Глава 4: Результат и анализ безопасности

4.1. Используемый стек программного обеспечения

4.1.1. Инструментарий

4.1.1.1. Среда разработки

Для кросс-компиляции компонентов Безопасного мира используется инструментарий RISC-V GNU, в частности, GCC версии 11.1.0 и Binutils 2.38.

Как сама Безопасная ОС, так и Доверенные приложения компилируются с флагами для архитектуры RISC-V.

В качестве основной системы для разработки рекомендуется использовать ОС на базе Linux (например, Ubuntu 20.04). Система сборки требует CMake версии 3.14 или выше. Отладка осуществляется с помощью GDB, который подключается к GDB серверу QEMU, что позволяет инспектировать состояние ядра Безопасной ОС на самом низком уровне.

4.1.2. Среда эмуляции

4.1.2.1. QEMU с поддержкой WorldGuard

Для работы используется надежная среда эмуляции на базе модифицированной версии QEMU. Такая среда предоставляет виртуальную платформу, симулирующую архитектуру RISC-V с расширением WorldGuard, что обеспечивает воспроизводимое тестирование без необходимости в физическом оборудовании.

Эмуляция основана на форке QEMU версии v9.1.50 riscv64-softmmu, в который были внесены патчи для реализации ISA-расширения WorldGuard. Эти модификации включают реализацию специфичных для WorldGuard регистров управления и состояния (CSR), а также логики контроля доступа к памяти на основе идентификатора мира (world ID).

4.1.2.2. Конфигурация QEMU

Система эмулируется с использованием машинной модели virt с 4 ядрами: харядро 0 эксклюзивно выделено для Безопасной ОС, а остальные - для Linux. Файл дерева устройств (DTB) определяет физическую карту памяти, используемых устройств.

QEMU запускается со следующими флагами:

```
qemu-system-riscv64 -M virt,wg=on -m 4G -nographic -smp 4 -bios sbi.bin
```

4.1.3. Linux

4.1.3.1. Linux с поддержкой WorldGuard

Стандартное ядро Linux выступает в роли операционной системы Обычного мира. В этой среде работают пользовательские приложения и драйвер ядра, отвечающий за инициацию взаимодействия с Безопасной ОС и её Доверенными приложениями.

В проекте используется ядро Linux, основанное на LTS-версии 6.12.0. Оно включает набор внешних патчей, которые в содержат драйвер символьного устройства, реализующий клиентскую часть протокола TEE. Этот драйвер предоставляет стандартный интерфейс на основе ioctl для взаимодействия пользовательских приложений с Безопасным миром.

4.1.3.2. Конфигурация Linux

Ядро собирается с минимальной конфигурацией на основе defconfig для архитектуры RISC-V. Ключевая включенная опция - это CONFIG_WG_TEE. Конфигурация нацелена на создание компактного образа ядра с минимальным набором драйверов, необходимых для запуска в среде QEMU.

4.1.3.3. Образ Linux

Загрузочный компонент Linux состоит из образа на базе дистрибутива debian 10. Файловая система создается примонтированном в QEMU диске.

4.1.4. Система сборки

4.1.4.1. Конфигурация CMake

Для компиляции всех программных компонентов проекта используется централизованная и модульная система сборки на основе CMake версии не менее 3.14.

4.1.4.2. Дизайн системы сборки

Система сборки организована с корневым файлом CMakeLists.txt, который подключает подкаталоги с ядром Безопасной ОС, стандартными библиотеками и Доверенными приложениями.

4.1.4.3. Процесс сборки Доверенного приложения (ДП)

Каждое Доверенное приложение собирается как отдельный позиционно-независимый ELF-файл. Процесс сборки ДП включает компиляцию его исходного кода с использованием стандартной библиотеки Безопасной ОС и обработку файла-манифеста. Манифест, определяющий UUID приложения и его начальные права доступа, встраивается в специальную секцию итогового ELF-файла, откуда он считывается Безопасной ОС при инициализации.

