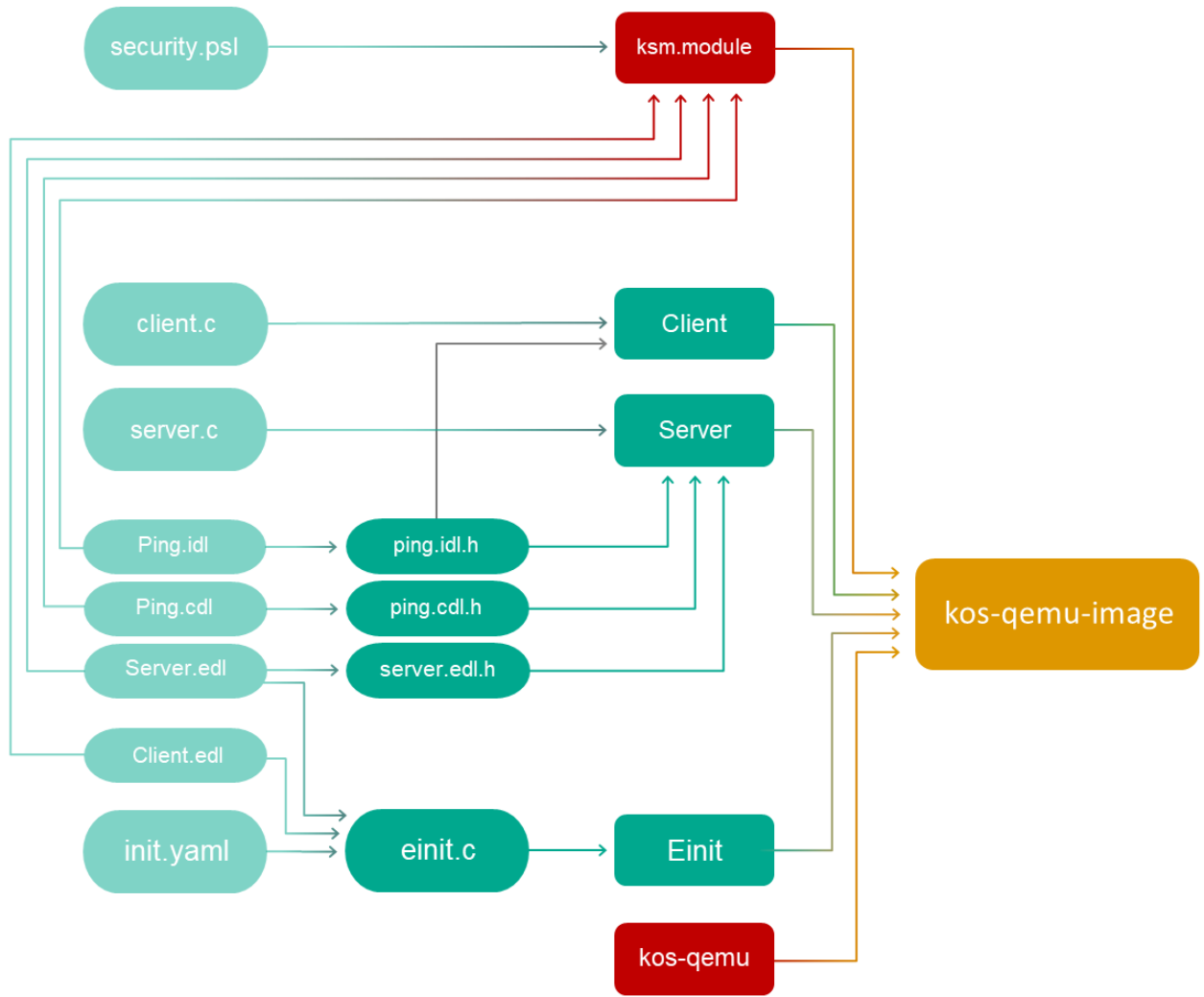
/* При вызове метода Pong проверить, что объект "request_state" находится в состоянии
"pong_next".
Если это так, разрешить вызов метода Pong и перевести объект "request_state" в
состояние "ping_next". */
request dst=Server, endpoint=pingComp.pingImpl, method=Pong {
    request_state.allow {sid: dst_sid, states: ["pong_next"]}
    request_state.enter {sid: dst_sid, state: "ping_next"}
}

```

## Сборка и запуск примера ping

См. ["Сборка и запуск примеров"](#).

Схема сборки примера ping выглядит следующим образом:



## Описание сущностей, компонентов и интерфейсов (EDL, CDL, IDL)

Все используемые в решении сущности, компоненты и интерфейсы должны быть статически описаны с помощью языков EDL, CDL и IDL. Соответствующие описания хранятся в файлах \*.edl, \*.cdl и \*.idl.

Файлы описаний используются при сборке решения для следующих целей:

- [генерация заголовочных файлов](#), содержащих транспортные методы и типы;
- [сборка модуля безопасности](#).

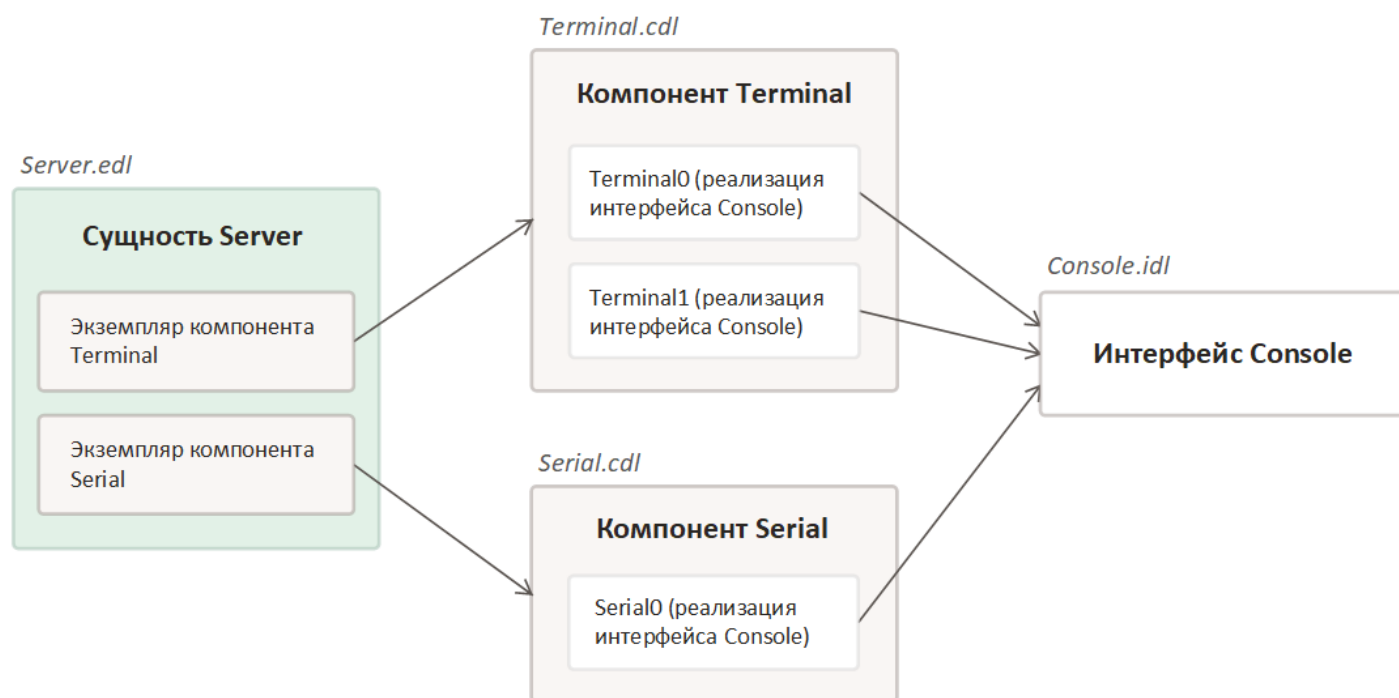
## Модель "сущность-компонент-интерфейс"

Каждая сущность может предоставлять один или более интерфейсов взаимодействия. Несколько интерфейсов могут быть объединены в компонент. Компоненты могут быть вложены в другие компоненты. Все используемые в решении сущности, компоненты и интерфейсы должны быть *статически описаны*.

В KasperskyOS есть три вида файлов статических описаний:

- **EDL-файл** содержит описание сущности на языке Entity Definition Language (далее [EDL](#)): ее имя, состав компонентов и интерфейсов, а также другую информацию.
- **CDL-файл** содержит описание компонента на языке Component Definition Language (далее [CDL](#)). Описание компонента включает в себя его имя, а также набор включенных в него компонентов и реализуемых интерфейсов.
- **IDL-файл** содержит описание пакета, содержащего один интерфейс на языке Interface Definition Language (далее [IDL](#)). Описание включает в себя имя пакета, объявление входящего в него интерфейса, а также объявление типов и именованных констант.

Для обеспечения гибкости сущность может содержать несколько экземпляров компонента. Несколько сущностей могут включать экземпляры одного и того же компонента. Сущность может не содержать ни одного компонента и не реализовывать ни один интерфейс. В этом случае она не предоставляет функциональности, доступной другим сущностям.



Сущность Server включает в себя экземпляры компонентов Terminal и Serial, содержащих различные реализации интерфейса Console

## EDL

Каждая сущность в KasperskyOS должна быть описана на языке Entity Definition Language в отдельном файле `<имя сущности>.edl`.

Имя EDL-файла должно совпадать с именем сущности, которую он описывает.

EDL-файл содержит следующие разделы (порядок важен):

- Имя сущности.** Обязательный раздел, начинающаяся с ключевого слова `entity`, за которым следует имя сущности.  
Имя сущности должно начинаться с заглавной буквы и не может содержать символ подчеркивания (`_`).
- Интерфейс безопасности, используемый сущностью.** Раздел не является обязательным и должен быть добавлен, только если сущность использует интерфейс безопасности. Декларируется ключевым словом `security`, за которым следует полное имя интерфейса.
- Перечень реализаций интерфейсов, предоставляемых сущностью.** Не является обязательным и описывается в секции `interfaces`. Каждая реализация интерфейса указывается отдельной строкой в следующем формате:

```

interfaces {
    <имя реализации интерфейса>:<имя интерфейса>
}
  
```

Сущность может содержать несколько реализаций одного интерфейса. Все реализуемые интерфейсы необходимо описать на языке [IDL](#) в IDL-файлах.

Имя реализации интерфейса не может содержать символ подчеркивания (`_`).

4. **Список экземпляров компонентов, входящих в сущность.** Не является обязательным и описывается в секции `components`. Каждый экземпляр компонента указывается отдельной строкой в следующем формате:

```
components {
    <имя экземпляра компонента>:<имя компонента>
}
```

Для каждого указанного компонента необходимо создать отдельный файл `<имя компонента>.cdl`, содержащий описание компонента на языке CDL. В сущность можно добавить несколько экземпляров одного и того же компонента, причем каждый может иметь отдельное состояние (см. ниже пример сущности `UartDriver`).

Имя экземпляра компонента не может содержать символ подчеркивания (`_`).

Секции `interfaces` и `components` не являются обязательными и добавляются только для серверных сущностей. Реализации интерфейсов, объявленные в секции `interfaces`, а также реализации, входящие в компоненты из секции `components` могут использоваться сущностями-клиентами.

EDL поддерживает однострочные и многострочные комментарии в стиле C++:

```
/* Это комментарий
   И это тоже */
// Ещё один комментарий
```

## Примеры EDL-файлов

В простейшем случае сущность не использует интерфейс безопасности и не предоставляет функциональности другим сущностям, подобно сущности `hello` из примера `hello`. EDL-описание такой сущности содержит только ключевое слово `entity` и имя сущности.

`Hello.edl`

```
// Имя сущности: Hello
entity Hello
```

В следующем примере сущность `Efoo` содержит единственную реализацию интерфейса и не использует интерфейс безопасности.

`Efoo.edl`

```
// Имя сущности: Efoo
entity Efoo
// Сущность содержит именованную реализацию интерфейса IFooI. Имя реализации: foo.
interfaces {
    foo : IFoo
}
```

В следующем примере сущность `UartDriver` содержит два экземпляра компонента `UartComp` – по одному на каждое UART-устройство.

Сущность `UartDriver` не использует интерфейс безопасности.

#### UartDriver.edl

```
// Имя сущности: UartDriver
entity UartDriver
// uart0 и uart1 - имена экземпляров компонента UartComp,
// которые отвечают за два различных устройства
components {
    uart0: UartComp
    uart1: UartComp
}
```

## CDL

Каждый используемый в решении компонент должен быть описан на языке CDL в отдельном файле `<имя компонента>.cdl`.

Имя CDL-файла должно совпадать с именем компонента, который он описывает.

CDL-файл содержит следующие разделы:

1. **Имя компонента.** Перед именем компонента ставится ключевое слово `component`.  
Имя компонента должно начинаться с заглавной буквы и не может содержать символ подчеркивания (`_`).
2. **Интерфейс безопасности, содержащийся в этом компоненте.** Раздел не является обязательным и должен быть добавлен, только если в компоненте содержится интерфейс безопасности. Декларируется ключевым словом `security`, за которым следует полное имя интерфейса.
3. **Перечень реализаций интерфейсов, которые включены в этот компонент.** Интерфейсы декларируются в секции `interfaces`, внутри которой каждая реализация интерфейса указывается отдельной строкой в следующем формате:

```
interfaces {
    <имя реализации интерфейса>:<имя интерфейса>
}
```

Компонент может содержать несколько реализаций одного интерфейса. Все реализуемые интерфейсы необходимо описать на языке [IDL](#) в IDL-файлах.

Имя реализации интерфейса не может содержать символ подчеркивания (`_`).

4. **Список экземпляров компонентов, вложенных в этот компонент.** Раздел не является обязательным и добавляется в том случае, если компонент содержит вложенные компоненты. Компоненты декларируются в секции `components`, внутри которой каждый экземпляр компонента указывается отдельной строкой в следующем формате:

```
components {
    <имя экземпляра компонента>:<имя компонента>
}
```

Для каждого указанного компонента необходимо создать отдельный файл `<имя компонента>.cdl`, содержащий описание компонента на языке CDL. В компонент можно добавить несколько экземпляров одного и того же компонента, причем каждый может иметь отдельное состояние.

Имя экземпляра компонента не может содержать символ подчеркивания (`_`).

CDL поддерживает однострочные и многострочные комментарии в стиле C++.

## Примеры CDL-файлов

В простейшем случае компонент содержит единственную реализацию интерфейса, подобно компоненту `ping` из примера [echo](#).

### Ping.cdl

```
/* Имя компонента: Ping */
component Ping
/* Компонент содержит именованную реализацию интерфейса IPing. Имя реализации:
pingimpl.*/
components {
    pingimpl: IPing
}
```

В следующем примере компонент `CoFoo` содержит реализации двух интерфейсов, объявленных в двух разных пакетах `Foo` и `Baz` (т.е. в файлах `Foo.idl` и `Bar.idl`):

### CoFoo.cdl

```
/* Имя компонента: CoFoo */
component CoFoo
interfaces {
    /* Компонент содержит реализацию интерфейса Foo. Имя реализации: foo.*/
    foo: Foo
    /* Компонент содержит три разных реализации интерфейса Bar. Имена реализаций:
    bar1, bar2 и bar3.*/
    bar1: Bar
    bar2: Bar
    bar3: Bar
}
```

В следующем примере компонент `CoFoo` содержит единственную реализацию интерфейса, а также вложенный компонент.

### CoFoo.cdl

```
/* Имя компонента: CoFoo */
component CoFoo
interfaces {
    /* Компонент содержит реализацию интерфейса Foo. Имя реализации: foo.*/
    foo: Foo
}
components {
    /* Компонент содержит экземпляр компонента CoBar. Имя экземпляра: bar.*/
    bar: CoBar
}
```

# IDL

Все реализуемые в решении интерфейсы необходимо описать в IDL-файлах.

В каждом IDL-файле может быть объявлен только один интерфейс.

Имя IDL-файла должно совпадать с именем интерфейса, который он описывает.

IDL-файл содержит следующие разделы:

## 1. Имя интерфейса.

Единственный обязательный раздел.

Имя интерфейса должно начинаться с заглавной буквы и не может содержать символ подчеркивания (\_).

Объявление имени имеет следующий формат:

```
package <имя описываемого интерфейса>
```

Имя интерфейса может быть составным. В этом случае оно используется для формирования пути к IDL-файлу, например:

```
/* Module.idl расположен по следующему пути: a/b/c/Module.idl */  
package a.b.c.Module
```

## 2. Импорт внешних пакетов.

Инструкция импорта пакета имеет следующий формат:

```
import <имя внешнего пакета>
```

Позволяет ввести в область видимости IDL-файла элементы внешнего пакета: именованные константы и пользовательские типы, включая структуры и вариантыные типы.

## 3. Объявление структур, вариантных типов, именованных констант и синонимов.

## 4. Объявление методов интерфейса.

Объявление методов интерфейса имеет следующий синтаксис:

```
interface {  
    <Объявления методов>  
}
```

Внутри фигурных скобок находятся объявления методов, имеющие следующий синтаксис:

```
<Имя метода> (<аргументы>);
```



Имя метода не может содержать символ подчеркивания (\_). Рекомендуется начинать имена методов с заглавной буквы.

Аргументы разделяются на входные (in) и выходные (out) и перечисляются через запятую. Пример объявления интерфейса:

```
interface {  
    // Объявление метода Ping  
    Ping(in UInt32 value, out UInt32 result);  
}
```

Для возврата кодов ошибок, возникающих в результате обработки запроса сущностью-сервером, не рекомендуется использовать out-аргументы. Вместо них рекомендуется использовать специальные error-аргументы. Подробнее см. [Работа с ошибками в IDL](#).

IDL поддерживает однострочные и многострочные комментарии в стиле C++.

## Примеры IDL-файлов

В простейшем случае IDL-описание содержит только имя и объявление интерфейса:

### Ping.idl

```
/* Ping – имя интерфейса */  
package Ping  
/* Объявление методов интерфейса */  
interface {  
    // Объявление метода Ping  
    Ping(in UInt32 value, out UInt32 result);  
}
```

Пример более сложного IDL-описания:

### Foo.idl

```
// Объявление имени  
package Foo  
// Инструкции импорта  
import Bar1  
import Bar2  
// Объявление именованной константы  
const UInt32 MaxStringSize = 1024;  
// Объявление синонима для пользовательского типа  
typedef sequence <Char, MaxStringSize> String;  
// Объявление структуры  
struct BazInfo {  
    Char x;  
    array <String, 100> y;  
    sequence <sequence <UInt32, 100>, 200> y;  
}  
// Объявление методов интерфейса  
interface {  
    Baz(in BazInfo a, out UInt32 b);  
}
```

# Типы данных IDL

## Примитивные типы данных IDL

В IDL поддерживаются следующие примитивные типы данных:

- `SInt8`, `SInt16`, `SInt32`, `SInt64` (знаковое целое);
- `UInt8`, `UInt16`, `UInt32`, `UInt64` (беззнаковое целое);
- `Handle` (дескриптор).

Примеры объявления переменных примитивных типов:

```
SInt32 value;  
UInt8 ch;  
Handle ResID;
```

Для примитивных типов, кроме типа `Handle`, можно объявить именованную константу с помощью ключевого слова `const`:

```
const UInt32 c0 = 0;  
const UInt32 c1 = 1;
```

Далее мы рассмотрим сложные (составные) типы данных: `union` (вариантный тип), `struct` (структура), `array` (массив) и `sequence` (последовательность).

## Тип union

Вариантный тип `union`, подобно обычному объединению в языке C, позволяет хранить значения разных типов в одной области памяти. Однако при передаче IPC-сообщения тип `union` снабжается дополнительным полем `tag`, позволяющим определить, какое именно поле объединения используется в переданном значении. Поэтому тип `union` также называют *объединением с тегом*. Пример объявления переменной типа `union`:

```
union foo {  
    UInt32 bar;  
    UInt8 baz;  
}
```

Объявление вариантного типа является объявлением верхнего уровня.

## Тип struct

Структуры в языке IDL аналогичны структурам в языке C. Пример объявления структуры:

```
struct BazInfo {
    UInt64 x;
    UInt64 y;
}
```

Объявление структуры является объявлением верхнего уровня.

## Тип array

Массив (array) имеет фиксированный размер, задаваемый в момент объявления:

```
array <ТипЭлемента, ЧислоЭлементов>
```

В IDL массивы анонимны и могут объявляться непосредственно при объявлении других сложных типов, а также при объявлении интерфейсов.

Для назначения имени объявляемому типу массива используется ключевое слово `typedef`, например:

```
typedef array <SInt8, 1024> String;
```

## Тип sequence

Последовательность (sequence) напоминает массив переменного размера. При объявлении последовательности указывается максимальное число элементов, однако фактически можно передать меньшее их число. При этом для передачи будет задействовано столько памяти, сколько занимают передаваемые элементы.

Как и массивы, последовательности анонимны и могут объявляться непосредственно при объявлении интерфейсов, а также при объявлении других сложных типов:

```
sequence <ТипЭлемента, ЧислоЭлементов>
```

Пример объявления именованной последовательности:

```
typedef sequence <SInt8, 1024> String;
```

## Вложенные сложные типы

Сложные типы можно использовать при объявлении других сложных типов. При этом массивы (array) и последовательности (sequence) могут объявляться непосредственно внутри другого составного типа:

```
struct BazInfo {
    UInt64 x;
    array <UInt8, 100> y;
```

```
sequence <sequence <UInt32, 100>, 200> z;  
}
```

Типы `union` или `struct` должны объявляться перед использованием:

```
union foo {  
    UInt32 value1;  
    UInt8 value2;  
}  
  
struct bar {  
    UInt32 a;  
    UInt8 b;  
}  
  
struct BazInfo {  
    foo x;  
    bar y;  
}
```

## Объявление синонима типа (typedef)

Ключевое слово `typedef` вводит имя, которое внутри своей области видимости является синонимом для указанного типа, например:

```
typedef array <UInt32, 20> value;
```

Конструкции `typedef struct` и `typedef union` недопустимы — ввести новое имя для структуры или вариантного типа можно только если они объявлены заранее:

```
union foo {  
    UInt32 value1;  
    UInt8 value2;  
}  
  
typedef foo bar;
```

Объявление `typedef` не вводит новый тип данных — оно вводит новые имена для уже существующих типов.

## Работа с ошибками в IDL

`Out`-аргументы в [IDL-описании](#) не рекомендуется использовать для возврата сущности-клиенту кодов логических ошибок, возникающих в результате обработки запроса сущностью-сервером. Вместо этого рекомендуется использовать `error`-аргументы.

Логическая ошибка – это ошибка, которая возникла при обработке запроса сущностью-сервером и должна быть возвращена сущности-клиенту. Например, если сущность-клиент пытается открыть несуществующий файл, сервер файловой системы возвращает логическую ошибку. Внутренние ошибки сущности-сервера, такие как нехватка памяти или выполнение недопустимой инструкции, не являются логическими ошибками.

Использование `error`-аргументов позволяет:

- не пересылать все `out`-аргументы методов в сообщении-ответе от сущности-сервера к сущности-клиенту в случае возврата ошибки. Вместо этого, при необходимости вернуть код ошибки, сообщение-ответ будет содержать только `error`-аргументы.
- связывать политики безопасности не только с событиями отправки ответов, но также с событиями возврата ошибок.

Язык IDL и компилятор NK предоставляют два способа использования `error`-аргументов:

- **Упрощенная обработка ошибок**

Этот способ используется компилятором NK по умолчанию.

При использовании этого способа методы интерфейсов могут иметь *не более одного* `error`-аргумента типа `UInt16`. Значения возвращаемого `error`-аргумента метода упаковываются в код возврата соответствующего клиентского интерфейсного метода.

- **Расширенная обработка ошибок**

Для использования этого способа необходимо использовать параметр `--extended-errors` компилятора NK.

При использовании этого способа методы интерфейса могут иметь несколько `error`-аргументов любого типа. Значения возвращаемых `error`-аргументов метода содержатся во внутренних полях структуры сообщения-ответа.

Типы и макросы для работы с ошибками содержатся в файле `/opt/KasperskyOS-Community-Edition-<version>/toolchain/include/nk/types.h`

## Упрощенная обработка ошибок

При использовании упрощенной обработки ошибок методы интерфейсов могут иметь не более одного `error`-аргумента типа `UInt16` с именем `status`.

В случае ошибки:

- Реализации методов на стороне серверной сущности должны использовать макрос `NK_ELOGIC_MAKE()` для упаковки кода ошибки и возвращать результат его работы.
- Соответствующие клиентские интерфейсные методы возвращают код ошибки (логической или транспортной), упакованный в значение типа `nk_err_t`. Для проверки на наличие в нем логических и транспортных ошибок используются макросы `NK_ELOGIC_CHECK()` и `NK_ETRANSPORT_CHECK()` соответственно.

В случае успешной обработки запроса:

- Реализации методов на стороне серверной сущности должны поместить результаты в соответствующие `out`-аргументам поля сообщения-ответа и вернуть `NK_EOK`.

- Соответствующие клиентские интерфейсные методы также возвращают `NK_EOK`.

Например, рассмотрим IDL-описание интерфейса, содержащего один метод:

#### Connection.idl

```
typedef UInt16 ErrorCode
// пример кода ошибки
const ErrorCode ESendFailed = 10;

interface {
    Send(in Handle handle,
        in Packet packet,
        out UInt32 sentLen,
        // метод Send имеет один error-аргумент status типа UInt16
        error ErrorCode status
    );
}
```

Реализация серверной сущности:

#### server.c

