SAMSUNG

Применение и сопровождение санитайзеров на крупномасшабной кодовой базе

Михаил Кашкаров

29 июня 2020 г.

Содержание

Introduction

1. **Introduction**
   - Tizen
   - Problem
2. Building sanitized OS
3. Running sanitized OS
4. Other problems
5. Results

Tizen

OS

Tizen это open source операционная система основанная на Linux kernel и GNU C Library имплементации Linux API.

Тулчейн

- GNU GCC
- GNU Glibc
- GNU Binutils

Package management

Tizen использует `rpm` в качестве пакетного менеджера и `OBS` для билда.

К чему применимо

Tizen OS содержит большое количества кода на языке `C/C++` и может содержать ошибки, которые сложно обнаружить во время компиляции, но могут проявлятся в дальнейшем. Санитайзеры помогают найти такие проблемы во время работы приложений.

Задача

Address Sanitizer

AddressSanitizer (ASan) - открытый набор тулов от Google которые позволяют находить ошибки адресации памяти во время работы программ.

Задача

Background

Исследование применимости AddressSanitizer началось в SRR в 2013 г., после того, как технология была стабилизированна для полного применения.
Использование и применение в приложениях Tizen началось в 2015 г.
После успешного применения мы решили расширить идею на полную санитизацию прошивки, а не только на несколько ключевых приложений.

Задача

Project target

Собрать каждое приложение из Tizen с AddressSanitizer и наладить сборку прошивки, полностью повторяющую обычную сборку, но в которой всё санитизированно.

Задача

Дополнительно

- Проверить текущую кодовую базу и сообщить разработчикам о проблемах
- Предоставить инструменты для санитайзеров в используемом тулчейне
- Документирование по использованию

Building sanitized OS

1. Introduction
2. **Building sanitized OS**
   - Integration approach
   - Compiler
   - Build system
3. Running sanitized OS
4. Other problems
5. Results

Integration steps

1. Сборка GCC с поддержкой санитайзеров
2. Интеграция в общий OBS билд
3. Предотвратить постоянные ошибки сборки с санитайзерами
4. Предоставить кросс-сборку с санитайзерами
5. Запуск на девайсе
6. UI интерфейс

GCC Build

Introducing compiler feature

Сборка Tizen завязанна на определённом тулчейне, время для внесения изменения связанно с датами релизов.
Небольшие изменения возможны только если они не затрагивают сборку без санитайзеров.

Примененный подход

- Предоставить сборку инфраструктуры для санитайзеров
- Создать новый `rpm`-пакет с `libasan.so`
- Проверить тулчейн с и без включенных санитайзеров
- Предоставить полный билд базового проекта

GCC Build

GCC sanitization

Следует ли нам собирать сам компилятор с `–with-build-config=bootstrap-asan` и весь тулчейн с санитайзерами или нет?
Наше решение: оба варианта.

- Санитизированный тулчейн для внутреннего подготовительного проекта
- Не-санитизированный GCC для внешнего “release” проекта

OBS Integration

Build procedure

- Каждая сборка происходит в изолированном контейнере (`qemu-kvm`)
- Результаты сборки сохраняются в `rpm` пакет (или несколько пакетов)
- `rpm`'s пакеты объединяются в *проекты*
- Каждый проект настраивается независимо и имеет общие базовые настройки

OBS Integration

Naive approach

У проектов есть специальный макрос `Optflags` содержащий флаги компиляции, которые передаются в каджый пакет во время сборки.

Issues

- Не все пакеты собираются с помощью GNU Autotools или CMake
- Не все пакеты искользуют полученные опции из `Optflags`

OBS Integration

Working approach

Предоставить обёртку над компилятором с возможностью тонкой настройки которую мы назвали `gcc-force-options`

OBS Integration

ASan environment in container

Каждый контейнер для билда `rpm` пакеты создаётся перед началом сборки, поэтому настройка окружения для санитайзеров должна производится вместе с этой фазой. Как минимум, необходимы следующие шаги:

- Установка `libasan.so`
  С помощью глобальных настроек проекта и `Preinstall`
- Добавить `libasan.so` в `LD_PRELOAD`
  Сделано с помощью вспомогательного пакета с post-install скриптом
- Предоставить `ASAN_OPTIONS`
  Сделано с помощью модификации рантайма санитайзеров. Наш `libasan.so` получает опции из файла `/ASAN_OPTIONS`
- Сборка ASan логов после окончания билда
  Сделано с помощью дополнительных `rpm` скриптов