4.1.5. Интеграция с CI

4.1.5.1. Настройка непрерывной интеграции

Для обеспечения качества и стабильности кода используется конвейер непрерывной интеграции (CI), который автоматизирует процессы сборки и тестирования.

Автоматизация подтверждает, что вносимые изменения не приводят к регрессиям, и

гарантирует работоспособность всего программного стека.

Проект использует GitHub Actions в качестве платформы для CI. Рабочий процесс настроен так, что он запускается при каждой отправке изменений (push) в основные ветки. Он последовательно выполняет серию задач: сначала собирается кросс-компилятор, затем модифицированный QEMU, OpenSBI, Безопасная ОС со своими ДП и, наконец, ядро Linux.

4.1.5.2. Скрипты автоматизированного тестирования

На заключительном этапе конвейера CI выполняется автоматизированный тест, реализованный на Python. Этот скрипт запускает полностью собранную систему в QEMU с поддержкой WorldGuard, в Безопасной ОС при этом запускаются тестовые сценарии. Таким образом проверяется, что и Безопасная ОС, и Linux успешно загрузились, а также подтверждается корректная инициализация драйвера связи и выполнение тестовой команды в образцовом Доверенном приложении.

4.2 Демонстрация работы Безопасной ОС

В данном разделе демонстрируется практическое применение Безопасной ОС на примере полного цикла: от сборки всего программного стека и разработки минимального Доверенного приложения (ДП) до его запуска в эмулируемой среде RISC-V. Цель - проверить сквозную интеграцию всех компонентов системы, от клиентского приложения в Обычном мире (Normal World) до самого ДП, работающего в Безопасном мире (Secure World).

4.2.1. Сборка программного стека

В этом подразделе приводится краткое руководство по сборке всех необходимых программных компонентов из исходного кода для получения загрузочного образа системы, готового к запуску в среде эмуляции QEMU.

Клонирование репозиториев проекта

Первым шагом является клонирование исходного кода всех ключевых компонентов: модифицированного QEMU с поддержкой WorldGuard, адаптированной прошивки OpenSBI, ядра Linux с драйвером поддержки TEE, а также Безопасной ОС вместе с её библиотеками и примерами ДП.

клонирование производится командой

```
git clone git@github.com:HALIP192/TEEgRISCv.git
cd TEEgRISCv
git submodule update --init --recursive
```

Сборка инструментария для кросс-компиляции

Для компиляции всех артефактов требуется инструментарий RISC-V GNU, который обеспечивает двоичную совместимость (ABI) между компонентами. Процесс включает настройку кросс-компилятора GCC и утилит для целевой архитектуры riscv64.

Для начала необходимо установить следующие пакеты:

```
sudo apt install -y \
 git                    \
 make                   \
 cmake                  \
 bear                   \
 device-tree-compiler   \
 cpio                   \
 libncurses-dev         \
 gawk                   \
 flex                   \
 bison                  \
 openssl                \
 libssl-dev             \
 dkms                   \
 libelf-dev             \
 libudev-dev            \
 libpci-dev             \
 libiberty-dev          \
 autoconf               \
 bc                     \
 python3                \
 python3-venv           \
 ninja-build            \
 libglib2.0-dev         \
 libgcrypt20-dev        \
 zlib1g-dev             \
 autoconf               \
 automake               \
 libtool                \
 libpixman-1-dev        \
 libslirp-dev
```

Загрузка тулчейна:

```
mkdir -p toolchain;
cd toolchain;
wget $TOOLCHAIN_URL --output-document "sc-dt.tar.gz";
tar xf "sc-dt.tar.gz";
```

Сборка QEMU с поддержкой WorldGuard

Среда эмуляции создаётся путём сборки реализации QEMU с расширением WorldGuard. Эмулятор конфигурируется для машины virt с riscv64-softmmu, что активирует симуляцию аппаратных механизмов, необходимых для разделения на два мира.