```
...
// реализация метода Send
nk_err_t Send_impl(struct Connection *self,
    const struct Connection_Send_req *req,
    const struct nk_arena* req_arena,
    struct Connection_Send_res* res,
    struct nk_arena* res_arena)
{
    if (...) {
        // в случае успешной обработки запроса помещаем результат в out-аргумент и
        // возвращаем NK_EOK
        res->sentLen = value;
        return NK_EOK;
    } else {
        // в случае ошибки возвращаем результат работы макроса NK_ELOGIC_MAKE
        return NK_ELOGIC_MAKE(Connection_ESendFailed);
    }
}
```

Реализация клиентской сущности:

#### client.c

```
// значения возвращаемого error-аргумента метода упаковываются в код возврата
// соответствующего клиентского интерфейсного метода
nk_err_t ret = Connection_Send(ctx,
    &req, &req_arena,
    &res, NK_NULL);

if (ret == NK_EOK) {
    // если ошибок нет, res содержит значения out-аргументов
    return res.sentLen;
} else {
    if (NK_ELOGIC_CHECK(ret)) {
        // обрабатываем логическую ошибку
    } else {
```

```

    // обрабатываем транспортную ошибку
}
}

```

## Расширенная обработка ошибок

При использовании расширенной обработки ошибок методы интерфейсов могут иметь более одного `error` аргумента любого типа.

В случае ошибки:

- Реализации методов на стороне серверной сущности должны:
  - использовать макрос `nk_err_reset()`, чтобы выставить в сообщении-ответе флаг ошибки;
  - поместить коды ошибок в соответствующие `error`-аргументам поля сообщения-ответа;
  - вернуть `NK_EOK`.
- Соответствующие клиентские интерфейсные методы возвращают код транспортной ошибки или `NK_EOK`, если транспортной ошибки нет. Для проверки наличия в сообщении-ответе логических ошибок используется макрос `nk_msg_check_err()`. При этом значения возвращаемых `error`-аргументов метода содержатся во внутренних полях структуры сообщения-ответа.

В случае успешной обработки запроса:

- Реализации методов на стороне серверной сущности должны поместить результаты в соответствующие `out`-аргументам поля сообщения-ответа и вернуть `NK_EOK`.
- Соответствующие клиентские интерфейсные методы также возвращают `NK_EOK`.

Например, рассмотрим IDL-описание интерфейса, содержащего один метод:

### Connection.idl

```

typedef UInt16 ErrorCode

interface {
    Send(in Handle handle,
        in Packet packet,
        out UInt32 sentLen,
        // метод Send имеет один error-аргумент status типа UInt16
        error ErrorCode status
    );
}

```

При этом `union`-тип, сгенерированный компилятором NK для фиксированной части сообщения-ответа для этого метода, содержит как `out`-аргументы, так и `error`-аргументы:

### Connection.idl.h

```

...
struct Connection_Send_res {
    union {

```

```

    struct {
        struct nk_message base_;
        nk_uint32_t sentLen;
    };
    struct {
        struct nk_message base_;
        nk_uint32_t sentLen;
    } res_;
    struct kl_Connection_Send_err {
        struct nk_message base_;
        nk_uint16_t status
    } err_;
};
...

```

Реализация серверной сущности:

server.c

```

...
// реализация метода Send
nk_err_t Send_impl(struct Connection *self,
                   const struct Connection_Send_req *req,
                   const struct nk_arena* req_arena,
                   struct Connection_Send_res* res,
                   struct nk_arena* res_arena)
{
    if (...) {
        // в случае успешной обработки запроса помещаем результат в out-аргумент и
        // возвращаем NK_EOK
        res->sentLen = value;
        return NK_EOK;
    } else {
        // в случае ошибки вызываем макрос nk_err_reset(), чтобы выставить флаг ошибки в
        // сообщении-ответе
        nk_err_reset(res);
        // заполняем поля err-аргументов в структуре res->err_
        res->err_.status = 10;
        // возвращаем NK_EOK
        return NK_EOK;
    }
}

```

Реализация клиентской сущности:

client.c

```

nk_err_t ret = Connection_Send(ctx,
                               &req, &req_arena,
                               &res, NK_NULL);

if (ret == NK_EOK) {
    if (nk_msg_check_err(res)) {
        // обрабатываем логическую ошибку
        // значения возвращаемых error-аргументов метода содержится во внутренних
        // полях структуры сообщения-ответа
        return res.err_.status;
    } else {

```



```

        // если ошибок нет, res содержит значения out-аргументов
        return res.sentLen;
    }
} else {
    // обрабатываем транспортную ошибку
}

```

## Составные имена сущностей, компонентов и интерфейсов

Для сущностей, компонентов и интерфейсов допустимо использовать составные имена, содержащие одну или несколько точек. При этом часть имени, следующая за последней точкой, называется *кратким именем*.

Например, если полное имя сущности – `k1.VfsEntity`, то краткое имя – `VfsEntity`. Интерфейс с полным именем `main.security.Verify` имеет краткое имя `Verify`.

Если имя несоставное (например, `server`), то оно является и кратким, и полным именем одновременно.

### Имена и расположение файлов описаний

Имя EDL-файла должно совпадать с кратким именем сущности, которую он описывает. При этом составное имя используется для формирования пути к EDL-файлу. Например, сущность `k1.VfsEntity` должна быть описана в файле `k1/VfsEntity.edl`.

То же правило действует для компонентов и интерфейсов: имя CDL-файла должно совпадать с кратким именем компонента, а имя IDL-файла должно совпадать с кратким именем интерфейса. При этом составное имя используется для формирования пути к CDL- или IDL-файлу соответственно. Например, компонент `net.Updater` должен описываться в файле `net/Updater.cd1`, интерфейс с полным именем `main.security.Verify` должен описываться в файле `main/security/Verify.idl`.

Краткие имена сущностей, компонентов и интерфейсов не могут содержать символ подчеркивания (`_`). В сегментах составного имени символ подчеркивания разрешен.

### Имена исполняемых файлов для запуска сущностей

При старте решения [сущность Einit](#) запускает другие сущности следующим образом: в ROMFS (в [образе решения](#)) выбирается для запуска исполняемый файл с именем, которое совпадает с кратким именем запускаемой сущности. Например, сущности `Client` и `net.Client` по умолчанию будут запущены из исполняемого файла с именем `Client`. Чтобы при старте решения сущность запускалась из исполняемого файла с другим именем, необходимо в [init-описании](#) использовать ключевое слово `path`.

### Полные имена в описаниях

В EDL-, CDL- и IDL-описаниях, а также в `init`-описании и в конфигурации безопасности используются только *полные имена сущностей, компонентов и интерфейсов*. Для примера возьмем уже упоминавшиеся выше сущность `k1.VfsEntity`, компонент `net.Updater` и интерфейс `main.security.Verify`.

Пример EDL-описания сущности `k1.VfsEntity`:

```
VfsEntity.edl
```

```
entity kl.VfsEntity
...
```

Пример CDL-описания компонента `net.Updater`:

```
Updater.cdl
```

```
component net.Updater
...
```

Пример Init-описания:

```
init.yaml
```

```
entities:
- name: kl.VfsEntity
...
```

Пример конфигурации безопасности:

```
security.psl
```

```
...
/* Объявление сущности "kl.VfsEntity". */
use EDL kl.VfsEntity;
...
/* Конфигурирование запросов для вызова метода Check любых реализаций интерфейса
"Verify". */
request interface=main.security.Verify, method=Check { grant () }
...
```

## Имена NK-сгенерированных методов и типов

Имена [сгенерированных типов и методов](#) построены на основе *полных имен сущностей, компонентов и интерфейсов*. Например, для компонента `net.Updater` будет сгенерирована структура `net_Updater_component`, функция `net_Updater_component_init`, диспетчер `net_Updater_component_dispatch`, а также структуры запроса `net_Updater_component_req` и ответа `net_Updater_component_res`.

## Запуск сущностей

### Сущность Einit

Одной из важнейших сущностей в KasperskyOS является сущность с именем Einit, которая первой запускается ядром операционной системы при загрузке образа. В большинстве решений на базе KasperskyOS сущность Einit запускает все остальные сущности, входящие в решение, то есть служит *инициализирующей сущностью*.

В составе пакета инструментов KasperskyOS Community Edition поставляется [утилита einit](#), которая позволяет сгенерировать код сущности Einit на языке C на основе файла [init.yaml](#) (так называемого *init-описания*). Сущность Einit, созданная с помощью скрипта einit, выполняет следующие инициализирующие функции:

- создает все сущности, входящие в решение;
- создает необходимые соединения (IPC-каналы) между сущностями;
- копирует в окружение каждой сущности информацию о ее соединениях;
- запускает сущности.

Стандартным способом использования утилиты einit является интеграция ее вызова в один из шагов сборочного скрипта, в результате которого утилита einit на основе файла [init.yaml](#) сгенерирует файл [einit.c](#), содержащий код сущности Einit. На одном из следующих шагов сборочного скрипта необходимо скомпилировать файл [einit.c](#) в исполняемый файл сущности Einit и включить в образ решения.

Для сущности Einit не требуется создавать файлы [статических описаний](#). Эти файлы поставляются в составе пакета инструментов KasperskyOS Community Edition и автоматически подключаются при сборке решения. При этом сущность Einit должна быть описана в файле [security.psl](#).

## Файл init.yaml

Файл [init.yaml](#) (*init-описание*) используется утилитой einit для генерации исходного кода [инициализирующей сущности Einit](#). Этот файл содержит данные в формате YAML, которые идентифицируют:

- сущности, запускаемые при загрузке KasperskyOS;
- IPC-каналы, используемые сущностями для взаимодействия между собой.

Эти данные представляют собой словарь с ключем [entities](#), содержащий список словарей сущностей. Ключи словаря сущности приведены в таблице ниже.

Ключи словаря сущности в *init-описании*

Ключ	Обязательный	Описание
name	Да	Имя сущности
task	Нет	Идентификатор сущности, который по умолчанию совпадает с именем сущности. У каждой сущности должен быть уникальный идентификатор. Можно запустить несколько сущностей с одним именем, но разными идентификаторами.
path	Нет	Путь к исполняемому файлу в ROMFS (в <a href="#">образе решения</a> ), из которого будет запущена сущность. По умолчанию сущность будет запущена из файла в ROMFS с именем, совпадающим с <a href="#">кратким именем сущности</a> .

		<p>Например, сущности <code>Client</code> и <code>net.Client</code> по умолчанию будут запущены из файла <code>Client</code>.</p> <p>Можно запустить несколько сущностей с одним именем из разных исполняемых файлов. При этом идентификаторы этих сущностей должны быть разными.</p>
connections	Нет	<p>Ключ словаря, содержащего список словарей <a href="#">IPC-каналов</a> сущности. Этот список задает статически создаваемые IPC-каналы, клиентскими дескрипторами которых будет владеть сущность. По умолчанию список пуст. (Помимо статически создаваемых IPC-каналов, сущности могут использовать <a href="#">динамически создаваемые IPC-каналы</a>.)</p>

Ключи словаря IPC-каналов сущности приведены в таблице ниже.

Ключи словаря IPC-каналов сущности в init-описании

Ключ	Обязательный	Описание
id	Да	<p>Идентификатор IPC-канала, который может быть задан как конкретным значением, так и ссылкой вида</p> <pre>{var: &lt;имя константы&gt;, include: &lt;путь к заголовочному файлу&gt;}</pre> <p>У каждого IPC-канала должен быть уникальный идентификатор.</p> <p>(Идентификатор IPC-канала используется сущностями, чтобы <a href="#">получить IPC-дескриптор</a>.)</p>
target	Да	Идентификатор сущности, которая будет владеть серверным дескриптором IPC-канала

## Примеры init-описаний

В приведенных примерах файл с init-описанием называется `init.yaml`, хотя может иметь любое имя.

`init.yaml`

```
# init-описание решения, содержащего сущность-клиент и сущность-сервер
entities:
# Сущность Client будет отправлять запросы сущности Server.
- name: Client
  connections:
    # Идентификатор сущности-сервера, которой сущность Client будет
    # отправлять запросы
    - target: Server
      # Идентификатор IPC-канала для обмена IPC-сообщениями
      # между сущностями
      id: server_connection
# Сущность Server будет выступать в роли сервера
# (будет отвечать на запросы сущности Client).
- name: Server
```

`init.yaml`

```
# init-описание, где идентификатор IPC-канала задан ссылкой
entities:
- name: Client
  connections:
```

```

- target: Server
  # Идентификатор IPC-канала содержится в константе SERVER_CONN
  # в файле src/example.h
  id: {var: SERVER_CONN, include: src/example.h}
- name: Server

```

#### init.yaml

```

# init-описание, где заданы пути к исполняемым файлам, из
# которых будут запущены сущности Client, ClientServer и
# MainServer
entities:
- name: Client
  path: cl
  connections:
    - target: ClientServer
      id: server_connection_cs
- name: ClientServer
  path: srvs/csr
  connections:
    - target: MainServer
      id: server_connection_ms
- name: MainServer
  path: srvs/msr

```

#### init.yaml

```

# init-описание, при котором сущности MainServer и BkServer
# будут запущены из одного исполняемого файла
entities:
- name: Client
  connections:
    - id: server_connection_ms
      target: MainServer
    - id: server_connection_bs
      target: BkServer
- name: MainServer
  path: srv
- name: BkServer
  path: srv

```

#### init.yaml

```

# init-описание, при котором будут запущены две сущности
# Server с разными идентификаторами из одного исполняемого
# файла
entities:
- name: Client
  connections:
    - id: server_connection_us
      # Идентификатор сущности-сервера
      target: UserServer
    - id: server_connection_ps
      # Идентификатор сущности-сервера
      target: PrivilegedServer
- task: UserServer
  name: Server

```

```
- task: PrivilegedServer
  name: Server
```

## IPC и транспорт

### Реализация IPC

## Основы IPC в KasperskyOS

В KasperskyOS единственным способом межпроцессного взаимодействия (IPC) является обмен сообщениями.

### Виды сообщений и роли сущностей

Обмен сообщениями в KasperskyOS построен по клиент-серверной модели.

Во взаимодействии двух сущностей одна из них является клиентом (клиентская сущность), а вторая — сервером (серверная сущность). Клиент инициирует взаимодействие, отправляя сообщение-запрос (далее также "запрос"). Сервер получает запрос и отвечает на него, отправляя клиенту сообщение-ответ (далее также "ответ").

### IPC-каналы

Чтобы две сущности могли обмениваться сообщениями, между ними должен быть установлен [IPC-канал](#) (далее также "канал" или "соединение"). Каждая сущность может иметь несколько соединений, выступая как клиент для одних сущностей и как сервер — для других.

### Системные вызовы

KasperskyOS предоставляет три системных вызова для обмена IPC-сообщениями: `Call`, `Recv` и `Reply`. Соответствующие функции объявлены в файле `syscalls.h`:

- `Call()` — используется клиентом для отправки запроса и получения ответа. Принимает IPC-дескриптор, буфер с запросом и буфер под ответ.
- `Recv()` — используется сервером для получения запроса. Принимает IPC-дескриптор и буфер под запрос.
- `Reply()` — используется сервером для отправки ответа. Принимает IPC-дескриптор и буфер с ответом.

Функции `Call()`, `Recv()` и `Reply()` возвращают `rcOk` или код ошибки.

Вызовы `Call()` и `Recv()` являются блокирующими, то есть обмен сообщениями происходит по принципу рандеву:

- Серверный поток, выполнивший вызов `Recv()`, остается заблокированным до момента получения запроса.
- Клиентский поток, выполнивший вызов `Call()`, остается заблокированным до момента получения ответа от сервера.

Системные вызовы редко используются в коде сущности. Вместо них рекомендуется использовать более удобные NK-сгенерированные методы, которые, в свою очередь, выполняют системные вызовы.

## IPC-каналы

IPC-каналы соединяют сущности друг с другом и используются для обмена IPC-сообщениями.

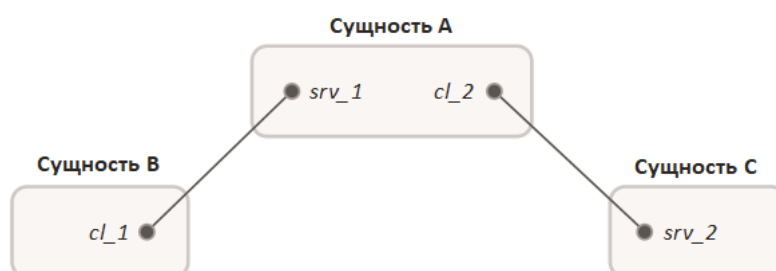
Канал представлен парой IPC-дескрипторов: *клиентским* и *серверным*, которые связаны друг с другом.



Две сущности соединены IPC-каналом

Канал является направленным. Сущность, которой принадлежит клиентский дескриптор канала (*сущность-клиент*), может отправлять запросы и получать ответы по этому каналу. Сущность, которой принадлежит серверный дескриптор канала (*сущность-сервер*), может получать запросы и отправлять ответы по этому каналу.

Сущность может иметь несколько соединений (каналов) с другими сущностями, выступая как клиент в одних соединениях, и как сервер – в других.



Сущность А является клиентом для сущности С и сервером – для сущности В. Обозначения: `cl_1`, `cl_2` – клиентские IPC-дескрипторы; `srv_1`, `srv_2` – серверные IPC-дескрипторы.

## Создание и использование каналов

IPC-каналы создаются статически и динамически.

Статически созданные IPC-каналы создаются сущностью `Einit` при старте решения. Код сущности `Einit` генерируется на основе [init-описания](#), в котором указываются все каналы (т.е. соединения), которые нужно создать.

Помимо статически создаваемых IPC-каналов, сущности могут использовать [динамически создаваемые IPC-каналы](#).

Чтобы отправить и принять IPC-сообщение по конкретному каналу, нужно знать соответствующие значения IPC-дескрипторов (клиентского и серверного). Для получения значений IPC-дескрипторов используется [локатор сервисов](#).

## Динамическое создание IPC-каналов

Возможность динамического создания IPC-каналов позволяет изменять топологию взаимодействия сущностей "на лету". Это требуется, если неизвестно, какая именно сущность-сервер станет поставщиком ресурса, требуемого сущности-клиенту. Например, может быть неизвестно, на какой именно накопитель нужно будет записывать данные.

Динамически созданный IPC-канал, в отличие от статически созданного, имеет следующие особенности:

- создается между запущенными сущностями;
- создается совместно сущностью-клиентом и сущностью-сервером;
- предусматривает удаление;
- предусматривает использование сервера имен (сущности `k1.core.NameServer`), который обеспечивает передачу сведений о доступных реализациях интерфейсов от сущностей-серверов к сущностям-клиентам.

При динамическом создании IPC-канала используются функции:

- интерфейса сервера имен (Name Server);
- менеджера соединений (Connection Manager).

Интерфейс сервера имен представлен для пользователя следующими файлами:

- `coresrv/ns/ns_api.h` – заголовочный файл библиотеки `libkos`;
- `coresrv/ns/NameServer.idl` – описание IPC-интерфейса сервера имен на языке IDL.

Менеджер соединений представлен для пользователя следующими файлами:

- `coresrv/cm/cm_api.h` – заголовочный файл библиотеки `libkos`;
- `services/cm/CM.idl` – описание IPC-интерфейса менеджера соединений на языке IDL.

Динамическое создание IPC-канала осуществляется по следующему сценарию:



1. Запускаются сущность-клиент, сущность-сервер и сервер имен.
2. Сущность-сервер подключается к серверу имен функцией `NsCreate()` и публикует имя сущности-сервера, имя интерфейса и имя реализации интерфейса функцией `NsPublishService()`.
3. Сущность-клиент подключается к серверу имен функцией `NsCreate()` и выполняет поиск имени сущности-сервера и имени реализации интерфейса по имени интерфейса функцией `NsEnumServices()`.
4. Сущность-клиент запрашивает доступ к реализации интерфейса функцией `KnCmConnect()`, передавая ей в качестве аргументов найденные имя сущности-сервера и имя реализации интерфейса.
5. Сущность-сервер вызывает функцию `KnCmListen()` для проверки наличия запроса доступа к предоставляемой реализации интерфейса от сущности-клиента.
6. Сущность-сервер принимает запрос доступа к предоставляемой реализации интерфейса функцией `KnCmAccept()`, передавая ей в качестве аргументов имя сущности-клиента и имя реализации интерфейса, которые получены функцией `KnCmListen()`.

Для использования сервера имен политика безопасности решения (`security.ps1`) должна разрешать взаимодействие сущности `kl.core.NameServer` с сущностями, между которыми могут динамически создаваться IPC-каналы.

Сущность-сервер может снимать с публикации на сервере имен ранее опубликованные реализации интерфейсов функцией `NsUnPublishService()`.

Сущность-сервер может отклонять запросы доступа к реализациям интерфейсов функцией `KnCmDrop()`.

#### ns\_api.h (фрагмент)

```
/**
 * Функция осуществляет попытку подключиться к серверу имен name
 * в течение msecс миллисекунд. Если параметр name имеет значение
 * RTL_NULL, функция пытается подключиться к серверу имен ns
 * (серверу имен по умолчанию). Выходной параметр ns содержит дескриптор
 * соединения с сервером имен.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode NsCreate(const char *name, rtl_uint32_t msecс, NsHandle *ns);

/**
 * Функция публикует реализацию интерфейса на сервере имен.
 * Параметр ns задает дескриптор соединения с сервером имен.
 * Параметр server задает имя сущности-сервера (например, ata).
 * Параметр type задает имя публикуемого интерфейса (например, IBlkDev).
 * Параметр service задает имя реализации публикуемого интерфейса
 * (например, blkdev.blkdev).
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode NsPublishService(NsHandle ns, const char *type, const char *server,
                          const char *service);

/**
 * Функция снимает с публикации реализацию интерфейса на сервере имен.
 * Параметр ns задает дескриптор соединения с сервером имен.
 * Параметр server задает имя сущности-сервера. Параметр type задает имя
 * опубликованного интерфейса. Параметр service задает имя реализации
```

```

* опубликованного интерфейса.
* В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
*/
Retcode NsUnPublishService( NsHandle ns, const char *type, const char *server,
                             const char *service);

/**
* Функция перечисляет реализации интерфейса, опубликованные на сервере
* имен. Параметр ns задает дескриптор соединения с сервером имен.
* Параметр type задает имя требуемого интерфейса. Параметр index
* задает индекс для перечисления реализаций интерфейса.
* Выходной параметр server содержит имя сущности-сервера, предоставляющей
* реализацию интерфейса. Выходной параметр service содержит имя реализации
* интерфейса.
* Например, перечисление реализаций интерфейса IBkDev осуществляется
* следующим образом.
* rc = NsEnumServices(ns, "IBkDev", 0, outServerName, outServiceName);
* rc = NsEnumServices(ns, "IBkDev", 1, outServerName, outServiceName);
* ...
* rc = NsEnumServices(ns, "IBkDev", N, outServerName, outServiceName);
* Вызовы функции с инкрементированием индекса продолжается до тех пор,
* пока функция не вернет rcResourceNotFound.
* В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
*/
Retcode NsEnumServices(NsHandle ns, const char *type, unsigned index,
                        char *server, char *service);

```

#### cm\_api.h (фрагмент)

```

/**
* Функция запрашивает доступ к реализации интерфейса с именем service,
* предоставляемой сущностью-сервером server. Параметр msecс задает время
* ожидания принятия запроса сущностью-сервером в миллисекундах. Выходной
* параметр endpoint содержит клиентский IPC-дескриптор. Выходной параметр
* rsid содержит идентификатор реализации интерфейса.
* В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
*/
Retcode KnCmConnect(const char *server, const char *service,
                    rtl_uint32_t msecс, Handle *endpoint,
                    rtl_uint32_t *rsid);

/**
* Функция проверяет наличие запроса доступа к реализации интерфейса
* с именем filter. Если параметр filter имеет значение RTL_NULL,
* проверяется наличие запроса доступа к любой реализации
* интерфейса. Параметр msecс задает время ожидания запроса
* в миллисекундах. Выходной параметр client содержит имя
* сущности-клиента. Выходной параметр service содержит имя реализации
* интерфейса.
* В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
*/
Retcode KnCmListen(const char *filter, rtl_uint32_t msecс, char *client,
                    char *service);

/**
* Функция отклоняет запрос доступа к реализации интерфейса
* с именем service от сущности-клиента client.
* В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
*/

```

```

Retcode KnCmDrop(const char *client, const char *service);

/**
 * Функция принимает запрос доступа к реализации интерфейса
 * с именем service от сущности-клиента client. Параметр rsid задает
 * идентификатор реализации интерфейса. Параметр listener задает
 * слушающий дескриптор. Если параметр listener имеет значение
 * INVALID_HANDLE, создается новый слушающий дескриптор.
 * Выходной параметр endpoint содержит серверный IPC-дескриптор.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnCmAccept(const char *client, const char *service, rtl_uint32_t rsid,
                    Handle listener, Handle *endpoint);

```

Параметр `filter` функции `KnCmListen()` является зарезервированным и не влияет на ее поведение. Поведение функции соответствует значению `RTL_NULL` параметра `filter`.

**Слушающий дескриптор** – серверный IPC-дескриптор с расширенными правами. Он создается при вызове функции `KnCmAccept()` с аргументом `INVALID_HANDLE` в параметре `listener`. Если при вызове функции `KnCmAccept()` передать ей слушающий дескриптор, то созданный серверный IPC-дескриптор обеспечит возможность получать запросы по всем IPC-каналам, ассоциированным с этим слушающим дескриптором. (Первый IPC-канал, ассоциированный со слушающим дескриптором, создается при вызове функции `KnCmAccept()` с аргументом `INVALID_HANDLE` в параметре `listener`. Второй и последующие IPC-каналы, ассоциированные со слушающим дескриптором, создаются при втором и последующих вызовах функции `KnCmAccept()` с передачей ей слушающего дескриптора, полученного при первом вызове.)

Слушающий дескриптор используется также при статическом создании IPC-каналов. Он создается при вызове функции `KnHandleConnectEx()` с аргументом `INVALID_HANDLE` в параметре `srListener`. Если при вызове функции `KnHandleConnectEx()` передать ей слушающий дескриптор, то созданный серверный IPC-дескриптор обеспечит возможность получать запросы по всем IPC-каналам, ассоциированным с этим слушающим дескриптором. Ассоциация слушающего дескриптора с несколькими IPC-каналами происходит, когда для одной сущности-сервера создается несколько IPC-каналов с одинаковыми именами.

Если динамически созданный IPC-канал больше не требуется, его клиентский и серверный дескрипторы нужно удалить. При необходимости IPC-канал может быть создан снова.

## Общая схема обмена сообщениями

Рассмотрим две сущности ("клиент" и "сервер"), между которыми установлен IPC-канал. Пусть `cl` – клиентский IPC-дескриптор этого канала, `sr` – серверный IPC-дескриптор этого канала.

Приведенный ниже код предназначен для демонстрации механизма IPC. Обычно в коде сущности системные вызовы не используются напрямую. Для удобного обмена сообщениями предназначены специальные [NK-сгенерированные методы](#), которые, в свою очередь, используют системные вызовы.

Код сущности-клиента:

```
client.c
```

...