OBS Integration

ASan build influence

Запуск всех тулов с Address Sanitzer обычно приводят к 2 главным проблемам:

- Ошибки памяти, пойманные ASan с последующей остановкой
  Решено с помощью *recovery mode*
- `configure`/`cmake` ошибки из-за логов ASan
  Решено с помощью перенаправления вывода

Running sanitized OS

1. Introduction
2. Building sanitized OS
3. **Running sanitized OS**
   - Running on device
   - Binary size
   - RAM usage
   - UX improvements
4. Other problems
5. Results

Running on device

Goals

- **Short term**
  Загрузить девайс с полностью санитизированной прошивкой и убедиться, что всё работает.
  - Запустить каждое приложение на девайсе и убедиться, что оно работает правильно (Как минимум не появились проблемы из-за санитайзеров)
- **Long term**
  Наладить процесс регулярых санитизированных билдов и тестирования полученных прошивок
  - Автоматический запуск тестов для санитизированных проектов
  - Интегрировать ASan в систему верификации

Running on device

Challenges

- **Размер санитизированных прошивок**
  Итоговый размер санитизированных прошивок сильно больше обычных
- **Потребеление памяти**
  Санитизированные прошивки потребляют больше физической памяти чем обычные
- **Проблемы внутри ASan**
  На текущий момент ASan достаточно стабилен, но мы до сих пор натыкаемся на некоторые граничные случаи.

Sanitized image size

Firmware size

Размер санитизированных прошивок сильно больше обычных

- Исходный размер (compressed tarball): **327.6 MB**
- Санитизированный (compressed tarball): **456.4 MB**
- Разница: **128.8 MB (40%)**

Reason: package size bloating

Section	Regular (MB)	Sanitized (MB)	Difference (MB)	Difference (%)
`.text`	29.3	108.1	78.8	268%
`.rodata`	4.4	19.0	14.6	332%
`.bss/.data`	1.8	9.7	8.1	450%
...	...	...	...	...
Total:	**39**	**146**	**107**	**274%**

Таблица: Binary size comparison for **libchromium.so**

Image size reduction

Recipes

- Общие оптимизации на размер бинарников
  `CFLAGS+=″-Os″`
- Отключение инструментирования глобальных переменных
  `CFLAGS+=″–param asan-globals=0″`
- Не использовать inline-инструментации
  `CFLAGS+=″–param asan-instrumentation-with-call-threshold=0″`

Memory consumption

Memory overhead sources

- **Allocator quarantine**
  Настраивается с помощью `quarantine_size_mb` опции выполнения
- **ASan redzones**
  Можно уменьшить отключив инструментацию частей приложения (например, глобальные пременные)
- **ASan shadow**
  Может быть немного уменьшено более компактной раскладкой (например, 16:1)
- **ASan fake stacks**
  Может быть отключенно с помощью `stack-use-after-return` опции
- **Code and data bloating**
  Уменьшается с оптимизациями на размер кода (`-Os, –param asan-instrumentation-with-call-threshold=0`)
- **Allocator implementation**
  Аллокатор ASan’а настроен на быстродействие и расширяемость. Но можно сконфигурировать для меньшего потребления памяти взамен.

Memory consumption

Runtime memory issues

- **Reduce quarantine size to minimally possible value (1MB in our case)**
  В теории уменьшение этого значения приводит к потери части use-after-free багов, но на практике проблем замечено не было
- **Disable stack-use-after-return detection**
  Use-after-return режим очень затратен по памяти (до $\times 2$ дополнительного расхода памяти)
- **Tweak ASan allocator**
  - Модифицировав стандартный аллокатор ASan'а мы получили дополнительное уменьшение потребляемой памяти на **100MB**.
- **Enable swapping**
  - Позволяет запускать тяжеловесные приложения как, например, браузер
  - Сделало прошивку в целом стабильнее

UX improvements

Points 1/2

- Recovery mode
  - Изначально внутренний патч, upstreamed by Yuri Gribov
  - Позволяет анализировать больше багов за один цикл тестирования
  - Доступен с GCC $6+$
- Automatic `/proc` mounting
  - Потребовалось для санитизации `systemd`
- `print_cmdline` runtime option
  - Для отладки дочерних процессов