сборка QEMU производится командой

```
mkdir -p build;
cd build;

../configure \
 --target-list=riscv64-softmmu \
 --disable-capstone \
```

```
 --disable-curl \
 --disable-guest-agent \
 --disable-libusb \
 --disable-vnc \
 --enable-slirp \
 --disable-docs;

make -j $(nproc)
```

Сборка модифицированного OpenSBI

Прошивка OpenSBI компилируется из репозитория, содержащего изменения стандартного процесса загрузки. Эти изменения позволяют OpenSBI инициализировать Безопасную ОС на первом ядре процессора перед передачей управления ядру Linux на остальных ядрах, тем самым устанавливая аппаратное разделение миров.

сборка OpenSBI производится командой

```
mkdir -p build;
cd build;

make \
  -C ../ \
  PLATFORM=generic \
  FW_PAYLOAD_PATH=kernel.bin \
  NWD_FW_PAYLOAD_PATH=LinuxImage \
  -j $(nproc)
```

Сборка образа Linux с поддержкой WorldGuard

Ядро Linux конфигурируется с опцией CONFIG_WG_TEE, которая включает драйвер для взаимодействия с Безопасной ОС. Этот драйвер предоставляет стандартный интерфейс на основе системного вызова ioctl, позволяя приложениям Обычного мира обращаться к TEE.

сборка Linux производится командой:

```
mkdir -p build;
cd build;

make \
 -C ../ \
 O=build \
 ARCH=riscv \
 LLVM=1 \
 teegriscv_defconfig \

make \
 -C ../ \
 O=build \
 ARCH=riscv \
 LLVM=1 \
 -j 16
```

Сборка Безопасной ОС

Сборка Безопасной ОС выполняется с помощью системы CMake. В ходе этого процесса компилируется микроядро, его стандартные библиотеки, а манифесты и ELF-файлы предопределённых Доверенных приложений упаковываются в конечный образ Безопасной ОС в соответствии с картой памяти платформы.

```
build.py \
 -manifest manifest.toml \
 -build build \
 -root blowfish

cmake -B build blowfish
cd build
make
```

Сборка загрузочного образа Linux

Загрузочный образ Linux создаётся с помощью скрипта create_image.sh поверх дистрибутива debian 10:

```
./create_image.sh examples/hello_world
```

Пример доверенного приложения

Реализация простого доверенного приложения

Для иллюстрации процесса разработки ДП создаётся базовое приложение, которое предоставляет Обычному миру простую функцию генерации случайных чисел.

Определение манифеста для модели полномочий

Для ДП создаётся файл манифеста, в котором указываются его уникальный идентификатор (UUID) и минимально необходимый набор полномочий. Затем этот манифест регистрируется в манифесте корневой задачи Безопасной ОС, что даёт ей право на запуск данного ДП.

пример манифеста:

```
uuid = "8aaaf200-2450-11e4-abe2-0002a5d5c51b"
name = "hello_world"

[kernel]
factory = ["create_vmo"]
vmspace = ["allocate"]

crypto = []
```

Сборка доверенного приложения

ДП компилируется в позиционно-независимый ELF-файл. Его манифест встраивается в специальную секцию этого файла, что позволяет Безопасной ОС при загрузке извлечь информацию о его идентификаторе и правах доступа:

```
build.py \
 -manifest hello.toml \
 -build build \
 -root blowfish
```

Демонстрация выполнения

В этом подразделе описаны шаги по запуску полностью собранной системы и показан полный цикл взаимодействия между клиентским приложением в Linux и примером ДП.

Загрузка системы

Система запускается с помощью QEMU с собранным загрузочным образом. Консольный вывод подтверждает успешную передачу управления от OpenSBI к Безопасной ОС и последующей загрузки ядра Linux:

```
OpenSBI v1.6
   _____                    ______ _____ ______
  / ___ \                  / _____|  _ \_   _|
 | |   | |_ __   ___ _ __ | (____ | |_) || |
 | |   | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |___| | |_) |  __/ | | |____) | |_) || |_
  \_____/| .__/ \___|_| |_|_____/|____/_____|
         | |
         |_|

Platform Name               : riscv-virtio,qemu

Switching... 80200000, wid 0
[BOOT] Blowfish start..
[SBI TIMER] Timer frequency is 0000000000989680
[PAGE_ARRAY] Page array base 0xffffffe0000ce000 size 00000000000004ce
[WGC] DRAM: Vendor ID 00000000
[WGC] DRAM: Num slots 00000010
[WGC] UART: Vendor ID 00000000
[WGC] UART: Num slots 00000001
[CONSOLE] Unregistering console Sbi console
[CONSOLE] Registering console Uart console
[TASK] System task initialized
[NWD] Registered NWD channel at pa 00000000802cb000
[ROOTTASK] Spawned root task
[BOOT] Blowfish boot end...
Booting NWD payload on CPU 1... 80400000
Switching... 80400000, wid 1
```