```
// Получение клиентского IPC-дескриптора c1 с помощью локатора сервисов
...
// Отправка запроса
Call(c1, &RequestBuffer, &ResponseBuffer);
...
```

Код сущности-сервера:

server.c

```
...
// Получение серверного IPC-дескриптора sr с помощью локатора сервисов
...
// Получение запроса
Recv(sr, &RequestBuffer);
...
// Обработка запроса
...
// Отправка ответа
Reply(sr, &ResponseBuffer);
...
```

Обмен сообщениями происходит следующим образом:

1. Один из потоков клиента выполняет системный вызов `Call()`, передав в аргументах дескриптор `c1` (клиентский дескриптор используемого канала), указатель на буфер с сообщением-запросом и указатель на буфер для ответа.
2. Сообщение-запрос передается подсистеме Kaspersky Security System для проверки. Если Kaspersky Security System возвращает решение "разрешено", переходим к пункту 3. В противном случае вызов `Call()` завершается с кодом ошибки `rcSecurityDisallow`, переходим к пункту 9.
3. Если сервер ожидает запрос от этого клиента (уже выполнил вызов `Recv()`), передав первым аргументом `sr`), переходим к пункту 4. В противном случае поток клиента остается заблокированным до тех пор, пока один из потоков сервера не выполнит системный вызов `Recv()` с первым аргументом `sr`.
4. Сообщение-запрос копируется в адресное пространство сервера. Поток сервера разблокируется, а вызов `Recv()` завершается с кодом `rcOk`.
5. Сервер обрабатывает полученное сообщение. Поток клиента остается заблокированным.
6. Сервер выполняет системный вызов `Reply()`, передав в аргументах дескриптор `sr` и указатель на буфер с сообщением-ответом.
7. Сообщение-ответ передается подсистеме Kaspersky Security System для проверки. Если Kaspersky Security System возвращает решение "разрешено", переходим к пункту 8. В противном случае вызовы `Call()` и `Reply()` завершаются с кодом ошибки `rcSecurityDisallow` (см. пункт 3).
8. Сообщение-ответ копируется в адресное пространство клиента. Поток сервера разблокируется, вызов `Reply()` завершается с кодом `rcOk`. Поток клиента разблокируется, вызов `Call()` завершается с кодом `rcOk`.
9. Обмен завершен.

Если в процессе передачи запроса произошла ошибка (нехватка памяти, неверный формат сообщения и т.п.), то потоки разблокируются, а вызовы `Call()` и `Reply()` возвращают код ошибки.

## Локатор сервисов (Service Locator)

Локатор сервисов — небольшая библиотека, которая позволяет:

- узнать значение IPC-дескриптора по имени соединения;
- узнать значение идентификатора интерфейса (RIID) по имени реализации интерфейса.

Значения IPC-дескриптора и RIID необходимы для обращения к конкретному интерфейсу конкретной серверной сущности. Эти значения используются при инициализации [транспорта](#).

Чтобы использовать локатор сервисов, нужно в коде сущности подключить файл `sl_api.h`:

```
#include <coresrv/sl/sl_api.h>
```

### Основные функции локатора сервисов

Функции, используемые на клиентской стороне:

- `ServiceLocatorConnect()` — принимает имя соединения и возвращает клиентский IPC-дескриптор, соответствующий этому соединению (каналу).
- `ServiceLocatorGetRiid()` — принимает IPC-дескриптор и имя реализации интерфейса в формате `<имя экземпляра компонента>.<имя реализации интерфейса>`. Возвращает соответствующий идентификатор RIID (порядковый номер реализации интерфейса).

Имя соединения указывается в файле [init.yaml](#), имя экземпляра компонента — в [EDL-файле](#), а имя реализации интерфейса — в [CDL-файле](#).

Функции, используемые на серверной стороне:

- `ServiceLocatorRegister()` — принимает имя соединения и возвращает серверный IPC-дескриптор, соответствующий этому соединению (каналу). Если канал входит в [группу](#), то функция вернет слушающий дескриптор этой группы.

### Пример использования локатора сервисов

Рассмотрим следующее решение, содержащее две сущности — `Client` и `Server`.

Клиентская сущность не реализует ни одного интерфейса.

```
Client.edl
```

```
entity Client
```

Серверная сущность содержит экземпляр компонента `Ping`. Имя экземпляра: `ping_comp`.

#### Server.edl

```
entity Server
components {
    ping_comp: Ping
}
```

Пусть компонент `Ping` содержит именованную реализацию интерфейса `Ping`, объявленного в файле `Ping.idl`. Имя реализации: `ping_impl`.

#### Ping.cdl

```
component Ping
interfaces {
    ping_impl: Ping
}
```

(Для краткости файл `Ping.idl` не приведен.)

В `init`-описании укажем, что сущности `Client` и `Server` должны быть соединены каналом с именем `server_connection`:

#### init.yaml

```
entities:

- name: Client
  connections:
    - target: Server
      id: server_connection

- name: Server
```

Использование локатора сервисов на стороне клиента:

#### client.c

```
...
/* Подключаем локатор сервисов */
#include <coresrv/sl/sl_api.h>
...
/* Получаем клиентский IPC-дескриптор канала "server_connection" */
Handle handle = ServiceLocatorConnect("server_connection");
...
/* Получаем идентификатор (RIID) реализации ping_impl, содержащейся в экземпляре
ping_comp */
nk_iid_t riid = ServiceLocatorGetRiid(handle, "ping_comp.ping_impl");
...
```

Использование локатора сервисов на стороне сервера:

#### server.c

```
...
```

```

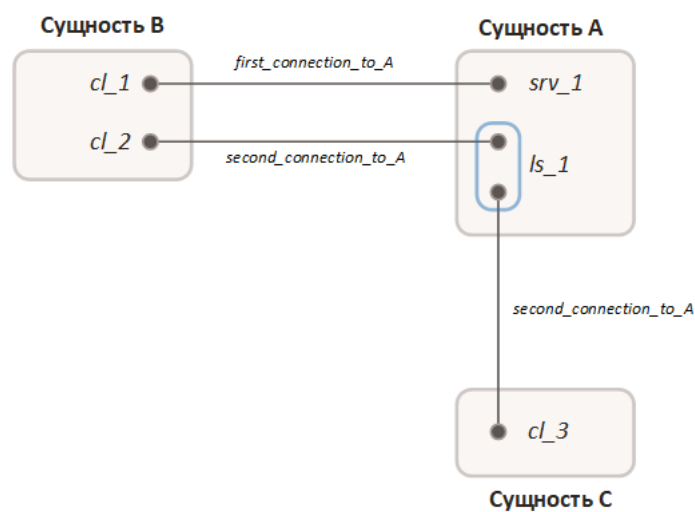
/* Подключаем локатор сервисов */
#include <coresrv/sl/sl_api.h>
...
nk_iid_t iid;
/* Получаем серверный IPC-дескриптор канала "server_connection" */
Handle handle = ServiceLocatorRegister("server_connection", NULL, 0, &iid);
...

```

## Группы каналов

Каналы, в которых сущность выступает сервером, могут быть объединены в одну или несколько *групп*.

Каждой группе каналов соответствует свой *слушающий дескриптор* (listener handle).



Два IPC-канала имеют одинаковое имя "second\_connection\_to\_A" и объединены в группу. Еще один канал с именем "first\_connection\_to\_A" не входит в эту группу. Обозначения: cl\_1, cl\_2, cl\_3 — клиентские IPC-дескрипторы; srv\_1 — серверный IPC-дескриптор; ls\_1 — слушающий IPC-дескриптор.

Слушающий дескриптор позволяет серверу принимать запросы сразу по всем каналам из группы. При этом нет необходимости создавать отдельный поток на каждый канал — достаточно в одном потоке сервера выполнить системный вызов `Recv()`, указав слушающий дескриптор.

## Создание группы каналов с помощью init-описания

Чтобы каналы образовали группу, они должны соединяться с одной серверной сущностью и иметь одинаковые имена. Например, чтобы получить систему каналов, изображенную выше, можно использовать следующее init-описание:

init.yaml

### entities:

```

# Сущность "A" выступает как сервер, поэтому ее список соединений пуст.
- name: A

# Сущность "B" будет соединена сущностью "A" двумя разными каналами.
- name: B

```

```

connections:
- target: A
  id: first_connection_to_A
- target: A
  id: second_connection_to_A

# Сущность "C" будет соединена с сущностью "A" каналом с именем
# "second_connection_to_A". Два канала с одинаковыми именами будут объединены
# в группу: на стороне сущности "A" они будут иметь один и тот же
# IPC-дескриптор (слушающий дескриптор ls_1).
- name: C
  connections:
  - target: A
    id: second_connection_to_A

```

## Обмен сообщениями в случае группы каналов

Рассмотрим, как происходит обмен сообщениями для группы каналов, описанной выше.

Клиентские сущности В и С получают значение клиентского дескриптора по имени соединения `first_connection_to_A` и отправляют запрос:

`entity_B.c, entity_C.c`

```

...
// Получение клиентского IPC-дескриптора cl, соответствующего соединению "
second_connection_to_A".
Handle cl = ServiceLocatorConnect("second_connection_to_A");
...
// Отправка запроса
Call(cl, &RequestBuffer, &ResponseBuffer);
...

```

Оба используемых канала имеют одно имя — `second_connection_to_A`. Однако по этому имени сущности В и С получают разные значения дескрипторов: сущность В получит значение `cl_2`, а сущность С — значение `cl_3`.

Серверная сущность А получает значение слушающего дескриптора `ls_1`. Далее сущность А ожидает запросы по двум каналам сразу (от В и от С), используя дескриптор `ls_1`. После получения и обработки запроса сущность А отправляет ответ, используя дескриптор `ls_1`. Ответ будет отправлен тому клиенту, который инициировал запрос:

`entity_A.c`

```

...
nk_iid_t iid;
// Получение слушающего дескриптора ls_1, соответствующего соединению
"second_connection_to_A"
Handle ls_1 = ServiceLocatorRegister("second_connection_to_A", NULL, 0, &iid);
...
// Ожидание запроса от сущности В или С
Recv(ls_1, &RequestBuffer);
...
// Обработка поступившего запроса

```



```
...  
// Отправка ответа тому клиенту, от которого пришел запрос  
Reply(ls_1, &ResponseBuffer);  
...
```

## Транспорт

## Структура IPC-сообщения

В KasperskyOS все взаимодействия между сущностями статически типизированы. Допустимые структуры IPC-сообщения определяются IDL-описанием интерфейсов сущности-получателя сообщения (сервера).

Корректное IPC-сообщение (как запрос, так и ответ) содержит *фиксированную часть* и *арену*.

### Фиксированная часть сообщения

Фиксированная часть сообщения содержит аргументы фиксированного размера, а также RIID и MID.

Аргументы фиксированного размера – это аргументы любых IDL-типов, кроме типа `sequence`.

RIID и MID идентифицируют вызываемый интерфейс и метод:

- RIID (Runtime Implementation ID) является порядковым номером вызываемой реализации интерфейса сущности (начиная с нуля).
- MID (Method ID) является порядковым номером метода в содержащем его интерфейсе (начиная с нуля).

Тип фиксированной части сообщения генерируется компилятором NK на основе IDL-описания интерфейса. Для каждого метода интерфейса генерируется отдельная структура. Также генерируются типы `union` для хранения любого запроса к серверу, компоненту или интерфейсу.

Например, для метода `Ping` интерфейса `Ping` (компонент `Ping` сущности `Server` в примере [echo](#)) компилятор NK создаст тип фиксированной части запроса `Ping_Ping_req` и фиксированной части ответа `Ping_Ping_res`. Также будут сгенерированы следующие типы `union`:

- `Ping_req` и `Ping_res` – фиксированные части запроса и ответа для любого метода интерфейса `Ping`;
- `Ping_component_req` и `Ping_component_res` – фиксированные части запроса и ответа для любого метода любого интерфейса, реализация которого включена в компонент `Ping`;  
При наличии вложенных компонентов эти типы также содержат структуры фиксированной части сообщения для любых методов любых интерфейсов, реализации которых включены во все вложенные компоненты. Подробнее см. [Сгенерированные методы и типы](#).
- `Server_entity_req` и `Server_entity_res` – фиксированные части запроса и ответа для любого метода любого интерфейса, реализация которого включена в любой компонент, экземпляр которого входит в сущность `Server`.

## Арена

Арена представляет собой буфер для хранения аргументов переменного размера (IDL-тип [sequence](#)).

## Проверка сообщения в Kaspersky Security System

Проверку структуры отправляемого сообщения на корректность выполняет подсистема Kaspersky Security System. Проверяются как запросы, так и ответы. Если сообщение имеет некорректную структуру, оно будет отклонено без вызова связанных с ним политик безопасности.

## Формирование структуры сообщения

В составе KasperskyOS Community Edition поставляются следующие инструменты, позволяющие упростить для разработчика создание и упаковку IPC-сообщения:

- Библиотека `transport-kos` для работы с транспортом [NkKosTransport](#).
- [Компилятор NK](#), позволяющий сгенерировать [специальные методы и типы](#).

Формирование простейшего IPC-сообщения показано в примере [echo](#).

## Транспорт NkKosTransport

Транспорт `NkKosTransport` является удобной надстройкой над системными вызовами `Call`, `Recv` и `Reply`. Он позволяет работать отдельно с фиксированной частью и ареной сообщений.

Структура `NkKosTransport` и методы работы с ней объявлены в файле `transport-kos.h`.

Чтобы инициализировать транспорт, достаточно вызвать функцию `NkKosTransport_Init()`, указав IPC-дескриптор канала, по которому должны передаваться сообщения (`handle`):

```
#include <coresrv/nk/transport-kos.h>
...
NkKosTransport transport;
NkKosTransport_Init(&transport, handle, NK_NULL, 0);
```

Канал имеет два IPC-дескриптора: клиентский и серверный. Поэтому на стороне клиента при инициализации транспорта нужно указать клиентский дескриптор, а на стороне сервера – серверный дескриптор используемого канала.

Для вызова транспорта используются функции `nk_transport_call()`, `nk_transport_recv()` и `nk_transport_reply()`, объявленные в `transport.h` (подключен в файле `transport-kos.h`).

Функция `nk_transport_call()` предназначена для отправки запроса и получения ответа:

```

/* Фиксированная часть (req) и арена запроса (req_arena) должны быть инициализированы
 * фактическими входными аргументами вызываемого метода. Фиксированная часть (req)
 * должна также содержать значения RIID и MID.
 * Функция nk_transport_call() формирует запрос и выполняет системный вызов Call.
 * Полученный от сервера ответ помещается в res (фиксированная часть ответа) и
 * res_arena (арена ответа). */
nk_transport_call(&transport.base, (struct nk_message *)&req, &req_arena, (struct
nk_message *)&res, &res_arena);

```

При использовании на стороне клиента сгенерированного интерфейсного метода (например, метод `IPing_Ping` из примера [echo](#)) в запросе автоматически проставляются соответствующие значения RIID и MID, после чего вызывается функция `nk_transport_call()`.

Функция `nk_transport_recv()` предназначена для получения запроса:

```

/* Функция nk_transport_recv () выполняет системный вызов Recv.
 * Полученный от клиента запрос помещается в req (фиксированная часть ответа) и
 * req_arena (арена ответа). */
nk_transport_recv(&transport.base, (struct nk_message *)&req, &req_arena);

```

Функция `nk_transport_reply()` предназначена для отправки ответа:

```

/* Фиксированная часть (res) и арена запроса (res_arena) должны быть инициализированы
 * фактическими выходными аргументами вызываемого метода сервера.
 * Функция nk_transport_reply() формирует ответ и выполняет системный вызов Reply. */
nk_transport_reply(&transport.base, (struct nk_message *)&res, &res_arena);

```

## Сгенерированные методы и типы

При сборке решения [компилятор NK](#) на основе [EDL-, CDL- и IDL-описаний](#) генерирует набор специальных методов и типов, упрощающих формирование, отправку, прием и обработку IPC-сообщений.

Рассмотрим статическое описание сущности `Server` из примера [echo](#). Это описание состоит из трех файлов: `Server.edl`, `Ping.cdl` и `Ping.idl`:

### Server.edl

```

/* Описание сущности Server. */
entity Server

/* pingComp - именованный экземпляр компонента Ping. */
components {
    pingComp: Ping
}

```

### Ping.cdl

```

/* Описание компонента Ping. */
component Ping

/* pingImpl - реализация интерфейса Ping. */
interfaces {
    pingImpl: Ping
}

```

#### Ping.idl

```

/* Описание интерфейса Ping. */
package Ping

interface {
    Ping(in UInt32 value, out UInt32 result);
}

```

На основе этих файлов будут сгенерированы файлы `Server.edl.h`, `Ping.cd1.h`, и `Ping.idl.h` содержащие следующие методы и типы:

### Методы и типы, общие для клиента и сервера

- **Абстрактные интерфейсы, содержащие указатели на реализации входящих в них методов.**

В нашем примере будет сгенерирован один абстрактный интерфейс – `Ping`:

```

struct Ping_ops {
    nk_err_t (*Ping)(struct Ping *,
                     const struct Ping_req *,
                     const struct nk_arena *,
                     struct Ping_res *,
                     struct nk_arena *); };

struct Ping {
    const struct Ping_ops *ops;
};

```

- **Набор интерфейсных методов.**

При вызове интерфейсного метода в запросе автоматически проставляются соответствующие значения [RIID и MID](#), после чего вызывается функция [nk\\_transport\\_call\(.\)](#).

В нашем примере будет сгенерирован единственный интерфейсный метод `Ping_Ping`:

```

nk_err_t Ping_Ping(struct Ping *,
                   const struct Ping_Ping_req *,
                   const struct nk_arena *,
                   struct Ping_Ping_res *,
                   struct nk_arena *);

```

### Методы и типы, используемые только на клиенте

- **Типы прокси-объектов.**

Прокси-объект используется как аргумент интерфейсного метода. В нашем примере будет сгенерирован единственный тип прокси-объекта `Ping_proxy`:

```
struct Ping_proxy {
    struct Ping base;
    struct nk_transport *transport;
    nk_iid_t iid;
};
```

- **Функции для инициализации прокси-объектов.**

В нашем примере будет сгенерирована единственная инициализирующая функция `Ping_proxy_init`:

```
void Ping_proxy_init(struct Ping_proxy *, struct nk_transport *, nk_iid_t);
```

- **Типы, определяющие структуру фиксированной части сообщения для каждого конкретного метода.**

В нашем примере будет сгенерировано два таких типа: `Ping_Ping_req` (для запроса) и `Ping_Ping_res` (для ответа).

```
struct Ping_Ping_req {
    struct nk_message base_;
    nk_uint32_t value;
};

struct Ping_Ping_res {
    struct nk_message base_;
    nk_uint32_t result;
};
```

## Методы и типы, используемые только на сервере

- **Тип, содержащий реализации всех интерфейсов компонента, а также инициализирующая функция. (Для каждого компонента сервера.)**

При наличии вложенных компонентов этот тип также содержит их экземпляры, а инициализирующая функция принимает соответствующие им инициализированные структуры. Таким образом, при наличии вложенных компонентов, их инициализацию необходимо начинать с самого вложенного.

В нашем примере будет сгенерирована структура `Ping_component` и функция `Ping_component_init`:

```
struct Ping_component {
    struct Ping *pingImpl;
};

void Ping_component_init(struct Ping_component *, struct Ping *);
```

- **Тип, содержащий реализации всех интерфейсов предоставляемых сущностью-сервером непосредственно; все экземпляры компонентов, входящие в сущность-сервер; а также инициализирующая функция.**

В нашем примере будет сгенерирована структура `Server_entity` и функция `Server_entity_init`:

```
struct Server_entity {
    struct Ping_component *pingComp;
};

void Server_entity_init(struct Server_entity *, struct Ping_component *);
```

- Типы, определяющие структуру фиксированной части сообщения для любого метода конкретного интерфейса.

В нашем примере будет сгенерировано два таких типа: `Ping_req` (для запроса) и `Ping_res` (для ответа).

```
union Ping_req {
    struct nk_message base_;
    struct Ping_Ping_req Ping;
};

union Ping_res {
    struct nk_message base_;
    struct Ping_Ping_res Ping;
};
```

- Типы, определяющие структуру фиксированной части сообщения для любого метода любого интерфейса, реализация которого включена в конкретный компонент.

При наличии вложенных компонентов эти типы также содержат структуры фиксированной части сообщения для любых методов любых интерфейсов, реализации которых включены во все вложенные компоненты.

В нашем примере будет сгенерировано два таких типа: `Ping_component_req` (для запроса) и `Ping_component_res` (для ответа).

```
union Ping_component_req {
    struct nk_message base_;
    union Ping_req pingImpl;
};

union Ping_component_res {
    struct nk_message base_;
    union Ping_res pingImpl;
};
```

- Типы, определяющие структуру фиксированной части сообщения для любого метода любого интерфейса, реализация которого включена в любой компонент, экземпляр которого входит в серверную сущность.

При наличии вложенных компонентов эти типы также содержат структуры фиксированной части сообщения для любых методов любых интерфейсов, реализации которых включены во все вложенные компоненты.

В нашем примере будет сгенерировано два таких типа: `Server_entity_req` (для запроса) и `Server_entity_res` (для ответа).

```
union Server_entity_req {
    struct nk_message base_;
    union Ping_req pingComp_pingImpl;
};
```

```
union Server_entity_res {
    struct nk_message base_;
    union Ping_res pingComp_pingImpl;
};
```

- **Dispatch-методы (диспетчеры) для отдельного интерфейса, компонента или сущности.**

Диспетчеры анализируют полученный запрос (значения [RIID и MID](#)), вызывают реализацию соответствующего метода, после чего сохраняют ответ в буфер. В нашем примере будут сгенерированы диспетчеры `Ping_dispatch`, `Ping_component_dispatch` и `Server_entity_dispatch`.

Диспетчер сущности обрабатывает запрос и вызывает методы, реализуемые этой сущностью. Если запрос содержит некорректный RIID (например, относящийся к другой реализации интерфейса, которой нет у этой сущности) или некорректный MID, диспетчер возвращает `NK_EOK` или `NK_ENOENT`.

```
nk_err_t Server_entity_dispatch(struct Server_entity *,
                               const union Server_entity_req *,
                               const struct nk_arena *,
                               union Server_entity_res *,
                               struct nk_arena *);
```

В специальных случаях можно использовать диспетчеры интерфейса и компонента. Они принимают дополнительный аргумент – ID реализации интерфейса (`nk_iid_t`). Запрос будет обработан только если переданный аргумент и [RIID из запроса](#) совпадают, а MID корректен. В противном случае диспетчеры возвращают `NK_EOK` или `NK_ENOENT`.