UX improvements

Points 2/2

- `libbacktrace` separate debuginfo support [2]
  - Существенно улучшает анализ для бинарников без debug info
  - Patch is upstreamed
- Reading `ASAN_OPTIONS` from file
  - Более гибкая настройка ASan в нашем окружении
- SMACK support
  - `setxattr(2)` ипользуется для проставления SMACK меток на ASan логи для удобства пользователей
- Всевозможные bugfixes: неверное выравнивание глобальных переменных, граничные случаи внутри libasan.so binary [1]

Resulting setup

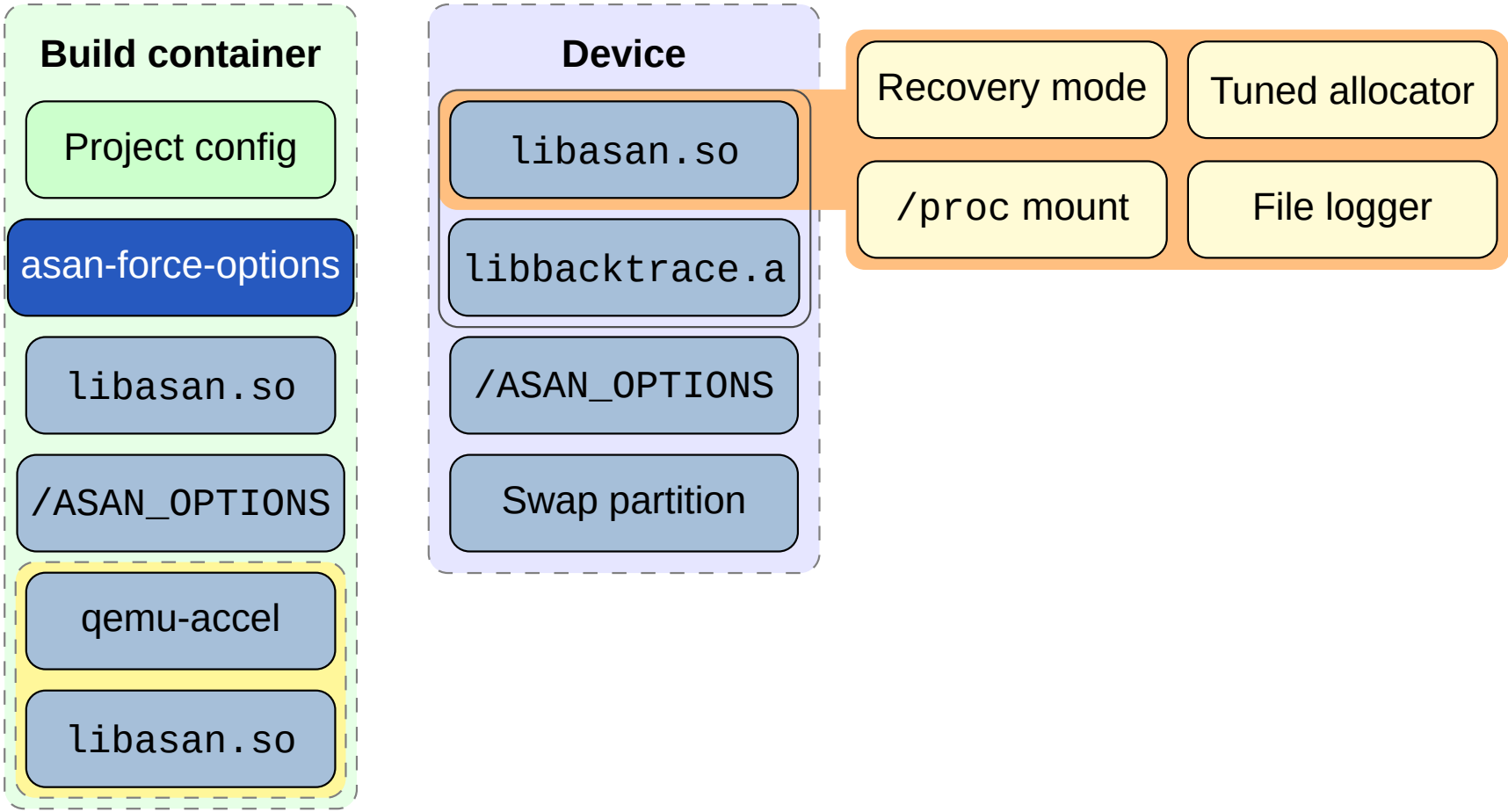


Рис.: Resulting ASan setup in Tizen

Other problems

Source code issues

Build issues

- Сборка статических пакетов
  Некоторые приложение должны быть собраны статически (например, внутренности `initrd`), которые требуют дополнительных действий для компиляции. Мы либо используем дополнительные шаги сборки либо используем несанитизированные версии в таких случаях.
- Поддержка тулов, например `patchelf`
  К сожалению, `patchelf` не идеален и иногда портит бинарники для arm билдов в сочетании с санитайзервами.
- Время пересборки
  Любой патч на `libsanitizer` приводит к ребилду компилятора, который провоцирует полную пересборку OS и занимает длительное время.

Source code issues

Open-Source code bugs

- Древние баги
  Иногда встречаются действительно древние баги, например в `bison` [3] в самых неожиданных местах
- Непредвиденные проблемы