Инициализация драйвера Linux

После загрузки Linux его модуль ядра для взаимодействия с TEE загружается командой insmod. Наличие устройства tee0 подтверждает успешную инициализацию драйвера TEE:

```
# ls /dev/tee*
/dev/tee0
```

Открытие сессии с доверенным приложением

Клиентское приложение в Linux использует TEE Client API для вызова `TEEC_OpenSession`, передавая UUID арифметического ДП. Это действие инициирует запрос к Безопасной ОС, которая проверяет его и создаёт новый экземпляр сессии для указанного ДП.

Пример кода, открывающий сессию с ДП:

```
TEEC_Result res;
TEEC_Session session;
TEEC_UUID uuid = TA_HELLO_WORLD_UUID;
uint32_t ret;

res = TEEC_InitializeContext("test", &ctx);
if (res != 0) {
        printf("Failed to open context %d\n", res);
        return 0;
}

res = TEEC_OpenSession(&ctx, &session, &uuid, 0, NULL, NULL, &ret);
if (res != 0) {
        printf("Failed to open seesion %d\n", res);
        return 0;
}

...
```

Вызов функции ДП и получение ответа

Далее клиентское приложение вызывает `TEEC_InvokeCommand`, передавая в качестве параметров целое число - seed генератора случайных чисел.

Запрос передаётся в ДП через очередь запросов, ДП выполняет вычисление случайного числа, и результат возвращается клиенту через очередь ответов.

Пример кода, вызывающего функцию ДП:

```
TEEC_Result res;
uint32_t ret;
TEEC_Operation op = {};

...

printf("Before call value %x\n", 0xdeadbeef);
op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INOUT, TEEC_NONE, TEEC_NONE
                                 TEEC_NONE);
op.params[0].value.a = 0xdeadbeef;

res = TEEC_InvokeCommand(&session, TA_INC_ARG, &op, &ret);
if (res != 0) {
        printf("Failed to invoke command %d\n", res);
        return 0;
```

```
}

printf("After call value %x\n", op.params[0].value.a);
```

Демонстрация работы с разделяемой памятью

ДП так же имеет возможность работать с разделяемой памятью, которая доступна для чтения и записи из ДП. Для создания участка разделяемой памяти вызывается TEEC_AllocateSharedMemory, который возвращает указатель на участок памяти.

Пример кода, создающего участок разделяемой памяти:

```
TEEC_Result res;
uint32_t ret;
TEEC_Operation op = {};
TEEC_SharedMemory shmem = {};

res = TEEC_AllocateSharedMemory(&ctx, &shmem);
if (res != 0) {
        printf("Failed to allocate shmem %d\n", res);
        return 0;
}

void *data = mmap(NULL, 1 << 12, PROT_READ | PROT_WRITE,
                  MAP_SHARED, shmem.fd, 0);

assert(data != MAP_FAILED);

*(unsigned *)data = 0x12345;
printf("Shmem before call %x\n", *(unsigned *) data);

op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INOUT, TEEC_NONE,
                                 TEEC_NONE, TEEC_NONE);
op.params[0].memref.parent = &shmem;
op.params[0].memref.size = 1 << 12;
op.params[0].memref.offset = 0;

res = TEEC_InvokeCommand(&session, TA_SHMEM, &op, &ret);
if (res != 0) {
        printf("Failed to invoke command %d\n", res);
        return 0;
}
```

Заключение

References

Appendices

Sample TA Code

TA Manifests

Secure OS Code

Secure Standard Library Code