```
nk_err_t Ping_dispatch(struct Ping *,
                      nk_iid_t,
                      const union Ping_req *,
                      const struct nk_arena *,
                      union Ping_res *,
                      struct nk_arena *);

nk_err_t Ping_component_dispatch(struct Ping_component *,
                                nk_iid_t,
                                const union Ping_component_req *,
                                const struct nk_arena *,
                                union Ping_component_res *,
                                struct nk_arena *);
```

## Классы политик безопасности

Политика безопасности или просто политика – это метод, определяющий правило проверки допустимости события.

Класс политик безопасности – набор семантически связанных политик, описывающих определенную модель безопасности.

Подробнее о политиках и их использовании см. ["Часть 3. Политика безопасности решения"](#).

## Класс политик Base

Класс Base предоставляет базовые политики безопасности.

PSL-описание класса Base находятся в следующем файле:

```
/opt/KasperskyOS-Community-Edition-<version>/toolchain/include/nk/base.psl
```

### Объект класса Base

Объект класса Base автоматически создается при подключении PSL-описания класса.

Необходимость создавать дополнительные объекты класса Base может возникнуть для объявления профилей аудита, связанных с этими объектами.

Объекты класса base не имеют конфигурации.

### Политики класса Base

Класс политик Base состоит из следующих политик-правил:

- `grant ()`

Не принимает параметров. Возвращает решение "разрешено".

Пример: `request { grant () }`

- `assert (Boolean)`

Возвращает решение "разрешено", если значение переданного выражения истинно. Иначе возвращает "запрещено".

Пример: `request { assert (message.port > 80) }`

- `deny ()`

Не принимает параметров. Возвращает решение "запрещено".

Пример: `request { deny () }`

### Пример

```
...
use nk.base._

/* Сущность foo, которой разрешено получать
   сообщения, но запрещено на них отвечать */
request src=foo { grant () }
response src=foo { deny () }
```



## Класс политик Regex

Класс политик `Regex` позволяет реализовать валидацию текстовых данных по статическим шаблонам (регулярным выражениям).

PSL-описание класса `Regex` находятся в следующем файле:

```
/opt/KasperskyOS-Community-Edition-<version>/toolchain/include/nk/regex.psl
```

### Объект класса Regex

Объект класса `Regex` автоматически создается при подключении PSL-описания класса.

Необходимость создавать дополнительные объекты класса `Regex` может возникнуть для объявления профилей аудита, связанных с этими объектами.

Объекты класса `Regex` не имеют конфигурации.

### Политики класса Regex

Класс политик `Regex` состоит из следующих политик-выражений:

- `match {text: Text, pattern: Pattern}`

Принимает строку `Text` и регулярное выражение `Pattern`. Возвращает значение типа `Boolean`: `True` при совпадении, иначе `False`.

Пример: `assert (re.match {text: message.text, pattern: "^[0-9]*$"})`

- `select {text: Text}`

Принимает строку. Предназначена для использования с оператором `choice` для проверки совпадений с несколькими регулярными выражениями.

Пример:

```
choice (re.select {text: "hello world"}) {  
  "^hello .*": grant ()  
  ".*world$" : grant ()  
  -           : deny ()  
}
```

## Класс политик HashSet

Класс политик `HashSet` реализует работу со структурой данных типа хеш-таблица. Позволяет ассоциировать с пользовательским ресурсом массив уникальных значений, добавлять и удалять элементы в этот массив, а также проверять, входит ли указанное значение в массив, ассоциированный с ресурсом.

PSL-описание класса `HashSet` находится в следующем файле:

## Объект класса HashSet

Объект класса `HashSet` представляет собой множество массивов – пул. В конфигурации объекта указывается тип элементов массивов, размер пула и размер массива. Предоставляются политики для:

- выделения массива из пула и связывания его с ресурсом, а также и для разрыва этой связи и возвращения массива в пул;
- добавления элемента в массив и удаления элемента из массива;
- проверки, является ли элемент членом массива, связанного с ресурсом.

## Конфигурация объекта

Конфигурация объекта класса `HashSet` содержит следующие элементы:

- `type Entry` – тип элемента массива. Поддерживаются целочисленные типы, а также `Boolean`. Также поддерживаются составные типы.
- `config`
  - `set_size` – размер массива.
  - `pool_size` – размер пула массивов.

Все параметры конфигурации обязательны при создании объекта класса.

## Конфигурация аудита

Объект класса `HashSet` не содержит дополнительных полей конфигурации аудита.

## Пример

```
security.psl

...
use nk.hashmap._

policy object S : HashSet {
    type Entry = UInt32 // элементы массива являются целочисленными

    config =
        { set_size: 5
          , pool_size: 2
        }
}
```

## Политика init класса HashSet

Выделяет из пула свободный массив и ассоциирует его с указанным ресурсом.

В рамках объекта класса `HashSet` с одним ресурсом может быть ассоциирован только один массив.

Тип: политика-правило.

### Синтаксис

```
init {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым будет ассоциирован массив.

### Возвращаемое решение

"Разрешено", если массив выделен.

"Запрещено" в следующих случаях:

- в пуле нет свободных массивов;
- указанный дескриптор ресурса уже ассоциирован с массивом в рамках объекта класса `HashSet`;
- указанный дескриптор ресурса некорректен.

### Пример

```
...
/* При запуске сущности Server с её дескриптором ассоциируется массив из пула объекта
S */
execute dst=Server {
    S.init {sid: dst_sid}
}
```

## Политика fni класса HashSet

Разрывает ассоциацию указанного ресурса с массивом и возвращает массив в пул.

Тип: политика-правило.

### Синтаксис

```
fini {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован массив.

## Возвращаемое решение

"Разрешено", если ассоциация разорвана и массив возвращен в пул.

"Запрещено" в следующих случаях:

- указанный дескриптор ресурса не ассоциирован с массивом в рамках объекта класса `HashSet`;
- указанный дескриптор ресурса некорректен.

## Политика add класса HashSet

Добавляет указанное значение в массив, ассоциированный с ресурсом.

Тип: политика-правило.

## Синтаксис

```
add {sid: Handle, entry: Entry}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован массив.
- `Entry` – значение, которое необходимо добавить в массив.

## Возвращаемое решение

"Разрешено", если значение добавлено или если в массиве уже содержится такое значение.

"Запрещено" в следующих случаях:

- массив, ассоциированный с указанным дескриптором ресурса, заполнен;
- указанный дескриптор ресурса не ассоциирован с массивом в рамках объекта класса `HashSet`;
- указанный дескриптор ресурса некорректен.

## Пример

```
...
/* При обращении сущности Server по интерфейсу безопасности в ассоциированный с ней
массив добавляется значение 5 */
security src=Server, method=Add {
  S.add {sid: src_sid, entry: 5}
}
```

## Политика remove класса HashSet

Удаляет указанное значение из массива, ассоциированного с ресурсом.

Тип: политика-правило.

### Синтаксис

```
remove {sid: Handle, entry: Entry}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован массив.
- `Entry` – значение, которое необходимо удалить из массива.

### Возвращаемое решение

"Разрешено", если значение удалено или если в массиве нет такого значения.

"Запрещено" в следующих случаях:

- указанный дескриптор ресурса не ассоциирован с массивом в рамках объекта класса `HashSet`;
- указанный дескриптор ресурса некорректен.

### Пример

```
...
/* При обращении сущности Server по интерфейсу безопасности из ассоциированного с ней
массива удаляется значение 5 */
security src=Server, method=Remove {
  S.remove {sid: src_sid, entry: 5}
}
```

## Политика contains класса HashSet

Проверяет, содержится ли указанное значение в массиве, ассоциированном с ресурсом.

Тип: политика-выражение.

## Синтаксис

```
contains {sid: Handle, entry: Entry}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован массив.
- `Entry` – значение, которое нужно проверить.

## Возвращаемое значение

`True`, если значение содержится в массиве.

`False`, если значение не содержится в массиве.

Решение "запрещено" в следующих случаях:

- указанный дескриптор ресурса не ассоциирован с массивом в рамках объекта класса `HashSet`;
- указанный дескриптор ресурса некорректен.

## Пример

```
...
/* При обращении сущности Server по интерфейсу безопасности проверяется,
содержится ли в ассоциированном с ней массиве значение 42 */
security src=Server, method=Check {
    assert(S.contains {sid: src_sid, entry: 42})
}
```

## Класс политик StaticMap

Класс политик `StaticMap` реализует работу со структурой данных типа "словарь со статическими ключами". Позволяет ассоциировать с ресурсом словарь элементов типа ключ-значение, а затем вычитывать и изменять значения элементов словаря.

Для каждого словаря существуют две копии: *актуальная* и *рабочая*. При инициализации словаря эти копии содержат одинаковые наборы элементов. Класс `StaticMap` предоставляет метод для записи в словарь нового значения по указанному ключу, при этом запись производится рабочей копии. Также предоставляются методы для копирования содержимого рабочей копии в актуальную и для копирования содержимого актуальной копии в рабочую. Таким образом есть возможность получить транзакционность поведения. Предоставляются методы для чтения элементов как актуальной, так и рабочей копий словаря.

PSL-описание класса `StaticMap` находится в следующем файле:

```
/opt/KasperskyOS-Community-Edition-<version>/toolchain/include/nk/staticmap.psl
```

## Объект класса `StaticMap`

Объект класса `StaticMap` представляет собой множество словарей – пул. Элементы словарей имеют тип ключ-значение. В конфигурации объекта указывается размер пула, тип значений, множество ключей и значения по-умолчанию для каждого ключа. Предоставляются политики для:

- выделения словаря из пула и связывания его с ресурсом, а также для разрыва этой связи и возвращения словаря в пул;
- чтения элементов словаря и изменения значений элементов;
- копирования содержимого рабочей копии словаря в актуальную и копирования содержимого актуальной копии в рабочую.

## Конфигурация объекта

Конфигурация объекта класса `StaticMap` содержит следующие элементы:

- `type Value` – тип значений элементов словаря. Поддерживаются только целочисленные типы.
- `config`
  - `keys` – словарь элементов типа ключ-значение со значениями по умолчанию.
  - `pool_size` – размер пула словарей.

Все параметры конфигурации обязательны при создании объекта класса.

## Конфигурация аудита

Объект класса `StaticMap` не содержит дополнительных полей конфигурации аудита.

## Пример

```
security.psl
```

```
...
use nk.staticmap._

policy object M : StaticMap {
    type Value = UInt16

    config =
        { keys:
            { "k1": 0
              , "k2": 1
```

```

    }
    , pool_size: 2
  }
}

```

## Политика init класса StaticMap

Выделяет из пула свободный словарь и ассоциирует его с указанным ресурсом.

Политика `init` устанавливает значения элементов как актуальной, так и рабочей копии в значения по умолчанию.

В рамках объекта класса `StaticMap` с одним ресурсом может быть ассоциирован только один словарь.

Тип: политика-правило.

## Синтаксис

```
init {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым будет ассоциирован словарь.

## Возвращаемое решение

"Разрешено", если словарь выделен.

"Запрещено" в следующих случаях:

- в пуле нет свободных словарей;
- указанный дескриптор ресурса уже ассоциирован с массивом в рамках объекта класса `StaticMap`;
- указанный дескриптор ресурса некорректен.

## Пример

```

...
/* При запуске сущности Server с её дескриптором ассоциируется словарь из пула объекта
M */
execute dst=Server {
    M.init {sid: dst_sid}
}

```



## Политика fini класса StaticMap

Разрывает ассоциацию указанного ресурса со словарем и возвращает словарь в пул.

Тип: политика-правило.

Синтаксис

```
fini {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован словарь.

### Возвращаемое решение

"Разрешено", если ассоциация разорвана и словарь возвращен в пул.

"Запрещено" в следующих случаях:

- указанный дескриптор ресурса не ассоциирован со словарем в рамках объекта класса `StaticMap`;
- указанный дескриптор ресурса некорректен.

## Политика set класса StaticMap

Присваивает указанное значение указанному ключу в *рабочей* копии словаря, ассоциированного с ресурсом.

Тип: политика-правило.

Синтаксис

```
set {sid: Handle, key: Key, value: Value}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован словарь.
- `Key` – ключ, которому необходимо присвоить значение.
- `Value` – значение, которое необходимо присвоить ключу.

### Возвращаемое решение

"Разрешено", если значение присвоено.

"Запрещено" в следующих случаях:

- указанный ключ не содержится в словаре, ассоциированном с ресурсом;
- указанный дескриптор ресурса не ассоциирован со словарем в рамках объекта класса `StaticMap`;
- указанный дескриптор ресурса некорректен.

## Пример

```
...  
/* При обращении сущности Server по интерфейсу безопасности, в ассоциированном с  
дескриптором  
этой сущности словаре ключу k1 присваивается значение 2 */  
security src=Server, method=Set {  
  M.set {sid: src_sid, key: "k1", value: 2 }  
}
```

## Политика commit класса StaticMap

В словаре, ассоциированном с указанным ресурсом, присваивает всем ключам *актуальной* копии соответствующие значения ключей *рабочей* копии.

Эту политику можно использовать как фиксацию текущей и начало новой транзакции.

Тип: политика-правило.

## Синтаксис

```
commit {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован словарь.

## Возвращаемое решение

"Разрешено", если значения ключей успешно присвоены.

"Запрещено" в следующих случаях:

- указанный дескриптор ресурса не ассоциирован со словарем в рамках объекта класса `StaticMap`;
- указанный дескриптор ресурса некорректен.

## Пример

```
...
/* При обращении сущности Server по интерфейсу безопасности в ассоциированном с ней
словаре
всем ключам актуальной копии присваиваются значения соответствующих ключей рабочей
копии */
security src=Server, method=Set {
    M.commit {sid: src_sid}
}
```

## Политика rollback класса StaticMap

В словаре, ассоциированном с указанным ресурсом, присваивает всем ключам *рабочей* копии соответствующие значения ключей *актуальной* копии.

Эту политику можно рассматривать как откат текущей транзакции.

Тип: политика-правило.

## Синтаксис

```
rollback {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован словарь.

## Возвращаемое решение

"Разрешено", если значения ключей успешно присвоены.

"Запрещено" в следующих случаях:

- указанный дескриптор ресурса не ассоциирован со словарем в рамках объекта класса `StaticMap`;
- указанный дескриптор ресурса некорректен.

## Пример

```
...
/* При обращении сущности Server по интерфейсу безопасности в ассоциированном с ней
словаре
всем ключам рабочей копии присваиваются значения соответствующих ключей актуальной
копии */
security src=Server, method=Set {
```

```
M.rollback {sid: src_sid}
}
```

## Политика get класса StaticMap

Возвращает текущее значение указанного ключа в *актуальной* копии словаря, ассоциированного с ресурсом.

Тип: политика-выражение.

### Синтаксис

```
get {sid: Handle, key: Key}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован словарь.
- `Key` – ключ, значение которого необходимо нужно получить.

### Возвращаемое значение

Политика возвращает значение указанного ключа или решение "запрещено" в следующих случаях:

- указанный ключ не содержится в словаре, ассоциированном с ресурсом;
- указанный дескриптор ресурса не ассоциирован со словарем в рамках объекта класса `StaticMap`;
- указанный дескриптор ресурса некорректен.

### Пример

```
...
/* При обращении сущности Server по интерфейсу безопасности, значение ключа "k1" в
актуальной копии
ассоциированного с ней словаря сравнивается с числом 0 */
security src=Server, method=Get {
  assert(M.get {sid: src_sid, key: "k1" } != 0)
}
```

## Политика get\_uncommitted класса StaticMap

Возвращает текущее значение указанного ключа в *рабочей* копии словаря, ассоциированного с ресурсом.

Тип: политика-выражение.

## Синтаксис

```
get_uncommitted {sid: Handle, key: Key}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован словарь.
- `Key` – ключ, значение которого необходимо нужно получить.

## Возвращаемое значение

Политика возвращает значение указанного ключа или решение "запрещено" в следующих случаях:

- указанный ключ не содержится в словаре, ассоциированном с ресурсом.
- указанный дескриптор ресурса не ассоциирован со словарем в рамках объекта класса `StaticMap`;
- указанный дескриптор ресурса некорректен.

## Пример

```
...  
/* При обращении сущности Server по интерфейсу безопасности, значение ключа "k1" в  
рабочей копии  
ассоциированного с ней словаря сравнивается с числом 0 */  
security src=Server, method=GetUncommitted {  
  assert(M.get_uncommitted { sid: src_sid, key: "k1" } != 0)  
}
```

## Класс политик Flow

Класс политик `Flow` реализует модель безопасности "конечный автомат на уровне пользовательского ресурса". Позволяет определить множество внутренних состояний, задать правила перехода между ними и управлять текущим состоянием для каждого ресурса.

PSL-описание класса `Flow` находится в следующем файле:

```
/opt/KasperskyOS-Community-Edition-<version>/toolchain/include/nk/flow.psl
```

## Объект класса Flow

Объект класса `Flow` – реализация конечного автомата, описание которого задается в конфигурации объекта.

## Конфигурация объекта

Конфигурация объекта класса `Flow` содержит следующие элементы:

- `type States` – тип, определяющий множество допустимых внутренних состояний.

По умолчанию допускает любые строковые значения. При создании объекта класса, необходимо уточнить этот тип до списка допустимых строковых значений, объединенных символом `|` (OR).

- `config`
  - `states` – множество внутренних состояний.
  - `initial` – начальное состояние.
  - `transitions` – таблица переходов между состояниями. Для каждого текущего состояния указывается список состояний, в которые можно перейти.

Все параметры конфигурации обязательны при создании объекта класса.

## Конфигурация аудита

При объявлении профиля аудита, объект класса `Flow` содержит следующие поля конфигурации аудита:

```
{ <имя объекта>:
  { kss: [ "granted", "denied" ]
    , omit: [<states>] // список внутренних состояний объекта, в которых
    результаты вызовов политик не попадут в журнал аудита.
  }
```

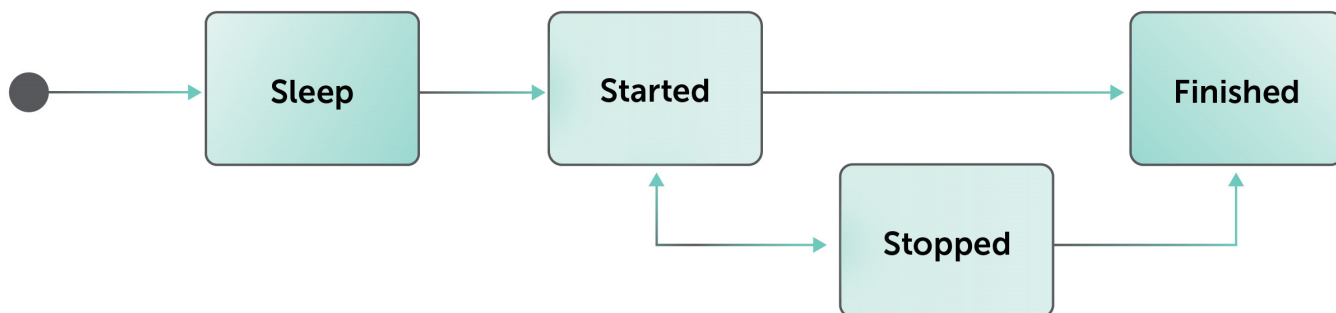
## Пример

security.psl

```
...
use nk.flow._

/* service_flow – пример реализации абстрактного сервиса
с конфигурацией из четырех состояний.
Дальнейшие примеры использования класса Flow основаны на этом примере. */
policy object service_flow : Flow {
  type States = "sleep" | "started" | "stopped" | "finished"
  config = {
    states      : ["sleep", "started", "stopped", "finished"],
    initial     : "sleep",
    transitions : {
      "sleep"   : ["started"],
      "started" : ["stopped", "finished"],
      "stopped" : ["started", "finished"]
    }
  }
```

```
}  
}
```



Пример конечного автомата, используемый в этом разделе

## Политика init класса Flow

Инициализирует конечный автомат для указанного ресурса.

В рамках объекта класса с одним ресурсом может быть ассоциирован только один конечный автомат.

Тип: политика-правило.

### Синтаксис

```
init {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым будет ассоциирован конечный автомат.

### Возвращаемое решение

"Разрешено", если конечный автомат инициализирован.

"Запрещено" в следующих случаях:

- указанный дескриптор ресурса уже ассоциирован с конечным автоматом в рамках объекта класса `Flow`;
- указанный дескриптор ресурса некорректен.

#### Пример

```
...  
/* При запуске сущности Server инициализируется конечный автомат, ассоциированный с  
её дескриптором */  
execute dst=Server {  
    service_flow.init {sid: dst_sid}  
}
```

## Политика `fini` класса `Flow`

Финализирует конечный автомат для указанного ресурса.

Тип: политика-правило.

Синтаксис

```
fini {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован конечный автомат.

### Возвращаемое решение

"Разрешено", если конечный автомат финализирован.

"Запрещено" в следующих случаях:

- указанный дескриптор ресурса не ассоциирован с конечным автоматом в рамках объекта класса `Flow`;
- указанный дескриптор ресурса некорректен.

## Политика `query` класса `Flow`

Возвращает текущее состояние объекта класса `Flow`.

Тип: политика-выражение.

Синтаксис

```
query {sid: Handle}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован конечный автомат.

### Возвращаемое значение

Текущее состояние объекта класса `Flow`.



## Пример

```
...
/* При отправке запроса сущностью "ResourceDriver" будет проверено состояние объекта
"service_flow".
Если "service_flow" находится в состоянии "started" или "stopped", то отправка запроса
разрешена,
иначе – запрещена. */
request src=ResourceDriver {
    choice (service_flow.query {sid: src_sid}) {
        "started"    : grant ()
        "stopped"    : grant ()
        –            : deny  ()
    }
}
```

## Политика enter класса Flow

Переводит объект класса `Flow` в указанное состояние, если такой переход разрешен в конфигурации объекта класса.

Тип: политика-правило.

### Синтаксис

```
enter {sid: Handle, state: State}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован конечный автомат.
- `State` – состояние, в которое необходимо перейти.

### Возвращаемое решение

"Разрешено", если переход в указанное состояние осуществлен.

"Запрещено" в следующих случаях:

- переход в указанное состояние не осуществлен;
- для указанного дескриптора ресурса не инициализирован конечный автомат;
- указанный дескриптор ресурса некорректен.

## Пример

```
...
/* При отправке запроса сущности Server, перевести ассоциированный с ней конечный
автомат в состояние "started" */
request dst=Server {
    service_flow.enter {sid: dst_sid, state: "started"}
}
```

## Политика allow класса Flow

Разрешает событие, если текущее состояние входит в переданный список состояний.

Тип: политика-правило.

### Синтаксис

```
allow {sid: Handle, states: [States]}
```

Здесь:

- `Handle` – дескриптор ресурса, с которым ассоциирован конечный автомат.
- `States` – список состояний для проверки.

### Возвращаемое решение

"Разрешено", если текущее состояние входит в переданный список состояний.

"Запрещено", если:

- текущее состояние не входит в переданный список состояний;
- для указанного дескриптора ресурса не инициализирован конечный автомат;
- указанный дескриптор ресурса некорректен.

### Пример

```
...
/* Отправка запроса сущности Server разрешена только если конечный автомат,
ассоциированный с ее дескриптором находится в состоянии "started" */
request dst=Server {
    service_flow.allow {sid: dst_sid, states: ["started"]}
}
```

## Класс политик Rbac

Класс политик Rbac позволяет реализовать модель управления доступом на основе ролей (Role Based Access Control, далее также RBAC). Описание модели RBAC задается в конфигурации объекта класса Rbac.

Декларация класса rbac находится в следующем файле:

```
/opt/KasperskyOS-Community-Edition-<version>/toolchain/include/nk/rbac.ps1
```

## Основные понятия RBAC

### Типы

*Тип* – идентификатор, являющийся характеристикой ресурса с точки зрения контроля доступа, который может быть ассоциирован с ресурсом с помощью политик класса Rbac.

Тип может быть ассоциирован как с субъектом (например, сущностью), так и с объектом действия (например, файлом).

Список типов задается статически в конфигурации объекта класса.

### Полномочия

*Полномочие* – идентификатор некоего действия или действий.

Список полномочий задается статически в конфигурации объекта класса.

### Роли

*Роль* – это матрица прав, которая описывает, какие полномочия имеют субъекты при обращении к объектам. Другими словами, роль определяет бинарное отношение на множестве типов.

Список ролей задается статически в конфигурации объекта класса.

### Ограничения/правила для операций

Класс политик Rbac предоставляет политики для создания субъектов и объектов, а также для изменения типа объекта и добавления новых ролей к субъекту.

Каждая из этих операций имеет ограничения и правила, статически заданные в конфигурации объекта класса.

## Объект класса Rbac

Объект класса `Rbac` описывает модель управления доступом на основе типов, полномочий, ролей, и ограничений/правил для операций. Эти параметры задаются в конфигурации объекта класса `Rbac`.

## Конфигурация объекта

Конфигурация объекта класса `Rbac` содержит следующие элементы:

- `types` – список допустимых моделью типов объектов и субъектов.  
Пример: `types: [core, einit, tls, smm, auth, app]`
- `images` – список образов, из которых создаются субъекты.
- `permissions` – список допустимых моделью полномочий (см. [Полномочия](#)).
- `roles` – список допустимых моделью ролей (см. [Роли](#)).
- `create_subject` – ограничения и правила для операции создания субъекта (см. [Ограничения/правила: создание субъекта](#)).
- `create_object` – ограничения и правила для операции создания объекта (см. [Ограничения/правила: создание объекта](#)).
- `retype_object` – ограничения и правила для операции изменения типа объекта (см. [Ограничения/правила: изменение типа объекта](#)).
- `add_roles` – ограничения и правила для операции добавления ролей к субъекту (см. [Ограничения/правила: добавление ролей к субъекту](#)).

Все параметры конфигурации обязательны при создании объекта класса.

## Конфигурация аудита

Объект класса `Rbac` не содержит дополнительных полей конфигурации аудита.

## Пример

```
security.psl

use nk.rbac._

policy object rbac0 : Rbac {
  config =
    { types: ["actors", "userland", "obscura"]
    , images: []
    , permissions:
      { mundane: ["observe"]
      , unusual: ["research"]
      }
    , roles:
      { admin:
        { extends: []
        , rights: []
        }
      }
    }
```

```

, user:
  { extends: []
  , rights:
    [ { from: ["actors"]
      , to: ["userland"]
      , permissions: ["mundane/observe"]
      }
    ]
  }
, obscurantist:
  { extends: []
  , rights:
    [ { from: ["actors"]
      , to: ["obscura"]
      , permissions: ["mundane/observe"]
      }
      { from: ["actors"]
      , to: ["obscura"]
      , permissions: ["unusual/research"]
      }
    ]
  }
, hyperadmin:
  { extends: ["user", "admin", "obscurantist"]
  , rights: []
  }
}
, create_subject:
[ { source_type: ["actors"]
  , source_role: ["admin"]
  , target_type: ["actors"]
  // it might be nice to have @any target_role selector
  , target_role: ["user", "obscurantist"]
  , target_type_auto: ()
  , target_role_auto: []
  , image: []
  }
, { source_type:["@any"]
  , source_role:["@any"]
  , target_type: ["actors"]
  , target_role: ["admin"]
  , target_type_auto: ()
  , target_role_auto: []
  , image: []
  }
]
, create_object:
[ { source_type: ["actors"]
  , source_role: ["user"]
  , target_type: ["userland"]
  , target_type_auto: ()
  , container_type: ["@any"]
  }
, { source_type: ["actors"]
  , source_role: ["obscurantist"]
  , target_type: ["obscura"]
  , target_type_auto: ()
  , container_type: ["@any"]
  }
]

```