  Всевозможные странных проблемы, например в `gzip`.

```
[gzip_cv_underline=yes
 AC_TRY_COMPILE([int foo() {return 0;}], [],
   [$NM conftest.$OBJEXT | grep _foo >/dev/null 2>&1 || #
     _GLOBAL__sub_I_00099_0_foo
      gzip_cv_underline=no])]
```

Integration issues

OS integration issues

- `systemd` timeout
  Замедление производительности с ASan и как следствие некоторые сервисы `systemd` завершает.
- `systemd` slice limits
  Большинство сервисов в `cgroup` и стандартные лимиты по времени слишком малы для санитизированных бинарников.

Integration issues

Corporate issues

- Company size
  Множество команд работает над операционной системой со своими задачами и требованиями, включая процесс разработки и искользуемые стредства.
- Rules and processes
  Существуют корпоративные ограничения, поэтому поддержка различных дивизионов и открытие исходного кода довольно затруднительна.

Results

Results

Short-term goals reached

После успешной первой сбоки с Address Sanitizer и загрузки прошивки моментально были найдены 12 багов:

- 1 **SEGV** type bug
- 2 **stack-buffer-overflow** type bugs
- 3 **heap-buffer-overflow** type bugs
- 1 **global-buffer-overflow** type bug
- 4 **heap-use-after-free** type bugs
- 1 **stack-use-after-return** type bug

И эти баги были получены только после запуска девайса и открытии нескольких приложений!
Разработчики получили найденные репорты об ошибках и исправили их.