```

, retype_object:
[ { source_type: ["actors"]
  , source_role: ["admin"]
  , target_type: ["userland", "obscura"]
  , original_type: ["userland", "obscura"]
  , container_type: ["@any"]
  }
]
, add_roles:
[ { source_type: ["actors"]
  , source_role: ["admin"]
  , target_type: ["@source_type"]
  , target_role: ["admin", "user", "obscurantist"]
  }
]
}
}

```

## Полномочия

Список полномочий имеет двухуровневую структуру, которая позволяет группировать полномочия в категории полномочий.

Пример:

```

permissions: {
  handle: [alloc, free, init, fini, allocIpcHandle],
  tls: [handshake, describe],
  stream: [read, write, close, get_status, set_status],
  conn: [connect, accept, shutdown]
}

```

## Роли

Существует два способа описать роль, которые могут использоваться одновременно:

- расширение (extension) одной или нескольких ролей;
- явное задание матрицы прав.

Результирующая матрица прав для роли определяется как объединение множеств полномочий в соответствующих ячейках матриц расширяемых ролей и явно заданной матрицы.

## Расширение ролей

Расширение ролей выполняется с помощью элемента `extends`, который содержит список расширяемых ролей (список может быть пустым). Все расширяемые роли должны быть ранее определены:

```

roles: {
  ...
  roleX : {
    extends : [roleY, roleZ] // Роль roleX расширяет роли roleY и roleZ
  }
  ...
}

```

## Явное задание матрицы прав

Явное задание матрицы выполняется с помощью элемента `rights`, который представляет собой массив троек: субъект (элемент `from`), объект, к которому требуется доступ (элемент `to`) и множество полномочий (элемент `permissions`).

Субъект и объект задаются одним из следующих способов:

- конкретный тип: `from: tls`
- список типов: `from: [app0, app1]`
- любой тип: `from: @any`

Полномочия задаются одним из следующих способов:

- конкретное полномочие: `permissions: stream/read`
- конкретная категория полномочий: `permissions: stream/@any`
- список конкретных полномочий и/или категорий: `permissions: [conn/@any, stream/set_status]`

Пример:

```

roles : {
  system : {
    rights :[
      { from : @any /* любой mun */
      , to : connection_manager /* конкретный mun */
      , permissions : conn/@any /* любое полномочие из категории conn */
      },
      { from : tls
      , to : connection_manager
      , permissions : [ stream/read, stream/set_status ] /* любое из перечисленных полномочий */
      },
      { from : @any
      , to : tls
      , permissions : stream/@any
      }
    ]
  },
  camera : {
    rights : [
      { from : @any
      , to : camera

```

```

        , permissions : camera/@any
      }
    ]
  },
  gps : {
    rights : [
      { from : @any
        , to : gps
        , permissions : gps/@any
      }
    ]
  },
  user : {
    extends : system /* расширяет роль system */
  },
  admin : {
    extends : [user, camera] /* расширяет роли user & camera */
  }
}

```

## Ограничения/правила: создание субъекта

При создании субъекта применяются правила, связывающие тип и роль запускающего (родительского) субъекта и образ (image) запускаемого субъекта с типом и множеством ролей, которые назначаются новому субъекту.

## Элементы source\_type, source\_role и image

Элемент `create_subject` может содержать несколько правил. Правила применяются последовательно (порядок важен) до тех пор, пока не будет найдено первое, непротиворечащее комбинации значений `source_type`, `source_role` и `image`. Другими словами, для применения правила необходимо, чтобы:

- тип субъекта совпал с одним из указанных в `source_type`.
- одна из ролей субъекта совпала с одной из указанных в `source_role`.
- имя образа, из которого создается новый субъект, совпало с одним из указанных в `image`.

Варианты задания типа субъекта, инициирующего создание нового субъекта (элемент `source_type`):

- конкретный тип: `source_type: core`
- список типов: `source_type: [core, dispatcher]`
- любой тип: `source_type: @any`

Варианты задания роли субъекта, инициирующего создание нового субъекта (элемент `source_role`):

- конкретная роль: `source_role: system`
- список ролей: `source_role : [system, user]`
- любая роль: `source_role: @any`



Варианты задания имени образа, из которого создается новый субъект (элемент `image`):

- конкретное имя образа: `image: core`
- список имен образов: `image: [core, einit]`
- любой образ: `image: @any`

## Элементы `target_type` и `target_type_auto`

Элементы `target_type` и `target_type_auto` используются для определения допустимых типов для создаваемого субъекта.

При этом `target_type` используется в случае, когда желаемый тип указывается при создании субъекта, а `target_type_auto` используется, когда желаемый тип не указан, и политика должна сама вычислить допустимый тип.

Варианты задания допустимых типов в случае явного указания типа (элемент `target_type`):

- конкретный тип: `target_type: core`
- список типов: `target_type: [core, dispatcher]`
- тип, совпадающий с типом сущности-инициатора: `target_type: @source_type`
- любой тип: `target_type: @any`

Назначение желаемого типа новому субъекту будет разрешено, если этот тип совпадет с одним из указанных в `target_type`. Если элемент `target_type` не задан, то явное назначение типа запрещено.

Варианты задания автоматически назначаемого типа в случае отсутствия явного указания типа (элемент `target_type_auto`):

- конкретный тип: `target_type_auto: core`
- тип, совпадающий с типом сущности-инициатора: `target_type_auto: @source_type`

Создаваемому субъекту будет назначен тип, указанный в `target_type_auto`. Если поле `target_type_auto` не задано, то неявное назначение типа запрещено.

## Элементы `target_role` и `target_role_auto`

Элементы `target_role` и `target_role_auto` используются для определения допустимых ролей для создаваемого субъекта.

При этом `target_role` используется в случае, когда желаемые роли указываются при создании субъекта, а `target_role_auto` используется, когда желаемые роли не указаны, и политика должна сама вычислить нужные роли.

Варианты задания допустимых ролей в случае явного указания ролей (элемент `target_role`):

- конкретная роль: `target_role: core`

- список ролей: `target_role: [core, dispatcher]`
- роли, совпадающие с ролями сущности-инициатора: `target_role: @source_role`
- любая роль: `target_role: @any`

Назначение желаемых ролей новому субъекту будет разрешено, если эти роли являются подмножеством ролей, указанных в `target_role`. Если элемент `target_role` не задан, то явное назначение ролей запрещено.

Варианты задания автоматически назначаемых ролей в случае отсутствия явного указания ролей (элемент `target_role_auto`):

- конкретная роль: `target_role_auto: core`
- список ролей: `target_role_auto: [core, dispatcher]`
- роли, совпадающие с ролями сущности-инициатора: `target_role_auto: @source_roles`
- любая роль: `target_role_auto: @any`

Создаваемому субъекту будет назначен тип, указанный в `target_role_auto`. Если элемент `target_role_auto` не задан, то неявное назначение типа запрещено.

## Пример

```
create_subject: {
  { source_type: core // субъект инициатор должен иметь тип "core"
    , source_role: system // субъект инициатор должен иметь роль "system"
    , image: einit // образ
    , target_type_auto: einit // автоматическое назначение типа возможно
    , target_role_auto: @source_roles // автоматическое назначение ролей: роли,
    совпадающие с ролями сущности-инициатора
  },
  { source_type: dispatcher
    , source_role: [system, user]
    , image: rpcservice
    , target_type: [user, admin] // тип создаваемого субъекта должен быть указан, и
    должен быть одним из списка
    , target_role_auto : @source_roles
  },
  { source: @any
    , image: tls
    , target_type: [@source_type, tls]
    , target_role: [user, admin]
  },
}
```

## Ограничения/правила: создание объекта

При создании объекта применяются правила, связывающие тип и роль создающего субъекта и тип родительского контейнера с типом который назначаются новому субъекту.

## Элементы `source_type`, `source_role` и `container_type`

Элемент `create_object` может содержать несколько правил. Правила применяются последовательно (порядок важен) до тех пор, пока не будет найдено первое, непротиворечащее комбинации значений `source_type`, `source_role` и `container`. Другими словами, для применения правила необходимо, чтобы:

- тип субъекта совпал с одним из указанных в `source_type`.
- одна из ролей субъекта совпала с одной из указанных в `source_role`.
- тип контейнера, в котором создается объект, совпал с одним из указанных в `container_type`.

Варианты задания типа субъекта, инициирующего создание нового объекта (элемент `source_type`):

- конкретный тип: `source_type: core`
- список типов: `source_type: [core, dispatcher]`
- любой тип: `source_type: @any`

Варианты задания роли субъекта, инициирующего создание нового объекта (элемент `source_role`):

- конкретная роль: `source_role: system`
- список ролей: `source_role : [system, user]`
- любая роль: `source_role: @any`

Варианты задания типа контейнера, в котором создается новый объект (элемент `container_type`):

- конкретный тип: `container_type: core`
- тип, совпадающий с типом сущности-инициатора: `container_type : @source_type`
- список типов: `container_type: [core, dispatcher, @source_type]`
- любой тип: `container_type : @any`

## Элементы `target_type` и `target_type_auto`

Элементы `target_type` и `target_type_auto` используются для определения допустимых типов для создаваемого объекта.

При этом `target_type` используется в случае, когда желаемый тип указывается при создании объекта, а `target_type_auto` используется, когда желаемый тип не указан, и политика должна сама вычислить допустимый тип.

Варианты задания допустимых типов в случае явного указания типа (элемент `target_type`):

- конкретный тип: `target_type: core`
- тип, совпадающий с типом сущности-инициатора: `target_type: @source_type`
- тип, совпадающий с типом объекта-контейнера: `target_type: @container_type`

- список типов: `target_type: [core, dispatcher, @container_type]`
- любой тип: `target_type: @any`

Назначение желаемого типа новому субъекту будет разрешено, если этот тип совпадет с одним из указанных в `target_type`. Если элемент `target_type` не задан, то явное назначение типа запрещено.

Варианты задания автоматически назначаемого типа в случае отсутствия явного указания типа (элемент `target_type_auto`):

- конкретный тип: `target_type_auto: core`
- тип, совпадающий с типом сущности-инициатора: `target_type_auto: @source_type`
- тип, совпадающий с типом объекта-контейнера: `target_type_auto: @container_type`

Создаваемому субъекту будет назначен тип, указанный в `target_type_auto`. Если поле `target_type_auto` не задано, то неявное назначение типа запрещено.

## Элементы `target_role` и `target_role_auto`

Элементы `target_role` и `target_role_auto` используются для определения допустимых ролей для создаваемого субъекта.

При этом `target_role` используется в случае, когда желаемые роли указываются при создании субъекта, а `target_role_auto` используется, когда желаемые роли не указаны, и политика должна сама вычислить нужные роли.

Варианты задания допустимых ролей в случае явного указания ролей (элемент `target_role`):

- конкретная роль: `target_role: core`
- список ролей: `target_role: [core, dispatcher]`
- роли, совпадающие с ролями сущности-инициатора: `target_role: @source_role`
- любая роль: `target_role: @any`

Назначение желаемых ролей новому субъекту будет разрешено, если эти роли являются подмножеством ролей, указанных в `target_role`. Если элемент `target_role` не задан, то явное назначение ролей запрещено.

Варианты задания автоматически назначаемых ролей в случае отсутствия явного указания ролей (элемент `target_role_auto`):

- конкретная роль: `target_role_auto: core`
- список ролей: `target_role_auto: [core, dispatcher]`
- роли, совпадающие с ролями сущности-инициатора: `target_role_auto: @source_roles`
- любая роль: `target_role_auto: @any`

Создаваемому субъекту будет назначен тип, указанный в `target_role_auto`. Если элемент `target_role_auto` не задан, то неявное назначение типа запрещено.

## Пример

```
create_object: {
  { source_type: realm
    , source_role: system
    , container_type: app_file
    , target_type_auto: @container_type
  },
  { source_type: realm
    , source_role: system
    , container_type: @source_type
    , target_type: [app_file, secure_file]
  },
}
```

## Ограничения/правила: изменение типа объекта

При изменении типа объекта применяются правила, связывающие тип и роль субъекта, инициирующего изменения, а также тип родительского контейнера и исходный тип объекта с целевым типом, который назначается субъекту.

Элемент `retype_object` может содержать несколько правил. Правила применяются последовательно (порядок важен) до тех пор, пока не будет найдено первое, непротиворечащее комбинации значений `source_type`, `source_role`, `container_type` и `original_type`. Другими словами, для применения правила необходимо, чтобы:

- тип субъекта совпал с одним из указанных в `source_type`.
- одна из ролей субъекта совпала с одной из указанных в `source_role`.
- тип контейнера, в котором содержится изменяемый объект, совпал с одним из указанных в `container_type`.
- исходный тип изменяемого объекта совпал одним из типов, указанных в `original_type`.

Варианты задания типа субъекта, инициирующего изменение типа объекта (элемент `source_type`):

- конкретный тип: `source_type: core`
- список типов: `source_type: [core, dispatcher]`
- любой тип: `source_type: @any`

Варианты задания роли субъекта, инициирующего изменение типа объекта (элемент `source_role`):

- конкретная роль: `source_role: system`
- список ролей: `source_role : [system, user]`
- любая роль: `source_role: @any`

Варианты задания типа контейнера, в котором находится изменяемый объект (элемент `container_type`):

- конкретный тип: `container_type: core`
- тип, совпадающий с типом сущности-инициатора: `container_type : @source_type`
- список типов: `container_type: [core, dispatcher, @source_type]`
- любой тип: `container_type: @any`

Варианты задания исходного типа целевого объекта (элемент `original_type`):

- конкретный тип: `original_type: core`
- тип, совпадающий с типом сущности-инициатора: `original_type : @source_type`
- тип, совпадающий с типом объекта-контейнера: `original_type: @container_type`
- исходный тип целевого объекта: `original_type: @original_type`
- список типов: `original_type: [core, dispatcher, @source_type]`
- любой тип: `original_type: @any`

Варианты задания допустимых типов для целевого объекта (элемент `target_type`):

- конкретный тип: `target_type: core`
- тип, совпадающий с типом сущности-инициатора: `target_type: @source_type`
- тип, совпадающий с типом объекта-контейнера: `target_type: @container_type`
- исходный тип целевого объекта: `target_type: @original_type`
- список типов: `target_type: [core, dispatcher, @container_type]`
- любой тип: `target_type: @any`

## Пример

```
retype_object : [
  { source_type: realm
    , source_role: system
    , container_type: secure_file
    , original_type: app_file
    , target_type: [app_file, secure_file]
  },
]
```

## Ограничения/правила: добавление ролей к субъекту

При добавлении роли к субъекту применяются правила, связывающие тип и роль субъекта, инициирующего изменения, с целевым типом субъекта, к которому добавляется роль.

Добавление ролей позволяет динамически (во время исполнения) расширять права субъектов.

Элемент `add_role` может содержать несколько правил. Правила применяются последовательно (порядок важен) до тех пор, пока не будет найдено первое, непротиворечащее комбинации значений `source_type`, `source_role`, и `target_type`. Другими словами, для применения правила необходимо, чтобы:

- тип субъекта-инициатора совпал с одним из указанных в `source_type`.
- одна из ролей субъекта-инициатора совпала с одной из указанных в `source_role`.
- тип изменяемого субъекта совпал одним из типов, указанных в `target_type`.

Варианты задания типа субъекта, инициирующего добавление ролей (элемент `source_type`):

- конкретный тип: `source_type: core`
- список типов: `source_type: [core, dispatcher]`
- любой тип: `source_type: @any`

Варианты задания роли субъекта, инициирующего добавление ролей (элемент `source_role`):

- конкретная роль: `source_role: system`
- список ролей: `source_role: [system, user]`
- любая роль: `source_role: @any`

Варианты задания исходного типа целевого субъекта (элемент `target_type`):

- конкретный тип: `original_type: core`
- тип, совпадающий с типом сущности-инициатора: `original_type : @source_type`
- список типов: `original_type: [core, dispatcher, @source_type]`
- любой тип: `original_type: @any`

Варианты задания допустимых ролей для целевого субъекта (элемент `target_role`):

- конкретная роль: `target_role: core`
- роли, совпадающие с ролями сущности-инициатора: `target_role: @source_role`
- список ролей: `target_role: [core, dispatcher]`
- \*любая роль: `target_role: @any`

## Пример

```
add_role : [  
  { source_type: dispatcher  
    , source_role: system
```

```
, target_type: [audit-service, file-service]
, target_role: [user, admin]
},
]
```

## Политика create\_subject класса Rbac

Создает новый субъект.

Тип: политика-правило.

### Синтаксис

```
create_subject:{source: SourceSid, target: TargetSid, image: ImageName, type: Type,
roles: Roles}
```

Здесь:

- **SourceSid** – дескриптор субъекта-инициатора.

Может быть пустым. В этом случае в соответствующем [правиле конфигурации](#) должно быть указано `source_type: @any` и `source_role: @any`.

- **TargetSid** – дескриптор создаваемого субъекта.

- **ImageName** – имя образа из которого создается новый субъект.

Может быть пустым. В этом случае в соответствующем правиле конфигурации должно быть указано `image : @any`.

- **Type** – желаемый тип создаваемого субъекта.

Может быть пустым. В этом случае в соответствующем правиле конфигурации должны быть указаны параметры для автоматического назначения типа.

- **Roles** – желаемые роли создаваемого субъекта.

Может быть пустым. В этом случае в соответствующем правиле конфигурации должны быть указаны параметры для автоматического назначения ролей.

### Возвращаемое решение

"Разрешено", если субъект создан.

"Запрещено", если субъект не создан.

### Пример

```
execute dst=SimpleServer {
```



```
rbac0.create_subject
{ source: ()
, target: src_sid
, image: ()
, type: "actors"
, roles: ["admin"]
}
}
```

## Политика create\_object класса Rbac

Создает новый объект.

Тип: политика-правило.

### Синтаксис

```
create_object: {source: SourceSid, target: TargetSid, container: ContainerType, type:
Type}
```

Здесь:

- **SourceSid** – дескриптор субъекта-инициатора.

Может быть пустым. В этом случае в соответствующем [правиле конфигурации](#) должно быть указано `source_type: @any` и `source_role: @any`.

- **TargetSid** – дескриптор создаваемого объекта.

- **ContainerType** – тип объекта-контейнера, внутри которого создается новый объект.

Может быть пустым. В этом случае в соответствующем правиле конфигурации должно быть указано `container_type: @any`.

- **Type** – желаемый тип создаваемого объекта.

Может быть пустым. В этом случае в соответствующем правиле конфигурации должны быть указаны параметры для автоматического назначения типа.

### Возвращаемое решение

"Разрешено", если объект создан.

"Запрещено", если объект не создан.

### Пример

```
execute dst=SimpleServer {
  rbac0.create_object
```

```
{ source:    src_sid
, target:    dst_sid
, container: ()
, type:      "userland"
}
```

## Политика check класса Rbac

Проверяет, имеет ли указанный субъект необходимые полномочия в отношении указанного объекта.

Тип: политика-правило.

### Синтаксис

```
check: {source: SourceSid, target: TargetSid, permissions: Permissions}
```

Здесь:

- `SourceSid` – дескриптор субъекта.
- `TargetSid` – дескриптор объекта.
- `Permissions` – список полномочий для проверки.

### Возвращаемое решение

"Разрешено", если указанный субъект имеет необходимые полномочия в отношении указанного объекта.

"Запрещено", если указанный субъект не имеет необходимых полномочий в отношении указанного объекта.

### Пример

```
request interface=test.idl.Ping, method=head0 {
  rbac0.check
  { source:    src_sid
  , target:    dst_sid
  , permissions: ["mundane/observe"]
  }
}
```

## Политика retype\_object класса Rbac

Изменяет тип объекта.

Тип: политика-правило.

## Синтаксис

```
retype_object: {source: SourceSid, target: TargetSid, container: ContainerType, type: Type}
```

Здесь:

- `SourceSid` – дескриптор субъекта-инициатора.  
Может быть пустым. В этом случае в соответствующем [правиле конфигурации](#) должно быть указано `source_type: @any` и `source_role: @any`.
- `TargetSid` – дескриптор целевого объекта.
- `ContainerType` – тип объекта-контейнера, содержащего целевой объект.  
Может быть пустым. В этом случае в соответствующем правиле конфигурации должно быть указано `container_type: @any`.
- `Type` – желаемый тип целевого объекта.

## Возвращаемое решение

"Разрешено", если тип объекта изменен.

"Запрещено", если тип объекта не изменен.

## Политика `add_roles` класса `Rbac`

Добавляет одну или несколько ролей к субъекту.

Тип: политика-правило.

## Синтаксис

```
add_roles: {source: SourceSid, target: TargetSid, roles: Roles}
```

Здесь:

- `SourceSid` – дескриптор субъекта-инициатора.  
Может быть пустым. В этом случае в соответствующем [правиле конфигурации](#) должно быть указано `source_type: @any` и `source_role: @any`.
- `TargetSid` – дескриптор целевого субъекта.
- `Roles` – список добавляемых ролей.

## Возвращаемое решение

"Разрешено", если роли добавлены.

"Запрещено", если роли не добавлены.

## Политика query\_type класса Rbac

Возвращает тип субъекта или объекта.

Тип: политика-выражение.

### Синтаксис

```
query_type {source: Handle}
```

Здесь:

- `Handle` – дескриптор объекта или субъекта.

## Возвращаемое значение

Тип субъекта или объекта.

## Инструменты для сборки решения

Этот раздел содержит описание скриптов, утилит, компиляторов и шаблонов сборки, поставляемых в рамках KasperskyOS Community Edition.

## Скрипты и компиляторы

Этот раздел содержит описание скриптов, утилит и компиляторов, поставляемых в рамках KasperskyOS Community Edition.

## Утилиты и скрипты сборки

В состав KasperskyOS Community Edition входят следующие утилиты и скрипты сборки:

- [nk-gen-c](#)

Компилятор NK (`nk-gen-c`) генерирует набор [транспортных методов и типов](#) на основе EDL-, CDL- и IDL-описаний сущностей, компонентов и интерфейсов. Транспортные методы и типы нужны для формирования, отправки, приема и обработки IPC-сообщений.

- [einit](#)

Утилита `einit` позволяет автоматизировать создание кода инициализирующей сущности `Einit`. Эта сущность первой запускается при загрузке KasperskyOS и запускает остальные сущности, а также создает каналы (соединения) между ними.

- [makekss](#)

Скрипт `makekss` создает модуль безопасности Kaspersky Security System для ядра KasperskyOS.

- [makeimg](#)

Скрипт `makeimg` создает финальный загружаемый образ решения на базе KasperskyOS со всеми запускаемыми сущностями и модулем Kaspersky Security System.

## nk-gen-c

Компилятор NK (`nk-gen-c`) генерирует набор [транспортных методов и типов](#) на основе EDL-, CDL- и IDL-описаний сущностей, компонентов и интерфейсов. Транспортные методы и типы нужны для формирования, отправки, приема и обработки IPC-сообщений.

Транспортные методы и типы генерируются с полностью квалифицированными именами. В именах в качестве префиксов используется [полное имя сущности/компонента/интерфейса](#). (объявленное в соответствующем EDL-, CDL- или IDL-файле) с заменой точек на подчеркивания (`_`).

Компилятор NK принимает EDL-, CDL- или IDL-файл и создает следующие файлы:

- `H`-файл, содержащий объявление и реализацию транспортных методов и типов.
- `D`-файл, в котором перечислены зависимости созданного `C`-файла. Этот файл может быть использован для автоматизации сборки с помощью утилиты `make`.

Синтаксис использования компилятора NK:

```
nk-gen-c [-I PATH][-o PATH][--types][--interface][--client][--server][--extended-errors][--enforce-alignment-check][--help][--version] FILE
```

Параметры:

- `FILE`

Путь к EDL-, CDL- или IDL-описанию сущности, компонента или интерфейса, для которого необходимо сгенерировать транспортные методы и типы.

- `-I PATH`

Путь к директории, содержащей вспомогательные файлы, необходимые для генерации транспортных методов и типов. По умолчанию эти файлы располагаются в директории `/opt/KasperskyOS-Community-Edition-<version>/sysroot-arm-kos/include`.

Также может использоваться для добавления других директорий для поиска файлов, необходимых для генерации.

Чтобы указать более одной директории, можно использовать несколько параметров `-I`.

- `-o PATH`

Путь к существующей директории, в которой будут созданы файлы, содержащие транспортные методы и типы.

- `-h, --help`

Отображает текст справки.

- `--version`

Отображает версию `nk-gen-c`

- `--enforce-alignment-check`

Включает обязательную проверку выравнивания обращений к памяти, даже если такая проверка отключена для целевой платформы. Если проверка включена, то компилятор NK добавляет дополнительные проверки выравнивания в код валидаторов IPC-сообщений.

По умолчанию параметры проверки выравнивания обращения к памяти задаются для каждой платформы в файле `system.platform`.

- `--extended-errors`

Включает [расширенную обработку ошибок](#) в коде клиентских и серверных методов.

## Выборочная генерация

Чтобы уменьшить количество генерируемого компилятором NK кода можно использовать флаги выборочной генерации. Например для сущностей, реализующих интерфейсы, удобно использовать флаг `--server`, а для сущностей, являющихся клиентами интерфейсов, удобно использовать флаг `--client`.

Если ни один из флагов выборочной генерации не указан, компилятор NK создаст все возможные для указанного файла транспортные типы и методы.

Флаги выборочной генерации для описаний интерфейсов (IDL-файлов):

- `--types`

Компилятор создаст только файлы, содержащие все константы и типы, включая переопределенные (`typedef`), из входного IDL-файла, а также типы из импортируемых IDL-файлов, которые используются в типах входного файла.

При этом константы и переопределенные типы из импортируемых IDL-файлов *не будут* явно включены в генерируемые файлы. Если вам необходимо использовать типы из импортируемых файлов в коде, нужно отдельно сгенерировать H-файлы для каждого такого IDL-файла.

- `--interface`

Компилятор создаст файлы, создаваемые с флагом `--types`, а также структуры сообщений запросов и ответов для всех методов этого интерфейса.

- `--client`

Компилятор создаст файлы, создаваемые с флагом `--interface`, а также клиентские прокси-объекты и функции их инициализации для всех методов этого интерфейса.

- `--server`

Компилятор создаст файлы, создаваемые с флагом `--interface`, а также типы и методы диспетчера этого интерфейса.

Флаги выборочной генерации для описаний компонентов (CDL-файлов) и сущностей (EDL-файлов):

- `--types`

Компилятор создаст файлы, создаваемые с флагом `--types` для всех интерфейсов, используемых в этом компоненте/сущности.

При этом явно включены в генерируемые файлы будут только те типы, которые используются в аргументах интерфейсных методов.

- `--interface`

Компилятор создаст файлы, создаваемые с флагом `--types` для этого компонента/сущности, а также файлы, создаваемые с флагом `--interface` для всех интерфейсов, используемых в этом компоненте/сущности.

- `--client`

Компилятор создаст файлы, создаваемые с флагом `--interface`, а также клиентские прокси-объекты и функции их инициализации для всех интерфейсов, используемых в этом компоненте/сущности.

- `--server`

Компилятор создаст файлы, создаваемые с флагом `--interface`, а также типы и методы диспетчера этого компонента/сущности и типы и методы диспетчеров для всех интерфейсов, используемых в этом компоненте/сущности.

## nk-psl-gen-c

Компилятор `nk-psl-gen-c` генерирует исходный код модуля безопасности Kaspersky Security System на основе файла [политики безопасности решения](#) (security.psl) и EDL-описаний сущностей, входящих в решение. Этот код используется скриптом [makekss](#).

Компилятор `nk-psl-gen-c` также позволяет генерировать и запускать код [тестовых сценариев](#) для политики безопасности решения, написанных на языке PAL.

Синтаксис использования компилятора `nk-psl-gen-c`:

```
nk-psl-gen-c [-I PATH][-o PATH][--audit PATH][--tests ARG][--help][--version] FILE
```

Параметры:

- `FILE`

Путь к PSL-описанию политики безопасности решения (security.psl)

- `-I, --include-dir PATH`

Путь к директории, содержащей вспомогательные файлы, необходимые для генерации транспортных методов и типов. По умолчанию эти файлы располагаются в директории `/opt/KasperskyOS-Community-Edition-<version>/sysroot-arm-kos/include`.

Компилятору `nk-psl-gen-c` потребуется доступ ко всем EDL-описаниям сущностей, перечисленных в конфигурации безопасности, а также к CDL- или IDL-описаниям их компонентов и интерфейсов. Для того, чтобы компилятор `nk-psl-gen-c` мог найти эти описания, нужно передать пути к расположению этих описаний в параметре `-I`.

Чтобы указать более одной директории, можно использовать несколько параметров `-I`.

- `-o, --output PATH`

Путь к создаваемому файлу, содержащему код модуля безопасности.

- `-t, --tests ARG`

Флаг контроля генерации кода и запуска тестовых сценариев для политики безопасности решения. Может принимать следующие значения:

- `skip` – код тестовых сценариев не генерируется. Это значение используется по умолчанию, если флаг `--tests` не указан.
- `generate` – код тестовых сценариев генерируется, но не компилируется и не запускается.
- `run` – код тестовых сценариев генерируется, компилируется с помощью компилятора `gcc` и запускается.

- `-a, --audit PATH`

Путь к создаваемому файлу, содержащему код декодера аудита.

- `-h, --help`

Отображает текст справки.

- `--version`

Отображает версию `nk-psl-gen-c`.

## einit

Утилита `einit` позволяет автоматизировать создание кода [инициализирующей сущности Einit](#). Эта сущность первой запускается при загрузке KasperskyOS и запускает остальные сущности, а также создает каналы (соединения) между ними.

Утилита `einit` принимает файл `init`-описания (по умолчанию `init.yaml`) и создает `.c` файл, в котором содержится код инициализирующей сущности `Einit`. Сущность `Einit` затем необходимо собрать с помощью компилятора C, поставляемого в рамках KasperskyOS Community Edition.

Синтаксис использования утилиты `einit`:

```
einit -I PATH -o PATH [--help] FILE
```

Параметры:

- `FILE`

Путь к файлу описания сущностей и соединений `init.yaml`.

- `-I PATH`

Путь к директории, содержащей вспомогательные файлы, необходимые для генерации инициализирующей сущности. По умолчанию эти файлы располагаются в директории `/opt/KasperskyOS-Community-Edition-<version>/sysroot-arm-kos/include`.

- `-o, --out-file PATH`

Путь к создаваемому `.c` файлу с кодом инициализирующей сущности.



- `-h, --help`

Отображает текст справки.

## makekss

Скрипт `makekss` создает [модуль безопасности](#) Kaspersky Security System.

Скрипт вызывает компилятор `nk-psl-gen-c` для генерации исходного кода модуля безопасности, и затем компилирует полученный код, вызывая компилятор C, поставляемый в рамках KasperskyOS Community Edition.

Скрипт принимает файл с описанием политики безопасности решения (по умолчанию `security.psl`) и создает файл модуля безопасности `kss.module`.

Синтаксис использования скрипта `makekss`:

```
makekss --target=ARCH --module=PATH --with-nk="PATH" --with-nktype="TYPE" --with-nkflags="FLAGS" [--output="PATH"] [--help] [--with-cc="PATH"] [--with-cflags="FLAGS"] FILE
```

Параметры:

- `FILE`

Путь к файлу конфигурации безопасности (`.psl`).

- `--target=ARCH`

Архитектура, для которой производится сборка.

- `--module=-lPATH`

Путь к библиотеке `kss_kss`. Этот ключ передается компилятору C для компоновки с этой библиотекой.

- `--with-nk=PATH`

Путь к компилятору `nk-psl-gen-c`, который будет использоваться для генерации исходного кода модуля безопасности. По умолчанию компилятор расположен в `/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin/nk-psl-gen-c`.

- `--with-nktype="TYPE"`

Указывает на тип компилятора NK, который будет использоваться. Для использования компилятора `nk-psl-gen-c`, необходимо указать тип `psl`.

- `--with-nkflags="FLAGS"`

Параметры, с которыми вызывается компилятор `nk-psl-gen-c`.

Компилятору `nk-psl-gen-c` потребуется доступ ко всем EDL-описаниям сущностей, перечисленных в конфигурации безопасности, а также к CDL- или IDL-описаниям их компонентов и интерфейсов. Для того, чтобы компилятор `nk-psl-gen-c` мог найти эти описания, нужно передать пути к расположению этих описаний в параметре `--with-nkflags`, используя параметр `-I` компилятора `nk-psl-gen-c`.

- `--output=PATH`

Путь к создаваемому файлу модуля безопасности.

- `--with-cc=PATH`

Путь к компилятору C, который будет использоваться для сборки модуля безопасности. По умолчанию используется компилятора, поставляемый в рамках KasperskyOS Community Edition.

- `--with-cflags=FLAGS`

Параметры, с которыми вызывается компилятор C.

- `-h, --help`

Отображает текст справки.

## makeimg

Скрипт `makeimg` создает финальный загружаемый образ решения на базе KasperskyOS со всеми запускаемыми сущностями и модулем Kaspersky Security System.

Скрипт принимает список файлов, включая исполняемые файлы всех сущностей, которые нужно добавить в ROMFS загружаемого образа и создает следующие файлы:

- образ решения;
- образ решения без таблиц символов (`.stripped`);
- образ решения с отладочными таблицами символов (`.dbg.syms`).

Синтаксис использования скрипта `makeimg`:

```
makeimg --target=ARCH --sys-root=PATH --with-toolchain=PATH --ldscript=PATH --img-
src=PATH --img-dst=PATH --with-init=PATH [--with-extra-asflags=FLAGS][--with-extra-
ldflags=FLAGS][--help] FILES
```

Параметры:

- `FILES`

Список путей к файлам, включая исполняемые файлы всех сущностей, которые нужно добавить в romfs.

По умолчанию, имя исполняемого файла сущности должно совпадать с её [кратким именем](#). Можно изменить имя файла, из которого будет запущена сущность, указав его в поле `path` [init-описания сущности](#).

Модуль безопасности (`ksm.module`) нужно указывать явно, иначе он не будет включен в образ решения. Сущность `Einit` указывать не нужно, так как она будет включена в образ решения автоматически.

- `--target=ARCH`

Архитектура, для которой производится сборка.

- `--sys-root=PATH`

Путь к корневому каталогу `sysroot`. По умолчанию этот каталог расположен в `/opt/KasperskyOS-Community-Edition-<version>/sysroot-arm-kos/`.

- `--with-toolchain=PATH`

Путь к набору вспомогательных утилит, необходимых для сборки решения. По умолчанию эти утилиты расположены в `/opt/KasperskyOS-Community-Edition-<version>/toolchain/`.

- `--ldscript=PATH`

Путь к скрипту компоновщика, необходимому для сборки решения. По умолчанию этот скрипт расположен в `/opt/KasperskyOS-Community-Edition-<version>/libexec/arm-kos/`.

- `--img-src=PATH`

Путь к заранее скомпилированному ядру KasperskyOS, не содержащему romfs. По умолчанию ядро расположено в `/opt/KasperskyOS-Community-Edition-<version>/libexec/arm-kos/`.

- `--img-dst=PATH`

Путь к создаваемому файлу образа.

- `--with-init=PATH`

Путь к исполняемому файлу инициализирующей сущности `Einit`.

- `--with-extra-asflags=FLAGS`

Дополнительные флаги для ассемблера AS.

- `--with-extra-ldflags=FLAGS`

Дополнительные флаги для компоновщика LD.

- `-h, --help`

Отображает текст справки.

## Кросс-компиляторы

### Свойства кросс-компиляторов KasperskyOS

Кросс-компиляторы, входящие в состав KasperskyOS Community Edition, поддерживают процессоры с архитектурой `arm`.

В toolchain в составе KasperskyOS Community Edition входят следующие инструменты для кросс-компиляции:

- GCC:
  - `arm-kos-gcc`
  - `arm-kos-g++`
- Binutils:
  - Ассемблер AS: `arm-kos-as`
  - Компоновщик LD: `arm-kos-ld`

В GCC, кроме стандартных макросов, определен дополнительный макрос `__KOS__=1`. Использование этого макроса упрощает портирование программного кода приложений на KasperskyOS, а также разработку платформонезависимых приложений.

Чтобы просмотреть список стандартных макросов GCC, выполните следующую команду:

```
echo '' | arm-kos-gcc -dM -E -
```

## Особенности работы компоновщика

При выполнении сборки исполняемого файла сущности компоновщик по умолчанию связывает следующие библиотеки в указанном порядке:

1. `libc` – стандартная библиотека языка C.
2. `libm` – библиотека, реализующая математические функции стандартной библиотеки языка C.
3. `libvfs_stubs` – библиотека, содержащая заглушки функций ввода/вывода (например, `open`, `socket`, `read`, `write`).
4. `libkos` – библиотека, состоящая из двух частей. Первая часть предоставляет C-интерфейс для доступа к функциям ядра KasperskyOS. Она доступна через заголовочные файлы, находящиеся в директории `coresrv`, например: `#include <coresrv/vmm/vmm_api.h>`. Вторая часть библиотеки `libkos` является оберткой над первой частью и содержит дополнительные функции синхронизации: `mutex`, `semaphore`, `event`. Взаимодействие других библиотек (включая `libc`) с ядром происходит через библиотеку `libkos`.
5. `libenv` – клиентская библиотека подсистемы настройки окружения сущностей (переменных окружения, аргументов функции `main` и пользовательских конфигураций).
6. `libsrvtransport-u` – внутренняя библиотека с реализацией транспорта межпроцессного взаимодействия сервисов ядра KasperskyOS.

## Подготовка загрузочного образа решения

Для автоматизации процесса подготовки загрузочного образа решения нужно настроить систему сборки. За основу можно взять системы сборки, реализованные в примерах.

Этот раздел описывает различные способы настройки системы сборки для подготовки загрузочного образа решения.

## Использование шаблона Makefile из состава KasperskyOS Community Edition

Для упрощения процесса подготовки загрузочного образа решения с использованием системы сборки `make` вы можете использовать шаблон `build.mk` из состава KasperskyOS Community Edition. Файл шаблона располагается по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/common/build.mk
```

Чтобы подготовить систему сборки `make` с использованием шаблона `build.mk`, в скрипте сборки `Makefile`:

1. Задайте значение переменной `targets`. В значении переменной перечислите через пробел все сущности, входящие в решение. Указывать сущности `Einit`, `kl.core.Core` необязательно, так как они обрабатываются отдельно.

2. Для каждой сущности, указанной в переменной `targets`, задайте значения следующим переменным:

- `<имя-сущности>-objects` – список объектных файлов сущности. Необходимо перечислить все объектные файлы.

Имена объектных файлов получаются из имен исходных файлов сущности по следующим правилам:

- `*.c → *.o`
- `*.idl → *.idl.o`
- `*.cdl → *.cdl.o`
- `*.edl → *.edl.o`
- `<имя-сущности>-ldflags` – список флагов, передаваемых компоновщику. Если сущность использует виртуальную файловую систему, необходимо передать флаги, указанные в переменной `LIBVFS_REMOTE`.
- `<имя-сущности>-base` – адрес загрузки сущности в шестнадцатеричной системе. Если переменная не указана, адрес присваивается автоматически. Используйте эту переменную для отладки сущности.

Этот же адрес передается отладчику с помощью следующей команды, которая может быть добавлена в `.gdbinit`-файл:

```
add-symbol-file <имя-сущности> <адрес-загрузки-сущности>
```

3. Если раздел `ROMFS` должен содержать дополнительные файлы, задайте значение переменной `ROMFS-FILES`. В значении переменной перечислите файлы через пробел. Если вы используете виртуальную файловую систему, необходимо передать файл, указанный в переменной `VFS_ENTITY`.

4. Добавьте инструкцию импорта шаблона `build.mk` с помощью следующей команды:

```
include /opt/KasperskyOS-Community-Edition-<version>/common/build.mk
```

В файле шаблона `build.mk` реализованы следующие цели сборки:

- **sim** (цель по умолчанию) – запуск загрузочного образа решения с помощью эмулятора QEMU. При запуске эмулятор QEMU может осуществлять захват мыши, о чем будет написано в заголовке окна эмулятора.
- **kos-image** – сборка загрузочного образа решения для запуска на целевой аппаратной платформе.
- **gdbsim** – запуск загрузочного образа решения с возможностью отладки. После старта эмуляции QEMU ожидает старта отладчика. Для этого в другой командной строке необходимо вызвать цель **gdb**. Убедитесь, что порт протокола TCP/IP 1234 открыт, например с помощью команды `netstat -anput`. Порт 1234 прослушивается программой `gdbserver`, которая используется для удаленной отладки приложений и входит в состав эмулятора QEMU. При использовании отладчика `gdb` необходимо использовать аппаратные точки останова (`hbreak`). Эмулятор QEMU, который используется в примерах, запускается с ключом `-enable-kvm`, из-за чего использование обычных точек останова (`break`) невозможно.
- **gdb** – запуск отладчика. После запуска отладчика для сущностей, которые требуется отлаживать, выполните следующие команды:

```
add-symbol-file <имя-сущности> <адрес-загрузки-сущности>
target remote localhost:1234
```

Пример

В этом примере используется система сборки make. Помимо действий, выполняемых в шаблоне build.mk, в скрипте сборки необходимо указать сущность hello и список объектных файлов этой сущности. Адрес загрузки указывается для целей отладки решения.

#### Makefile

```
# Список сущностей для сборки.
targets = hello

# Список объектных файлов сущности "hello".
hello-objects = hello.o hello.edl.o

# Адрес загрузки сущности "hello" (в hex).
hello-base = 800000

# Включить шаблон с общими правилами сборки.
include ../common/build.mk
```

Чтобы запустить систему сборки make, выполните команду `make hello`.

Чтобы запустить пример hello, находясь в директории `/opt/KasperskyOS-Community-Edition-<version>/examples/hello`, выполните команду `make`.

## Использование CMake из состава KasperskyOS Community Edition

Система автоматизации сборки CMake поддерживает кросс-компиляцию приложений. Для выполнения кросс-компиляции с помощью CMake требуется указать путь к файлу расширения системы сборки (`toolchain.cmake`).

Чтобы выполнить кросс-компиляцию приложения для KasperskyOS, задайте значение переменной: `DCMAKE_TOOLCHAIN_FILE=/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/toolchain.cmake`

В состав KasperskyOS Community Edition входит библиотека `platform`, содержащая набор готовых скриптов для системы CMake.

Чтобы подготовить приложение к отладке:

1. В файле CMakeLists.txt задайте значение параметра `LINK_FLAGS` для приложения, которое вы хотите отлаживать, следующим образом:

```
set_target_properties (<имя-приложения> PROPERTIES LINK_FLAGS "-Ttext <адрес-секции-text>")
```

Скрипт автоматически создает .gdbinit-файлы. В .gdbinit-файлах находится набор команд для отладчика GDB. Этот набор команд определяет, по какому адресу отладчик GDB загружает сущности для отладки.

2. Выполните сборку приложения.

## Использование собственной системы сборки

Вы можете использовать другие системы сборки или реализовать собственную систему сборки для подготовки загрузочного образа решения.

Для того чтобы подготовить загрузочный образ решения, система сборки должна включать следующие действия:

1. Генерацию кода [транспортных методов и типов](#), используемых для формирования, отправки, приема и обработки IPC-сообщений между сущностями, входящими в решение.

Для этого воспользуйтесь [компилятором NK](#). В аргументах команды передайте путь к файлам EDL-, CDL- и IDL-описаний сущностей, компонентов и интерфейсов.

2. Сборку всех сущностей, входящих в решение.

Для этого воспользуйтесь [кросс-компиляторами](#), входящими в состав KasperskyOS Community Edition.

3. Сборку инициализирующей [сущности Einit](#).

Для генерации кода сущности Einit воспользуйтесь утилитой [einit](#). В аргументах команды передайте путь к файлу файл init-описания (по умолчанию `init.yaml`).

Сущность Einit затем необходимо собрать с помощью компилятора C, поставляемого в рамках KasperskyOS Community Edition.

4. Сборку модуля ядра с подсистемой Kaspersky Security System.

Для этого воспользуйтесь скриптом [makekss](#). В аргументах команды передайте путь к файлу конфигурации безопасности (по умолчанию `security.ps1`).

5. Создание образа решения.

Для этого воспользуйтесь скриптом [makeimg](#). В аргументах команды передайте исполняемые ELF-файлы сущностей, модуль ядра с Kaspersky Security System, образ ядра KasperskyOS и любые дополнительные файлы.

## Развертывание загрузочного образа решения на целевых устройствах

Чтобы развернуть загрузочный образ решения на целевом устройстве:

1. Подключите носитель информации, с которого вы планируете запускать загрузочный образ решения на целевых устройствах.

2. Найдите блочное устройство, соответствующее подключенному носителю, например с помощью команды:

```
fdisk -l
```

3. При необходимости создайте на носителе новый раздел, в котором будет развернут загрузочный образ решения, например с помощью утилиты `fdisk`.

4. Если в разделе отсутствует файловая система, создайте ее, например с помощью утилиты `mkfs`.

Вы можете использовать любую файловую систему, которую поддерживает загрузчик GRUB 2.

5. Смонтируйте носитель информации.

```
mkdir /mnt/kos_device && mount /dev/sdXY /mnt/kos_device
```

Здесь `/mnt/kos_device` – точка монтирования; `/dev/sdXY` – имя блочного устройства; X – буква, соответствующая подключенному носителю; Y – номер раздела.

6. Установите на носитель загрузчик операционной системы [GRUB 2](#).

Чтобы установить GRUB 2, выполните следующую команду:

```
grub-install --force --removable \  
--boot-directory=/mnt/kos_device/boot /dev/sdX
```

Здесь `/mnt/kos_device` – точка монтирования `/dev/sdX` – имя блочного устройства; X – буква, соответствующая подключенному носителю.

7. Скопируйте загрузочный образ решения в корневую директорию смонтированного носителя.

8. В файле `/mnt/kos_device/boot/grub/grub.cfg` добавьте секцию `menuentry`, указывающую на загрузочный образ решения.

```
menuentry "KasperskyOS" {  
  multiboot /my_kasperskyos.img  
  boot  
}
```

9. Размонтируйте носитель.

```
umount /mnt/kos_device
```

Здесь `/mnt/kos_device` – точка монтирования.

После выполнения этих действий вы можете запускать KasperskyOS с этого носителя.



# Паттерны безопасности при разработке под KasperskyOS

Каждое решение на базе KasperskyOS имеет определенные сценарии использования и предназначено для противодействия конкретным угрозам безопасности. Тем не менее, существуют типовые сценарии и угрозы, которые встречаются во многих решениях. Этот раздел описывает типовые риски и угрозы, а также содержит описание архитектурных паттернов, применение которых позволит повысить безопасность решения.

*Паттерн (или шаблон) безопасности* описывает конкретную повторяющуюся проблему безопасности, которая возникает в определенных известных контекстах, а также предлагает хорошо зарекомендовавшую себя общую схему решения такой проблемы безопасности. Паттерн это не законченный проект, который можно преобразовать непосредственно в код, а решение общей проблемы, встречающейся в различных проектах.

*Система паттернов безопасности* – это набор паттернов безопасности вместе с инструкциями по их реализации, сочетанию и практическому использованию в проектировании безопасных программных систем.

Паттерны безопасности решают проблемы безопасности на разных уровнях: начиная от паттернов архитектурного уровня, включающих высокоуровневый дизайн системы, и заканчивая паттернами уровня реализации, содержащими рекомендации о том, как реализовать функции или методы.

Этот раздел содержит описание набора паттернов безопасности, примеры реализации которых содержатся в составе KasperskyOS Community Edition.

Паттернам безопасности посвящено множество работ в области информационной безопасности. Для каждого паттерна приводится список работ, использованных при подготовке его описания.

## Паттерн Distrustful Decomposition

### Описание

При использовании монолитного приложения появляется необходимость дать все необходимые для его работы привилегии одному процессу. Эту проблему решает паттерн **Distrustful Decomposition**.

Целью паттерна **Distrustful Decomposition** является разделение функциональности программы по отдельным процессам, требующим различного уровня привилегий, и контроля взаимодействия между этими процессами вместо создания монолитной программы.

Использование паттерна **Distrustful Decomposition** уменьшает:

- поверхность атаки для каждого из процессов;
- функциональность и данные, которые станут доступны злоумышленнику, если один из процессов будет скомпрометирован.

### Альтернативные названия

**Privilege Reduction.**

## Контекст

Различные функции приложения требуют разного уровня привилегий.

## Проблема

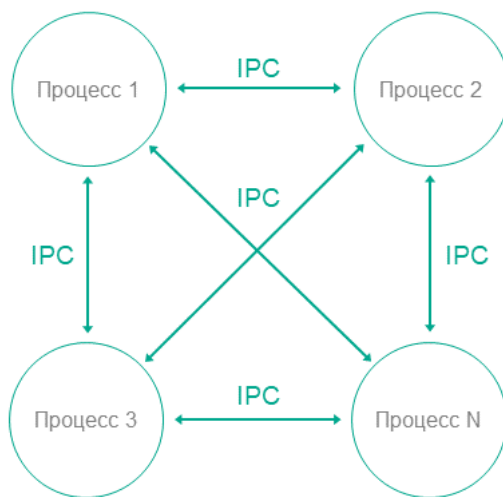
Наивная реализация приложения объединяет множество разнородных по необходимым привилегиям функций в одном компоненте, вынуждая его запускаться с максимальным из необходимых уровней привилегий.

## Решение

Паттерн **Distrustful Decomposition** разделяет функциональность по отдельным процессам и изолирует возможные уязвимости в небольшом подмножестве системы. Злоумышленник в случае успешной атаки будет иметь в своем распоряжении функциональность и данные только одного скомпрометированного компонента, но не всего приложения.

## Структура

Этот паттерн разбивает одно монолитное приложение на несколько, которые выполняются как отдельные процессы, потенциально имеющие разные привилегии. Каждый процесс реализует небольшой, четко определенный набор функций приложения. Процессы обмениваются данными, используя механизм межпроцессного взаимодействия.



## Работа

- В KasperskyOS приложение разбивается на сущности.
- Сущности могут обмениваться сообщениями по IPC.
- Пользователь или удаленная система подключается к процессу, который обеспечивает необходимую функциональность, с уровнем привилегий, достаточным для выполнения запрошенных функций.

## Рекомендации по реализации

Взаимодействие между процессами может быть однонаправленным или двунаправленным. Рекомендуется всегда, когда это возможно, использовать однонаправленное взаимодействие, в противном случае увеличивается поверхность атаки на отдельные компоненты и, соответственно, снижается уровень защищенности системы в целом. В случае двустороннего IPC процессы не должны доверять двустороннему обмену данными. Например, если для IPC используется файловая система, то нельзя доверять содержимому файла.

## Особенности реализации в KasperskyOS

В универсальных ОС (например Linux, Windows) этот паттерн не использует ничего, кроме стандартной модели процессов/привилегий, уже существующей в этих ОС. Каждая программа выполняется в собственном пространстве процессов с потенциально разными привилегиями пользователя в каждом процессе, однако атака на ядро ОС снижает ценность применения этого паттерна.