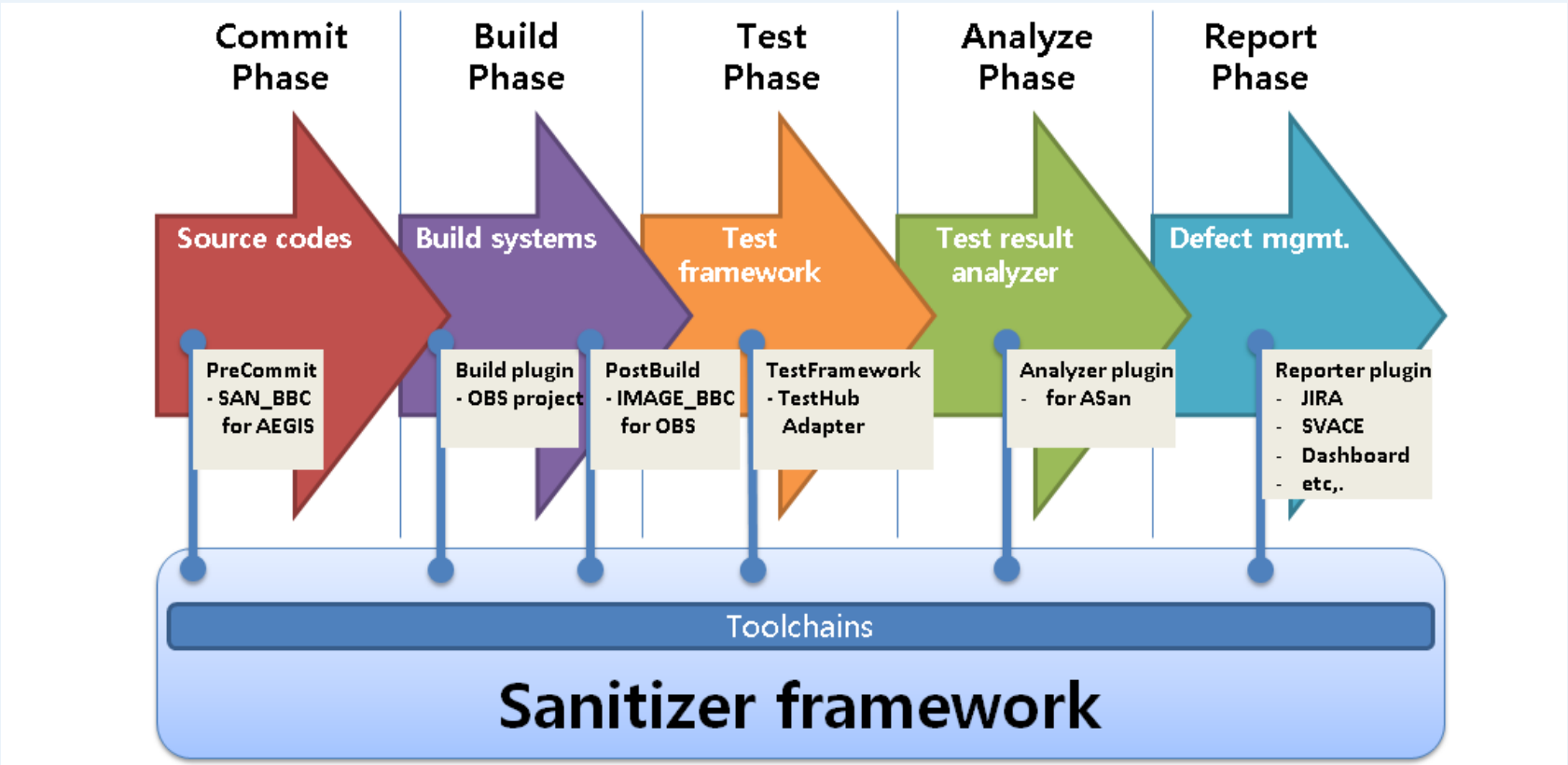
Results

Long-term goals reached

- Налажена регулярная сборка санитизированных прошивок
- Санитизированные прошивки регулярно проверяются QA командой
- Подготовленная инфарструктура позволила легко запустить остальные санитайзеры для регулярных проверок: `LSan`, `UBSan`, `TSan`, `ISan`