Специфика применения этого паттерна при разработке под KasperskyOS состоит в том, что контроль над процессами и IPC возложен на [микроядро](#), атака на которое сложна. Для контроля IPC используется модуль безопасности Kaspersky Security Module и [политики безопасности](#).

За счет использования механизмов KasperskyOS достигается высокий уровень надежности программной системы при том же или меньшем объеме усилий разработчика в сравнении с использованием этого же паттерна в программах под универсальные ОС.

Кроме этого, KasperskyOS предоставляет возможность гибкой настройки политик безопасности. При этом процесс задания и изменения политик безопасности потенциально независим от процесса разработки самих приложений.

## Связанные паттерны

Использование паттерна `Distrustful Decomposition` предполагает использование паттернов [Defer to Kernel](#) и [Policy Decision Point](#).

## Примеры реализации

Примеры реализации паттерна `Distrustful Decomposition`:

- [Secure Logger](#)
- [Separate Storage](#)

## Источники

Паттерн `Distrustful Decomposition` подробно рассмотрен в следующих работах:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March–October 2009). Software Engineering Institute.  
[https://resources.sei.cmu.edu/asset\\_files/TechnicalReport/2009\\_005\\_001\\_15110.pdf](https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf)
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. <https://dc.etsu.edu/etd/1119>

## Пример Secure Logger

Пример Secure Logger демонстрирует использование паттерна [Distrustful Decomposition](#) для решения задачи разделения функциональности чтения и записи в журнал событий.

### Архитектура примера

Цель безопасности в примере Secure Logger заключается в том, чтобы предотвратить возможность искажения или удаления информации в журнале событий. В примере для достижения этой цели безопасности используются возможности, предоставляемые KasperskyOS.

При рассмотрении системы журналирования можно выделить следующие функциональные шаги:

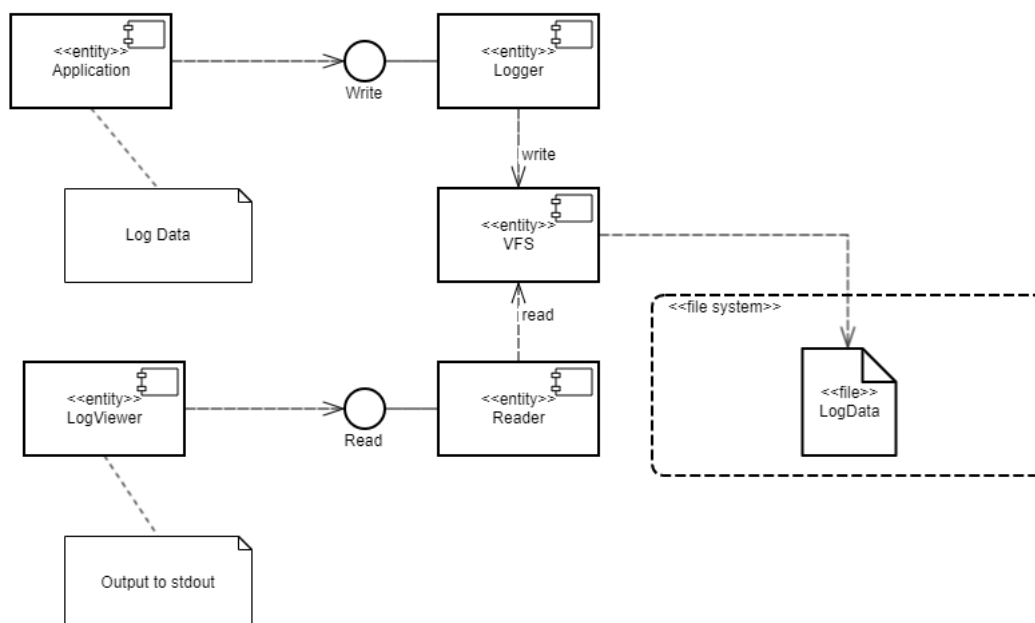
- генерация информации для записи в журнал;
- сохранение информации в журнал;
- чтение записей из журнала;
- предоставление записей в удобном для потребителя виде.

Таким образом, подсистему журналирования можно разделить на четыре процесса в зависимости от необходимых функциональных возможностей каждого процесса.

Для этого пример Secure Logger содержит четыре сущности: Application, Logger, Reader и LogViewer.

- Сущность Application инициирует создание записей в журнале событий, поддерживаемом сущностью Logger.
- Сущность Logger создает записи в журнале и записывает их на диск.
- Сущность Reader читает записи с диска для передачи сущности LogViewer.
- Сущность LogViewer передает записи пользователю.

IPC-интерфейс, который предоставляет сущность Logger, предназначен *только* для записи в хранилище. IPC-интерфейс сущности Reader предназначен только для чтения из хранилища. Архитектура примера выглядит следующим образом:



- Сущность **Application** использует интерфейс сущности **Logger** для сохранения записей.
- Сущность **LogViewer** использует интерфейс сущности **Reader** для чтения записей и предоставления их пользователю.

В общем случае сущность **LogViewer** имеет внешние каналы взаимодействия с пользователем (прием команд на чтение данных, предоставление данных пользователю). Очевидно, что эта сущность является недоверенным компонентом системы, через которую может проводиться атака. Однако даже в случае успешной атаки, вплоть до внедрения произвольно исполняемого кода в сущность **LogViewer**, информация в журнале не будет искажена, так как эта сущность может пользоваться только интерфейсом чтения данных, через который искажение или удаление невозможно. При этом **LogViewer** не имеет возможности получить доступ к другим IPC-интерфейсам, так как доступ контролируется модулем безопасности.

Политика безопасности в примере **Secure Logger** имеет следующие особенности:

- Сущность **Application** имеет возможность обращаться к сущности **Logger** для создания новой записи в журнале событий.
- Сущность **LogViewer** имеет возможность обращаться к сущности **Reader** для чтения записей из журнала событий.
- Сущность **Application** *не* имеет возможности обращаться к сущности **Reader** для чтения записей из журнала событий.
- Сущность **LogViewer** *не* имеет возможности обращаться к сущности **Logger** для создания новой записи в журнале событий.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/secure_logger
```

## Сборка и запуск примера

## Пример Separate Storage

Пример `Separate Storage` демонстрирует использование паттерна [Distrustful Decomposition](#) для решения задачи раздельного хранения данных для доверенных и недоверенных приложений.

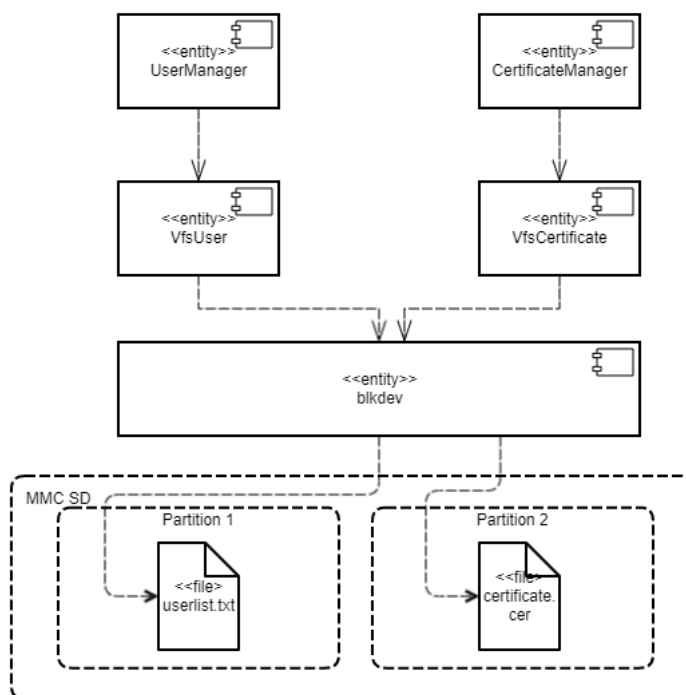
### Архитектура примера

Пример `Separate Storage` содержит две пользовательские сущности: `UserManager` и `CertificateManager`.

Эти сущности работают с данными, которые размещаются в соответствующих файлах:

- Сущность `UserManager` работает с данными из файла `userlist.txt`;
- Сущность `CertificateManager` работает с данными из файла `certificate.cer`.

Каждая из этих сущностей использует собственный экземпляр сущности VFS для доступа к отдельной файловой системе. При этом каждая сущность VFS включает в себя драйвер блочного устройства, связанный с отдельным логическим разделом диска. Сущность `UserManager` не имеет доступа к файловой системе сущности `CertificateManager` и наоборот.



Такая архитектура гарантирует, что в случае атаки или ошибки в любой из сущностей `UserManager` и `CertificateManager`, эта сущность не сможет получить доступ к файлу, который не предназначен для выполнения ее работы.

Политика безопасности в примере `Separate Storage` имеет следующие особенности:

- Сущность `UserManager` имеет доступ к файловой системе *только* через сущность `VfsUser`.
- Сущность `CertificateManager` имеет доступ к файловой системе *только* через сущность `VfsCertificate`.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/separate_storage
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Подготовка SD-карты для запуска на Raspberry Pi 4 B

Для запуска примера `Separate Storage` на Raspberry Pi 4 B необходимы следующие дополнительные действия:

- SD-карта, помимо загрузочного раздела с образом решения, должна также содержать 2 дополнительных раздела с файловой системой `ext2` или `ext3`.
- первый дополнительный раздел должен содержать файл `userlist.txt` из директории `./resources/files/`
- второй дополнительный раздел должен содержать файл `certificate.cer` из директории `./resources/files/`

Для запуска примера `Separate Storage` на Raspberry Pi 4 B можно использовать SD-карту, подготовленную для запуска примера `embed_ext2_with_separate_vfs` на Raspberry Pi 4 B, скопировав файлы `userlist.txt` и `certificate.cer` на соответствующие разделы.

## Паттерн Defer to Kernel

### Описание

Паттерн `Defer to Kernel` предполагает использование преимущества контроля разрешений на уровне ядра ОС.

Целью этого паттерна является четкое отделение функциональности, требующей повышенных привилегий, от функциональности, не требующей повышенных привилегий, с помощью механизмов, доступных на уровне ядра ОС. Использование механизмов ядра позволяет не реализовывать новых средств для арбитража решений безопасности на уровне пользователя.

### Альтернативные названия

`Policy Enforcement Point (PEP)`, `Protected System`, `Enclave`.

## Контекст

Паттерн `Defer to Kernel` применим, если система имеет следующие характеристики:

- В системе есть процессы без повышенных привилегий, в том числе пользовательские процессы.
- Некоторые функции системы требуют повышенных привилегий, которые необходимо проверять перед предоставлением процессам доступа к данным.
- Необходимо проверять не только привилегии запрашивающего процесса, но и общую допустимость запрошенной операции в контексте работы всей системы и ее общей безопасности.

## Проблема

В условиях разделения функциональности по разным процессам с разным уровнем привилегий необходимо проверять привилегии при выполнении запроса от одного процесса к другому. Выполнять такие проверки и выдавать разрешения должен доверенный код, минимально подверженный атакам. Доверенность прикладного кода почти всегда под вопросом как в силу его объема, так и в силу его направленности на реализацию функциональных требований.

## Решение

Отделить привилегированную функциональность и данные от непривилегированных на уровне процессов и отдать ядру ОС контроль межпроцессных взаимодействий (IPC) с проверкой прав доступа при запросе функциональности или данных, требующих повышенных привилегий, а также с проверкой общего состояния системы и состояний отдельных процессов в момент запроса.

## Структура



## Работа

- Функциональность и управление данными с разными привилегиями разделены между процессами.
- Изоляцию процессов обеспечивает ядро ОС.
- `Процесс-1` хочет запросить привилегированную функциональность или данные у `Процесса-2`, используя IPC.
- Ядро контролирует IPC и разрешает или не разрешает коммуникацию исходя из политик безопасности и доступной ему информации о контексте работы и состоянии `Процесса-1`.



## Рекомендации по реализации

Для того чтобы конкретная реализация паттерна работала безопасно и надежно, необходимо следующее:

- **Изоляция**

Необходимо обеспечить полную и гарантированную изоляцию процессов.

- **Невозможность обойти ядро**

Абсолютно все IPC-взаимодействия должны контролироваться ядром.

- **Самозащита ядра**

Необходимо обеспечить доверенность ядра, его собственную защиту от компрометации.

- **Доказуемость**

Требуется определенный уровень гарантий безопасности и надежности в отношении ядра.

- **Возможность внешнего вычисления разрешений о доступе**

Необходимо, чтобы разрешения о доступе вычислялись на уровне ОС, а не были реализованы в прикладном коде.

Для этого, в частности, необходимо предоставить инструменты для описания политик доступа, чтобы политики безопасности были отделены от бизнес-логики.

## Особенности реализации в KasperskyOS

Ядро KasperskyOS гарантирует изоляцию сущностей и представляет собой Policy Enforcement Point (PEP).

## Связанные паттерны

Паттерн `Defer to Kernel` является частным случаем паттернов [Distrustful Decomposition](#) и [Policy Decision Point](#). Паттерн `Policy Decision Point` определяет абстрактный процесс, перехватывающий все запросы к ресурсам и проверяющий их на соответствие заданной политике безопасности. Специфика паттерна `Defer to Kernel` в том, что эту проверку выполняет ядро ОС – это более надежное и портируемое решение, сокращающее время разработки и тестирования.

## Следствия

Перенос ответственности за применение политик доступа на ядро ОС приводит к отделению политик безопасности от бизнес-логики (которая может быть очень сложна), что упрощает разработку и повышает портируемость за счет использования функций ядра ОС.

Кроме этого, появляется возможность доказать безопасность решения в целом, доказав правильность работы ядра. Сложность доказуемости правильной работы кода нелинейно растет с увеличением его размера. Паттерн `Defer to Kernel` минимизирует объем доверенного кода – при условии, что ядро ОС невелико.

## Примеры реализации

Пример реализации паттерна `Defer to Kernel`: [Пример Defer to Kernel](#).

## Источники

Паттерн `Defer to Kernel` подробно рассмотрен в следующих работах:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March–October 2009). Software Engineering Institute.  
[https://resources.sei.cmu.edu/asset\\_files/TechnicalReport/2009\\_005\\_001\\_15110.pdf](https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf)
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. <https://dc.etsu.edu/etd/1119>
- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006).

## Пример `Defer to Kernel`

Пример `Defer to Kernel` демонстрирует использование паттернов [Defer to Kernel](#) и [Policy Decision Point](#).

Пример `Defer to Kernel` содержит три пользовательские сущности: `PictureManager`, `ValidPictureClient` и `NonValidPictureClient`.

В этом примере сущности `ValidPictureClient` и `NonValidPictureClient` обращаются к сущности `PictureManager` для получения информации.

Только сущности `ValidPictureClient` разрешено взаимодействие с сущностью `PictureManager`.

Ядро KasperskyOS гарантирует изоляцию сущностей.

Контроль взаимодействия сущностей в KasperskyOS вынесен в модуль безопасности Kaspersky Security Module. Эта подсистема анализирует каждый отправляемый запрос и ответ и на основе заданной политики безопасности выносит решение: разрешить или запретить его доставку.

Политика безопасности в примере `Defer to Kernel` имеет следующие особенности:

- Сущности `ValidPictureClient` явно разрешено взаимодействие с сущностью `PictureManager`.
- Сущности `NonValidPictureClient` взаимодействие с сущностью `PictureManager` не разрешено явно. Таким образом, это взаимодействие запрещено (*принцип Default Deny*).

## Динамическое создание IPC-каналов

Пример также демонстрирует возможность [динамического создания IPC-каналов между сущностями](#). Динамическое создание IPC-каналов осуществляется с помощью сервера имен – специального сервиса ядра, представленного сущностью `NameServer`. Возможность динамического создания IPC-каналов позволяет изменять топологию взаимодействия сущностей "на лету".

Любая сущность, которой разрешено взаимодействие с `NameServer` по IPC, может зарегистрировать в сервере имен свои интерфейсы. Другая сущность может запросить у сервера имен зарегистрированные интерфейсы, после чего осуществить подключение к нужному интерфейсу.

При этом все взаимодействия по IPC (даже созданные динамически) контролируются с помощью модуля безопасности.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/defer_to_kernel
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Паттерн Policy Decision Point

### Описание

Паттерн Policy Decision Point предполагает инкапсуляцию вычисления решений на основе политик безопасности в отдельный компонент системы, который обеспечивает выполнение проверок политик безопасности в полном объеме и в правильной последовательности.

### Альтернативные названия

Check Point, Access Decision Function.

### Контекст

Система имеет функции с разным уровнем привилегий, а политика безопасности нетривиальна (содержит много правил).

### Проблема

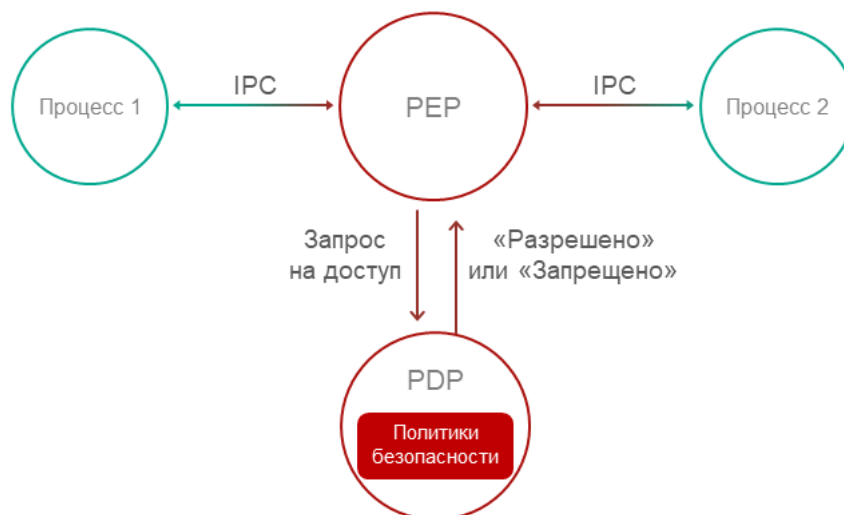
Если проверки соблюдения политик безопасности разнесены по разным компонентам системы, возникают следующие проблемы:

- необходимо тщательно контролировать, что выполняются все необходимые проверки во всех необходимых случаях;
- сложно обеспечивать правильный порядок выполнения проверок;
- сложно доказать правильность работы системы проверок, ее целостность и непротиворечивость;
- политики безопасности связаны с бизнес-логикой, поэтому их изменение влечет необходимость менять логику, что усложняет поддержку и увеличивает вероятность ошибок.

### Решение

Все проверки соблюдения политик безопасности проводятся в отдельном компоненте Policy Decision Point (PDP). Этот компонент отвечает за обеспечение правильного порядка проверок и за их полноту. Происходит отделение проверки политик от кода, реализующего бизнес-логику.

## Структура



## Работа

- Policy Enforcement Point (PEP) получает запрос на доступ к функциональности или данным. PEP может представлять собой, например, ядро ОС. Подробнее см. [Паттерн Defer to Kernel](#).
- PEP собирает атрибуты запроса, необходимые для принятия решений по управлению доступом.
- PEP запрашивает решение по управлению доступом у Policy Decision Point (PDP).
- PDP вычисляет решение о предоставлении доступа на основе политик безопасности и информации, полученной в запросе от PEP.
- PEP отклоняет или разрешает взаимодействие на основе решения PDP.

## Рекомендации по реализации

При реализации необходимо учитывать проблему "Время проверки vs. Время использования". Например, если политика безопасности зависит от быстро меняющегося статуса какого-либо объекта системы, вычисленное решение так же быстро теряет актуальность. В системе, использующей паттерн **Policy Decision Point**, необходимо позаботиться о минимизации интервала между принятием решения о доступе и моментом выполнения запроса на основе этого решения.

## Особенности реализации в KasperskyOS

Ядро KasperskyOS гарантирует изоляцию сущностей и представляет собой Policy Enforcement Point (PEP).

Контроль взаимодействия сущностей в KasperskyOS вынесен в модуль безопасности [Kaspersky Security Module](#). Этот модуль анализирует каждый отправляемый запрос и ответ и на основе заданной политики безопасности выносит решение: разрешить или запретить его доставку. Таким образом, Kaspersky Security Module выполняет роль Policy Decision Point (PDP).

## Следствия

Паттерн позволяет настраивать политики безопасности без внесения изменений в код, реализующий бизнес-логику, и делегировать сопровождение системы с точки зрения информационной безопасности.

## Связанные паттерны

Использование паттерна Policy Decision Point предполагает использование паттернов [Distrustful decomposition](#) и [Defer to Kernel](#).

## Примеры реализации

Пример реализации паттерна Policy Decision Point: [Пример Defer to Kernel](#).

## Источники

Паттерн Policy Decision Point подробно рассмотрен в следующих работах:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March–October 2009). Software Engineering Institute.  
[https://resources.sei.cmu.edu/asset\\_files/TechnicalReport/2009\\_005\\_001\\_15110.pdf](https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf) 
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. <https://dc.etsu.edu/etd/1119> 
- Schumacher, Markus, Fernandez–Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006).
- Bob Blakley, Craig Heath, and members of The Open Group Security Forum. "Security Design Patterns" (April 2004). The Open Group. <https://pubs.opengroup.org/onlinepubs/9299969899/toc.pdf> 

# Паттерн Privilege Separation

## Описание

Паттерн Privilege Separation предполагает использование непривилегированных изолированных модулей системы для взаимодействия с клиентами (другими модулями или пользователями), которые не имеют привилегий. Целью паттерна Privilege Separation является уменьшение количества кода, выполняемого с особыми привилегиями, не влияющее на функциональность программы и не ограничивающее ее.

Паттерн Privilege Separation является частным случаем [паттерна Distrustful Decomposition](#).

## Пример

Неаутентифицированный пользователь подключается к системе, в которой есть функции, требующие повышенных привилегий.

## Контекст

В системе есть компоненты с большой поверхностью атаки из-за большого числа связей с ненадежными источниками и/или сложной, потенциально подверженной ошибкам реализации.

## Проблема

Когда клиент, имеющий неизвестные привилегии, взаимодействует с привилегированным компонентом системы, возникают риски компрометации данных и функциональности, доступных этому компоненту.

## Решение

Взаимодействие с ненадежными клиентами необходимо вести только через специально выделенные компоненты, у которых нет привилегий. Важно, что паттерн *Privilege Separation* не изменяет функциональность системы, он лишь разделяет функциональность на компоненты с разными привилегиями.

## Работа

Работа паттерна делится на две фазы:

- **Pre-Authentication.** Клиент еще не аутентифицирован. Он отправляет запрос к привилегированному мастер-процессу. Мастер-процесс создает дочерний процесс, лишенный привилегий (в том числе, доступа к файловой системе), который выполняет аутентификацию клиента.
- **Post-Authentication.** Клиент аутентифицирован и авторизован. Привилегированный мастер-процесс создает новый дочерний процесс, обладающий привилегиями, соответствующими правам клиента. Этот процесс отвечает за все дальнейшее взаимодействие с клиентом.

## Рекомендации по реализации в KasperskyOS

На этапе **Pre-Authentication** мастер-процесс может хранить состояние каждого непривилегированного процесса в виде конечного автомата и изменять состояние автомата при аутентификации.

Запросы дочерних процессов к мастер-процессу выполняются с использованием стандартных механизмов IPC. При этом контроль взаимодействий осуществляется с помощью модуля безопасности Kaspersky Security Module.

## Следствия

Если атакующий получает контроль над непривилегированным процессом, он не получит доступа ни к каким привилегированным функциям или данным. Если он получает контроль над авторизованным процессом, он получит только привилегии этого процесса.

Кроме того, организованный таким образом код проще проверять и тестировать – особого внимания требует лишь функциональность, работающая с повышенными привилегиями.

## Примеры реализации

Пример реализации паттерна `Privilege Separation`: [Пример Device Access](#).

## Источники

Паттерн `Privilege Separation` подробно рассмотрен в следующих работах:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March–October 2009). Software Engineering Institute.  
[https://resources.sei.cmu.edu/asset\\_files/TechnicalReport/2009\\_005\\_001\\_15110.pdf](https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf)
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. <https://dc.etsu.edu/etd/1119>

## Пример Device Access

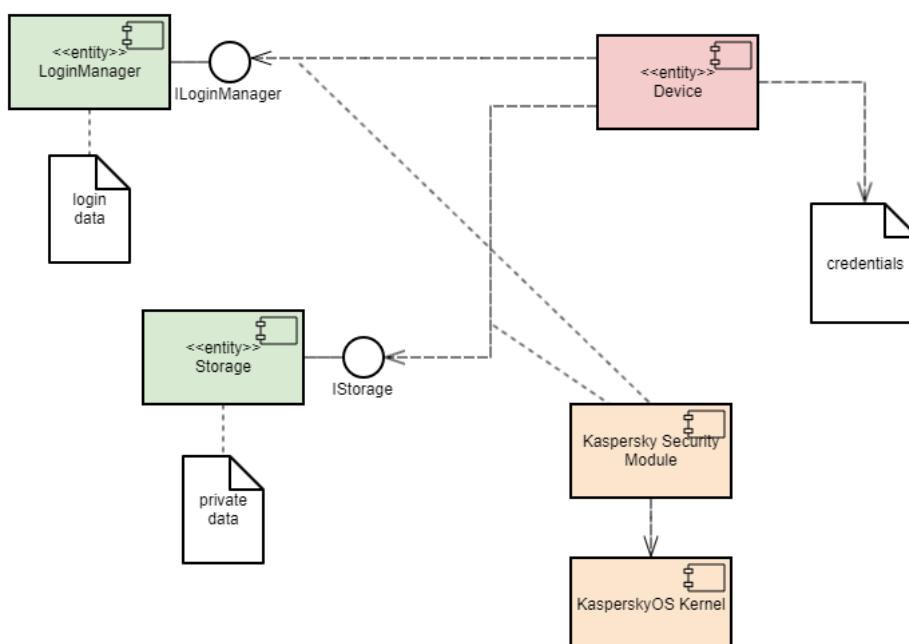
Пример `Device Access` демонстрирует использование паттерна `Privilege Separation`.

## Архитектура примера

Пример содержит три сущности: `Device`, `LoginManager` и `Storage`.

В этом примере сущность `Device` обращается к сущности `Storage` для получения информации и к сущности `LoginManager` для авторизации.

Сущность `Device` получает доступ к сущности `Storage` только после успешной авторизации.



Пример демонстрирует возможность разделения логики авторизации и логики доступа к данным на независимые компоненты. Такое разделение гарантирует, что доступ к данным может быть открыт только после успешной авторизации. При этом контроль за тем, что авторизация была проведена и закончилась успешно, осуществляется модулем безопасности. Кроме этого, такая архитектура позволяет производить независимую разработку и тестирование логики авторизации и логики предоставления доступа к данным.

Политика безопасности в примере `Device Access` имеет следующие особенности:

- Сущность `Device` имеет возможность обращаться к сущности `LoginManager` для авторизации.
- Вызовы метода `GetInfo()` сущности `Storage` контролируются с помощью [политик безопасности класса flow](#) (модель конечного автомата):
  - Конечный автомат, описанный в конфигурации объекта `session`, имеет два состояния: `unauthenticated` и `authenticated`.
  - Исходное состояние – `unauthenticated`.
  - Разрешены переходы из `unauthenticated` в `authenticated` и обратно.
  - Объект `session` создается при запуске сущности `Device`.
  - При успешном вызове сущностью `Device` метода `Login()` сущности `LoginManager` состояние объекта `session` изменяется на `authenticated`.
  - При успешном вызове сущностью `Device` метода `Logout()` сущности `LoginManager` состояние объекта `session` изменяется на `unauthenticated`.
  - При вызове сущностью `Device` метода `GetInfo()` сущности `Storage` проверяется текущее состояние объекта `session`. Вызов разрешается, только если текущее состояние объекта – `authenticated`.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/device_access
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

# Паттерн Information Obscurity

## Описание

Цель паттерна `Information Obscurity` – шифрование конфиденциальных данных в небезопасных средах с целью защиты данных от кражи.



## Контекст

Этот паттерн следует использовать, когда данные часто передаются между частями системы и/или между системой и другими (внешними) системами.

## Проблема

Конфиденциальные данные могут передаваться через недоверенную среду как внутри одной системы (через недоверенные компоненты), так и между разными системами (через недоверенные сети). В случае компрометации этой среды конфиденциальные данные могут быть получены злоумышленником.

## Решение

Политики безопасности должны разделять данные по уровню конфиденциальности, чтобы определить, какие данные следует зашифровать и какие алгоритмы шифрования использовать. Поскольку шифрование и дешифрование могут занять много времени, лучше по возможности ограничить их использование. Паттерн **Information Obscurity** решает эту проблему за счет использования уровня конфиденциальности для определения того, что необходимо скрыть с помощью шифрования.

## Примеры реализации

Пример реализации паттерна **Information Obscurity**: [Пример Secure Login](#).

## Источники

Паттерн **Information Obscurity** подробно рассмотрен в следующих работах:

- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. <https://dc.etsu.edu/etd/1119>
- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006).

## Пример Secure Login

Пример **Secure Login** демонстрирует использование паттерна **Information Obscurity**. Пример демонстрирует возможность передачи критической для системы информации через недоверенную среду.

## Архитектура примера

В примере имитируется получение удаленного доступа к IoT-устройству посредством передачи этому устройству учетных данных пользователя (имени пользователя и пароля). Недоверенной средой внутри IoT-устройства является веб-сервер, который обслуживает запросы пользователей. Практика показывает, что такой веб-сервер является легко обнаруживаемым и зачастую успешно атакуемым, так как IoT-устройства не имеют встроенных средств защиты от проникновения и других атак. Кроме того, пользователи получают доступ к IoT-устройству через недоверенную сеть. Очевидно, что в таких условиях для защиты учетных данных пользователя от компрометации необходимо использовать криптографические алгоритмы.

С точки зрения архитектуры в таких системах можно выделить следующие субъекты:

- Источник данных: браузер пользователя.
- Точка коммуникации с устройством: веб-сервер.
- Подсистема обработки информации от пользователя: подсистема аутентификации.

При этом для использования криптографической защиты необходимо выполнить следующие шаги:

1. Обеспечить взаимодействие источника данных и устройства по протоколу HTTPS. Это позволит избежать "прослушивания" HTTP-трафика и атак типа MITM (man in the middle).
2. Выработать между источником данных и подсистемой обработки информации общий секрет.
3. Использовать этот секрет для шифрования информации на стороне источника данных и расшифровки на стороне подсистемы обработки информации. Это позволит избежать компрометации данных внутри устройства (в точке коммуникации).

Пример Secure Login включает следующие компоненты:

- Веб-сервер Civetweb (недоверенный компонент, сущность WebServer).
- Подсистему аутентификацию пользователей (доверенный компонент, сущность AuthService).
- TLS-терминатор (доверенный компонент, сущность TlsEntity). Этот компонент поддерживает транспортный механизм TLS (transport layer security). TLS-терминатор совместно с веб-сервером поддерживают протокол HTTPS на стороне устройства (веб-сервер взаимодействует с браузером через TLS-терминатор).

Процесс аутентификации пользователя происходит по следующей схеме:

1. Пользователь открывает в браузере страницу по адресу `https://localhost:1106` (при запуске примера на QEMU) или по адресу `https://<IP-адрес Raspberry Pi>:1106` (при запуске примера на Raspberry Pi 4 B). HTTP-трафик между браузером и TLS-терминатором будет передаваться в зашифрованном виде, а веб-сервер будет работать с открытым HTTP-трафиком. (В примере используется самоподписанный сертификат, поэтому большинство современных браузеров сообщит о незащищенности соединения. Нужно согласиться использовать незащищенное соединение, которое тем не менее будет зашифрованным).
2. Веб-сервер Civetweb, запущенный в сущности WebServer, отображает страницу `index.html`, содержащую приглашение к аутентификации.
3. Пользователь нажимает на кнопку `Log in`.
4. Сущность WebServer обращается к сущности AuthService по IPC для получения страницы, содержащей форму ввода имени пользователя и пароля.
5. Сущность AuthService выполняет следующие действия:
  - генерирует закрытый ключ, открытые параметры, а также вычисляет открытый ключ по алгоритму Диффи-Хеллмана;
  - создает страницу `auth.html` с формой ввода имени пользователя и пароля (код страницы содержит открытые параметры и открытый ключ);

- передает полученную страницу сущности `WebServer` по IPC.
6. Веб-сервер `Civetweb`, запущенный в сущности `WebServer`, отображает страницу `auth.html` с формой ввода имени пользователя и пароля.
  7. Пользователь заполняет форму и нажимает на кнопку `Submit` (корректные данные для аутентификации содержатся в файле `secure_login/auth_service/src/authservice.cpp`).
  8. Код страницы `auth.html`, который исполняется на стороне браузера, осуществляет следующие действия:
    - генерирует закрытый ключ, вычисляет открытый ключ и общий секретный ключ по алгоритму Диффи-Хеллмана;
    - выполняет шифрование пароля операцией `XOR` с использованием общего секретного ключа;
    - передает веб-серверу имя пользователя, зашифрованный пароль и открытый ключ.
  9. Сущность `WebServer` обращается к сущности `AuthService` по IPC для получения страницы, содержащей результат аутентификации, передавая имя пользователя, зашифрованный пароль и открытый ключ.
  10. Сущность `AuthService` выполняет следующие действия:
    - вычисляет общий секретный ключ по алгоритму Диффи-Хеллмана;
    - расшифровывает пароль с использованием общего секретного ключа;
    - возвращает страницу `result_err.html` или страницу `result_ok.html` в зависимости от результата аутентификации.
  11. Веб-сервер `Civetweb`, запущенный в сущности `WebServer`, отображает страницу `result_err.html` или страницу `result_ok.html`.

Таким образом, конфиденциальные данные передаются через сеть и веб-сервер только в зашифрованном виде. Кроме того, весь HTTP-трафик передается через сеть в зашифрованном виде. Для передачи данных между компонентами используются взаимодействия по IPC, которые контролируются модулем `Kaspersky Security Module`.

## Unit-тестирование с использованием фреймворка GoogleTest

Помимо паттерна [Information Obscurity](#) пример `Secure Login` демонстрирует использование фреймворка `GoogleTest` для выполнения unit-тестирования программ, разработанных под `KasperskyOS` (`KasperskyOS Community Edition` содержит в своем составе этот фреймворк).

Исходный код тестов находится по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/secure_login/tests
```

Эти unit-тесты предназначены для верификации некоторых `cpp`-модулей подсистемы аутентификации и веб-сервера.

Чтобы запустить тестирование, выполните следующие действия:

1. Перейдите в директорию с примером `Secure Login`.

2. Удалите директорию `build` с результатами предыдущей сборки, выполнив команду:

```
sudo rm -rf build/
```

3. Откройте файл скрипта `cross-build.sh` в текстовом редакторе.

4. Добавьте в скрипт флаг сборки `-D RUN_TESTS="y" \` (например, после флага сборки `-D CMAKE_BUILD_TYPE:String=Release \`).

5. Сохраните файл скрипта, а затем выполните команду:

```
$ sudo ./cross-build.sh
```

Тесты выполняются в сущности `TestEntity`. Сущности `AuthService` и `WebServer` не запускаются, поэтому при выполнении тестирования пример нельзя использовать для демонстрации паттерна `Information Obscurity`.

После завершения тестирования выводятся результаты выполнения тестов.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/secure_login
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Приложения

Этот раздел содержит информацию, которая дополняет основной текст документа.

## Дополнительные примеры

Этот раздел содержит описания дополнительных примеров, входящих в состав KasperskyOS Community Edition.

### Пример net\_with\_separate\_vfs

Пример представляет собой простейший случай взаимодействия по сети с использованием сокетов Беркли.

Пример состоит из сущностей `Client` и `Server`, связанных TCP-сокетом с использованием loopback-интерфейса. В коде сущностей используются стандартные POSIX-функции.

Чтобы соединить сущности сокетом через loopback, они должны использовать один экземпляр сетевого стека, то есть взаимодействовать с "общей" [сущностью VFS](#) (в этом примере сущность называется `NetVfs`).

Для корректного соединения сущностей `Client` и `Server` с сущностью `NetVfs` необходимо также включить в решение [сущность Env](#).

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

### Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/net_with_separate_vfs
```

### Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

### Пример net2\_with\_separate\_vfs

Пример демонстрирует особенности решения, в котором сущность использует стандартные функции POSIX для взаимодействия с внешним сервером.

Пример `net2_with_separate_vfs` является видоизмененным примером [net\\_with\\_separate\\_vfs](#). В отличие от примера `net_with_separate_vfs`, в этом примере сущность взаимодействует по сети не с другой сущностью, а с внешним сервером.

Пример состоит из сущности `Client`, запущенной в KasperskyOS под QEMU, и программы `Server`, запущенной в хостовой операционной системе Linux. Сущность `Client` и процесс `Server` связаны TCP-сокетом. В коде сущности `Client` используются стандартные функции POSIX.

Чтобы соединить сущность `Client` и процесс `Server` сокетом, сущность `Client` должна взаимодействовать с сущностью `NetVfs`. Сущность `NetVfs` при сборке компонуется с сетевым драйвером, который обеспечит взаимодействие с процессом `Server`, запущенным в Linux.

Для корректного соединения сущности `Client` с сущностью `NetVfs` необходимо также включить в решение [сущность `Env`](#).

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/net2_with_separate_vfs
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Пример `embedded_vfs`

Пример показывает, как встроить [виртуальную файловую систему](#) (далее VFS), поставляемую в составе KasperskyOS Community Edition, в разрабатываемую сущность.

В этом примере сущность `Client` полностью инкапсулирует реализацию VFS из KasperskyOS Community Edition. Это позволяет избавиться от использования IPC для всех стандартных функций ввода-вывода (`stdio.h`, `socket.h` и так далее), например, для отладки или повышения производительности.

Сущность `Client` тестирует следующие операции:

- создание директории;
- создание и удаление файла;
- чтение из файла и запись в файл.

## Поставляемые ресурсы

В пример входит образ жесткого диска с файловой системой FAT32 – `hdd.img`.

Этот пример не содержит реализации драйверов блочных устройств, с которыми работает `Client`. Эти драйверы (сущности ATA и SDCard) поставляются в составе KasperskyOS Community Edition и добавляются в файл сборки `./CMakeLists.txt`.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/embedded_vfs
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Пример embed\_ext2\_with\_separate\_vfs

Пример показывает, как встроить новую файловую систему в [виртуальную файловую систему](#) (VFS), поставляемую в составе KasperskyOS Community Edition.

В этом примере сущность Client тестирует работу файловых систем (ext2, ext3, ext4) на блочных устройствах. Для этого Client обращается по IPC к драйверу файловой системы (сущности FileVfs), а FileVfs в свою очередь обращается по IPC к блочному устройству.

Файловые системы ext2 и ext3 работают с настройками по умолчанию. Файловая система ext4 работает, если отключить extent (`mkfs.ext4 -O ^64bit,^extent /dev/foo`).

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/embed_ext2_with_separate_vfs
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Подготовка SD-карты для запуска на Raspberry Pi 4 B

Для запуска примера `embed_ext2_with_separate_vfs` на Raspberry Pi 4 B необходимо, чтобы SD-карта, помимо загрузочного раздела с образом решения, также содержала 3 дополнительных раздела с файловыми системами ext2, ext3 и ext4 соответственно.

## Пример multi\_vfs\_ntpd

Этот пример показывает как использовать ntp-сервис в KasperskyOS. Сущность `k1.Ntpd` поставляется в составе KasperskyOS Community Edition и представляет собой реализацию ntp-клиента, который в фоновом режиме получает параметры времени от внешних ntp-серверов и передает их ядру KasperskyOS.

Пример также демонстрирует использование разных [виртуальных файловых систем](#) (далее VFS) в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется сущность `VfsNet`.
- Для работы с файловой системой используются сущности `VfsRamfs` и `VfsSdCardFs`.

Сущность `Client` использует стандартные функции библиотеки `libc` для получения информации о времени, которые транслируются в обращения к сущностям VFS по IPC.

[Сущность `Env`](#) используется для передачи переменных окружения и аргументов функции `main` другим сущностям.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

## Поставляемые ресурсы

В пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

Для каждого бэкенда в решении также используется отдельная сущность VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `./resources/sdcard/etc` содержат файлы конфигурации для сущностей VFS и `Ntpd`.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/multi_vfs_ntpd
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Пример `multi_vfs_dns_client`

Этот пример показывает как использовать внешний dns-сервис в KasperskyOS.

Пример также демонстрирует использование разных [виртуальных файловых систем](#) (далее VFS) в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется сущность `VfsNet`.
- Для работы с файловой системой используются сущности `VfsRamfs` и `VfsSdCardFs`.



Сущность `Client` использует стандартные функции библиотеки `libc` для обращения ко внешнему `dns`-сервису, которые транслируются в обращения к сущности `VfsNet` по IPC.

Сущность `Env` используется для передачи переменных окружения и аргументов функции `main` другим сущностям.

Для сборки и запуска примера используется система `CMake` из состава `KasperskyOS Community Edition`.

## Поставляемые ресурсы

В пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

Для каждого бэкенда в решении также используется отдельная сущность VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `./resources/sdcard/etc` содержат файлы конфигурации для сущностей VFS.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/multi_vfs_dns_client
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Пример `multi_vfs_dhcpd`

Пример использования сущности `k1.Dhcpd`.

Сущность `Dhcpd` представляет собой реализацию DHCP-клиента, который в фоновом режиме получает параметры сетевых интерфейсов от внешнего DHCP-сервера и передает их сущности виртуальной файловой системы (далее VFS).

Пример также демонстрирует [использование разных VFS](#) в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется сущность `VfsNet`.
- Для работы с файловой системой используются сущности `VfsRamfs` и `VfsSdCardFs`.

Сущность `Client` использует стандартные функции библиотеки `libc` для получения информации о сетевых интерфейсах (`ioctl`), которые транслируются в обращения к сущности VFS по IPC.

[Сущность Env](#) используется для передачи переменных окружения и аргументов функции main другим сущностям.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

## Поставляемые ресурсы

В пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

Для каждого бэкенда в решении также используется отдельная сущность VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `./resources/sdcard/etc` содержат файлы конфигурации для сущностей VFS и `Dhcpd`.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/multi_vfs_dhcpd
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Пример mqtt\_publisher

Пример использования протокола MQTT в KasperskyOS.

В этом примере MQTT-подписчик должен быть запущен в хостовой операционной системе, а MQTT-издатель в KasperskyOS. Сущность `Publisher` представляет собой реализацию MQTT-издателя, который публикует текущее время с интервалом 5 секунд.

В результате успешного запуска и работы примера MQTT-подписчик, запущенный в хостовой операционной системе, выведет сообщение `"received PUBLISH"` с топиком `"datetime"`.

Пример также демонстрирует использование разных [виртуальных файловых систем](#) (далее VFS) в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется сущность `VfsNet`.
- Для работы с файловой системой используются сущности `VfsRamfs` и `VfsSdCardFs`.

[Сущность Env](#) используется для передачи переменных окружения и аргументов функции main другим сущностям.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

## Запуск Mosquitto

Для запуска этого примера MQTT брокер Mosquitto должен быть установлен и запущен в хостовой системе. Для установки и запуска Mosquitto выполните следующие команды:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

Для запуска MQTT-подписчика в хостовой системе выполните следующую команду:

```
$ mosquitto_sub -d -t "datetime"
```

## Поставляемые ресурсы

В пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

Для каждого бэкенда в решении также используется отдельная сущность VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `./resources/sdcard/etc` содержат файлы конфигурации для сущностей VFS и `Ntpd`.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/mqtt_publisher
```

## Сборка и запуск примера

См. ["Сборка и запуск примеров"](#).

## Пример mqtt\_subscriber

Пример использования протокола MQTT в KasperskyOS.

В этом примере MQTT-издатель должен быть запущен в хостовой операционной системе, а MQTT-подписчик в KasperskyOS. Сущность `Subscriber` представляет собой реализацию MQTT-подписчика.

В результате успешного запуска и работы примера MQTT-подписчик, запущенный в KasperskyOS, выведет сообщение `"Got message with topic: my/awesome/topic, payload: hello"`.

Пример также демонстрирует использование разных [виртуальных файловых систем](#) (далее VFS) в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется сущность `VfsNet`.
- Для работы с файловой системой используются сущности `VfsRamfs` и `VfsSdCardFs`.

[Сущность `Env`](#) используется для передачи переменных окружения и аргументов функции `main` другим сущностям.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

## Запуск Mosquitto

Для запуска этого примера MQTT брокер Mosquitto должен быть установлен и запущен в хостовой системе. Для установки и запуска Mosquitto выполните следующие команды:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

Для запуска MQTT-издателя в хостовой системе выполните следующую команду:

```
$ mosquitto_pub -t "my/awesome/topic" -m "hello"
```

## Поставляемые ресурсы

В пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

Для каждого бэкенда в решении также используется отдельная сущность VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `./resources/sdcard/etc` содержат файлы конфигурации для сущностей VFS и `Ntpd`.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/mqtt_subscriber
```

## Сборка и запуск примера

См. ["Сборка и запуск примеров"](#).

## Пример gpio\_input

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность ввода GPIO пинов. Используется порт "gpio0". Все пины, кроме указанных в массиве `exceptionPinArr`, по умолчанию ориентированы на ввод, напряжение на пинах согласуется с состоянием регистров подтягивающих резисторов. Состояния всех пинов, начиная с GPIO0 (с учетом указанных в массиве `exceptionPinArr`), будут последовательно считаны, сообщения о состояниях пинов будут выведены в консоль. Задержка между считываниями смежных пинов определяется макроопределением `DELAY_S` (время указывается в секундах).

`exceptionPinArr` - массив номеров GPIO пинов, которые необходимо исключить из примера. Это может понадобиться в случае, если часть пинов уже задействована для других функций, например, если пины используются для UART соединения при отладке.

При [сборке и запуске этого примера на QEMU](#) возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При [сборке и запуске этого примера на Raspberry Pi 4 B](#) ошибка не возникает.

### Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/gpio_input
```

### Сборка и запуск примера

См. ["Сборка и запуск примеров"](#).

## Пример gpio\_output

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность вывода GPIO пинов. Используется порт "gpio0". Начальное состояние всех пинов GPIO должно соответствовать логическому нулю (напряжение на пине отсутствует). Все пины, кроме указанных в массиве `exceptionPinArr`, будут настроены на вывод. Каждый пин, начиная с GPIO0 (с учетом указанных в массиве `exceptionPinArr`), будет последовательно переведен в состояние логической единицы (появление на пине напряжения), а затем в состояние логического нуля. Задержка между изменениями состояния пинов определяется макроопределением `DELAY_S` (время указывается в секундах). Включение/выключение пинов производится от GPIO0 до GPIO27 и обратно до GPIO0.

`exceptionPinArr` - массив номеров GPIO пинов, которые необходимо исключить из примера. Это может понадобиться в случае, если часть пинов уже задействована для других функций, например, если пины используются для UART соединения при отладке.

При [сборке и запуске этого примера на QEMU](#) возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При [сборке и запуске этого примера на Raspberry Pi 4 B](#) ошибка не возникает.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/gpio_output
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Пример gpio\_interrupt

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность прерываний для GPIO пинов. Используется порт "gpio0". В битовой маске `pinsBitmap` структуры контекста прерываний `CallbackContext` пины из массива `exceptionPinArr` помечаются отработавшими, чтобы в дальнейшем пример мог корректно завершиться. Все пины, кроме указанных в массиве `exceptionPinArr`, переводятся в состояние `PINS_MODE`. Для всех пинов, кроме указанных в массиве `exceptionPinArr`, будет зарегистрирована функция обработки прерывания.

В бесконечном цикле происходит проверка условия равенства битовой маски `pinsBitmap` из структуры контекста прерываний `CallbackContext` битовой маске окончания работы примера `DONE_BITMASK` (соответствует условию, когда прерывание произошло на каждом GPIO пине). Также в цикле снимается функция-обработчик для последнего пина, на котором произошла обработка прерывания. При возникновении в первый раз прерывания на пине вызывается функция-обработчик, которая помечает соответствующий пин в битовой маске `pinsBitmap` в структуре контекста прерываний `CallbackContext`. Функция-обработчик для этого пина в дальнейшем снимается.

Следует учитывать возможное влияние начального состояния регистров подтягивающих резисторов для каждого пина на работу примера.

Прерывания для событий `GPIO_EVENT_LOW_LEVEL` и `GPIO_EVENT_HIGH_LEVEL` не поддерживаются.

`exceptionPinArr` - массив номеров GPIO пинов, которые необходимо исключить из примера. Это может понадобиться в случае, если часть пинов уже задействована для других функций, например, если пины используются для UART соединения при отладке.

При [сборке и запуске этого примера на QEMU](#) возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При [сборке и запуске этого примера на Raspberry Pi 4 B](#) ошибка не возникает.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/gpio_interrupt
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Пример gpio\_echo

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность ввода/вывода GPIO пинов, а также работу GPIO прерываний. Используется порт "gpio0". Пин вывода (GPIO\_PIN\_OUT) следует соединить с пином ввода (GPIO\_PIN\_IN). Устанавливается конфигурация для пина вывода (номер пина определяется в макросе GPIO\_PIN\_OUT), а также для пина ввода (GPIO\_PIN\_IN). Конфигурация пина ввода указана в макросе IN\_MODE. Регистрируется обработчик прерываний для пина ввода. Несколько раз изменяется состояние пина вывода. В случае корректной работы примера, при изменении состояния пина вывода должен вызываться обработчик прерываний, который выводит состояние пина ввода, при этом состояния пина вывода и пина ввода должны совпадать.

При [сборке и запуске этого примера на QEMU](#) возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При [сборке и запуске этого примера на Raspberry Pi 4 B](#) ошибка не возникает.

## Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/gpio_echo
```

## Сборка и запуск примера

См. "[Сборка и запуск примеров](#)".

## Лицензирование программы

Условия использования программы изложены в лицензионном договоре или подобном документе, на основании которого используется программа.



## Предоставление данных

KasperskyOS Community Edition не запрашивает, не хранит и не обрабатывает никакую персональную информацию, а также никакую другую информацию, не относящуюся к персональным данным.

## Информация о стороннем коде

Информация о стороннем коде содержится в файле `legal_notices.txt`, расположенном в папке установки программы.

## Уведомления о товарных знаках

Зарегистрированные товарные знаки и знаки обслуживания являются собственностью их правообладателей.

Arm и Mbed – зарегистрированные товарные знаки или товарные знаки Arm Limited (или дочерних компаний) в США и/или других странах.

CentOS – товарный знак компании Red Hat, Inc.

Debian – зарегистрированный товарный знак Software in the Public Interest, Inc.

Eclipse Mosquitto – товарный знак Eclipse Foundation, Inc.

GoogleTest – товарный знак Google, Inc.

Intel и Core – товарные знаки Intel Corporation, зарегистрированные в Соединенных Штатах Америки и в других странах.

Linux – товарный знак Linus Torvalds, зарегистрированный в США и в других странах.

Raspberry Pi – товарный знак Raspberry Pi Foundation.

Ubuntu – зарегистрированный товарный знак Canonical Ltd.

Visual Studio, Windows – товарные знаки Microsoft Corporation, зарегистрированные в Соединенных Штатах Америки и в других странах.