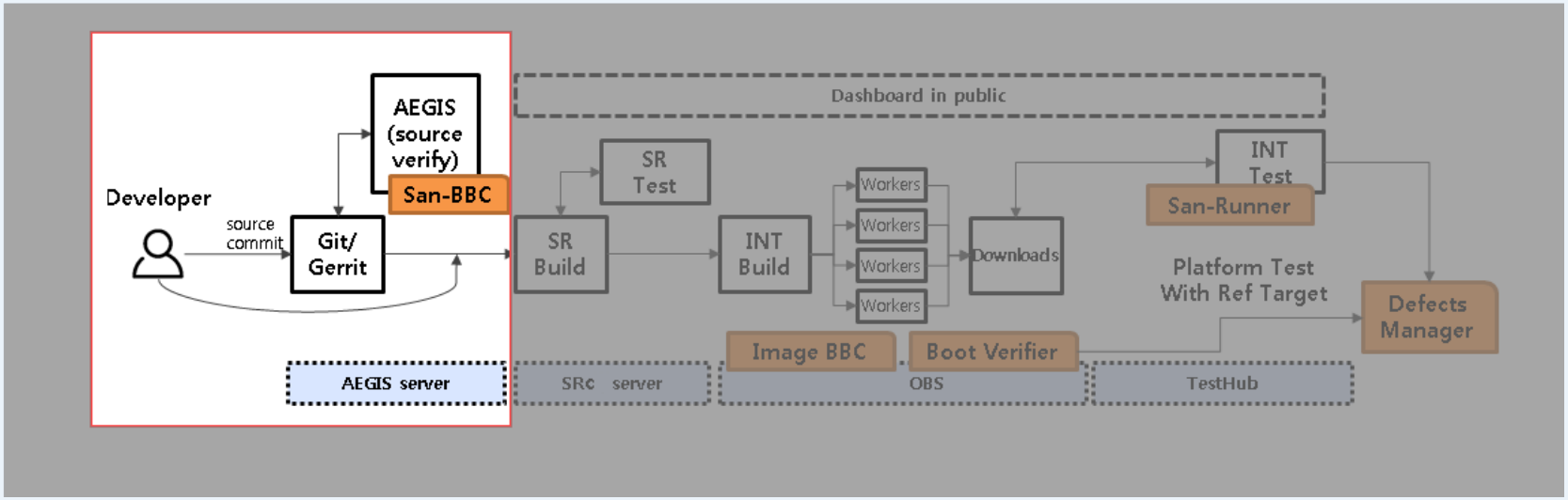
Integration with services

Overview



Integration with services

Overview

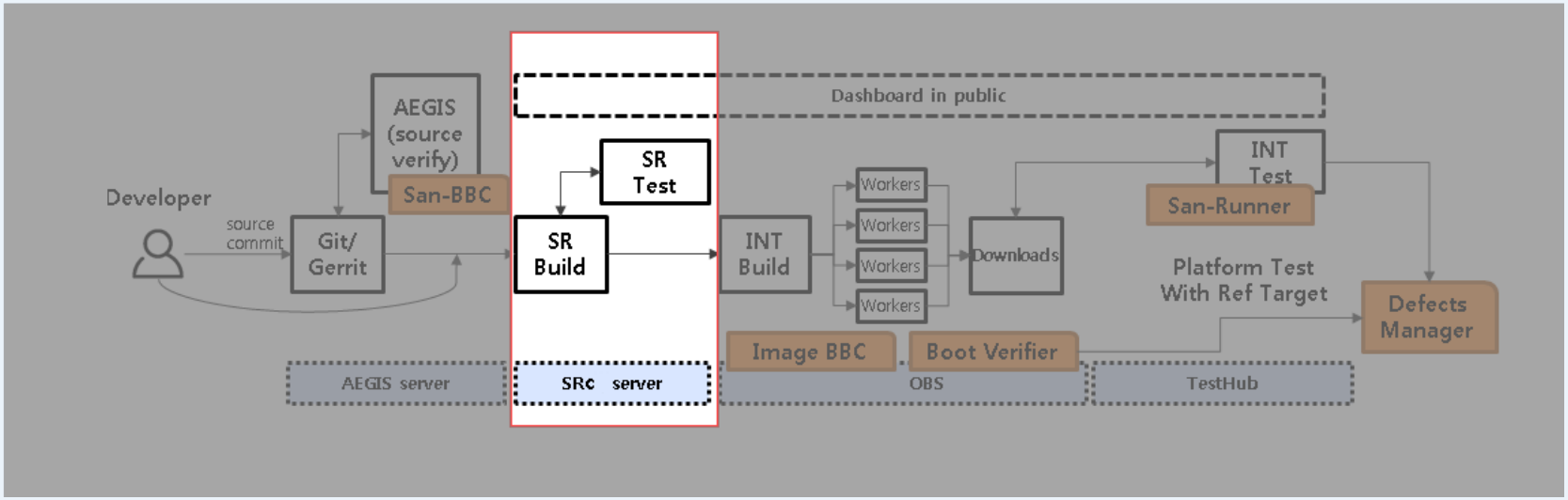


Process

- Build-break checker (BBC)
- Проверка сборки санитизированных приложений после новых изменений от разработчиков
- Уведомление о возникших ошибках на этапе сборки с санитайзерами

Integration with services

Overview

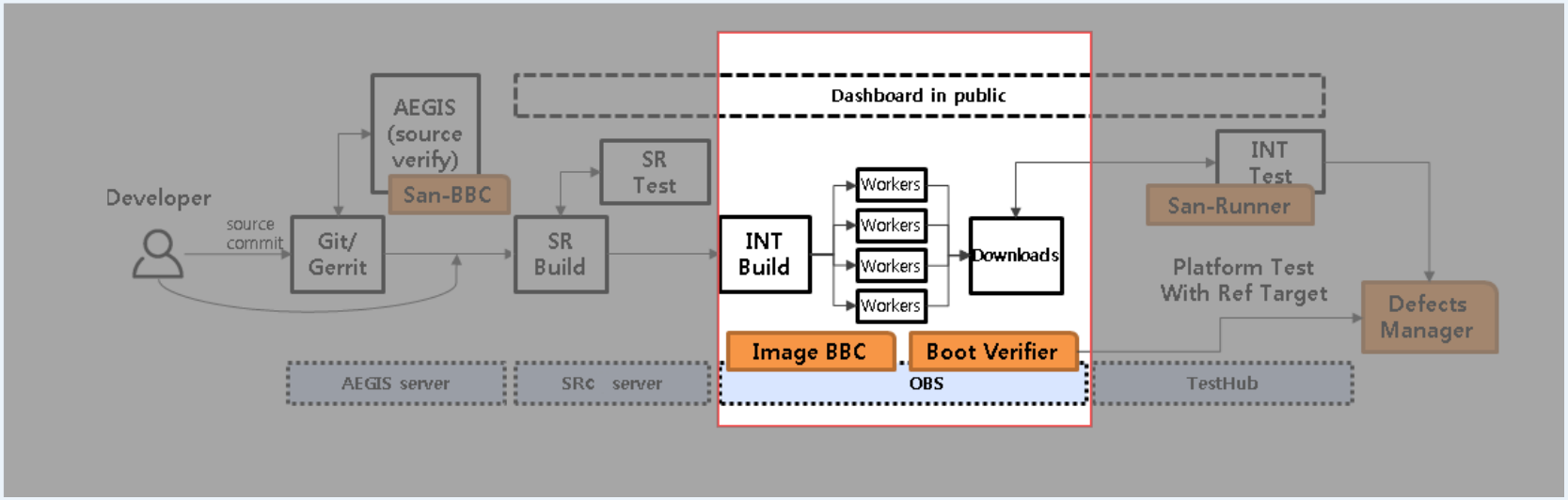


Process

- Сборка всех необходимых приложений с санитайзерами для создания прошивки

Integration with services

Overview

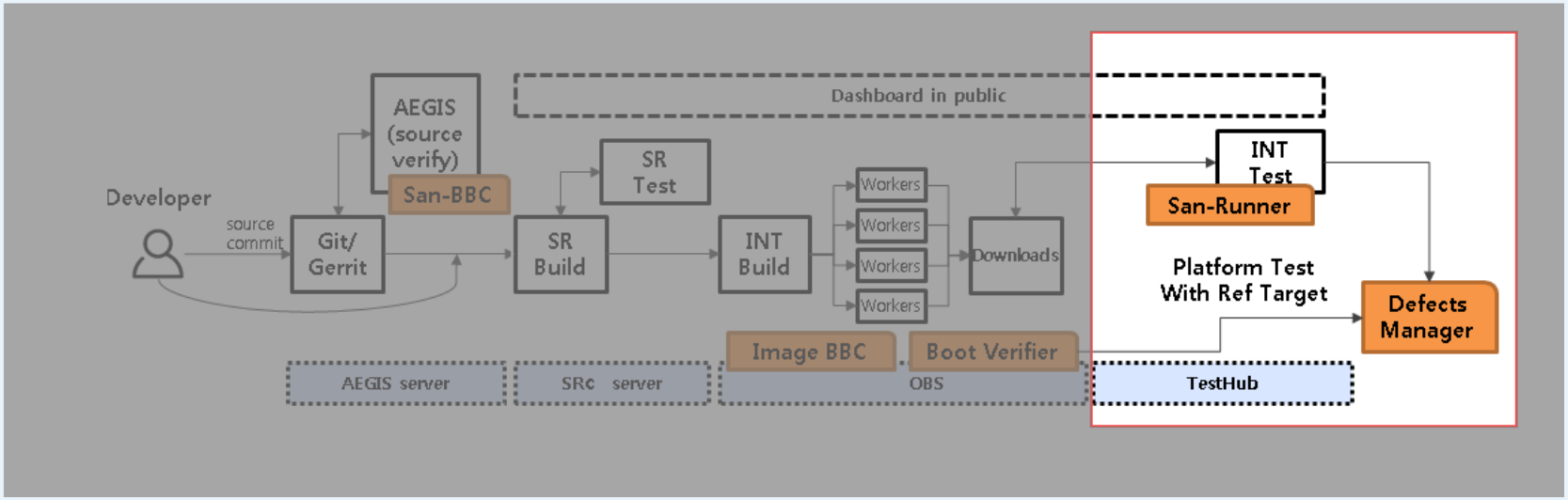


Process

- Создание прошивки из собранных приложений
- Проверка загрузки прошивки на девайс (размер, стабильность)

Integration with services

Overview

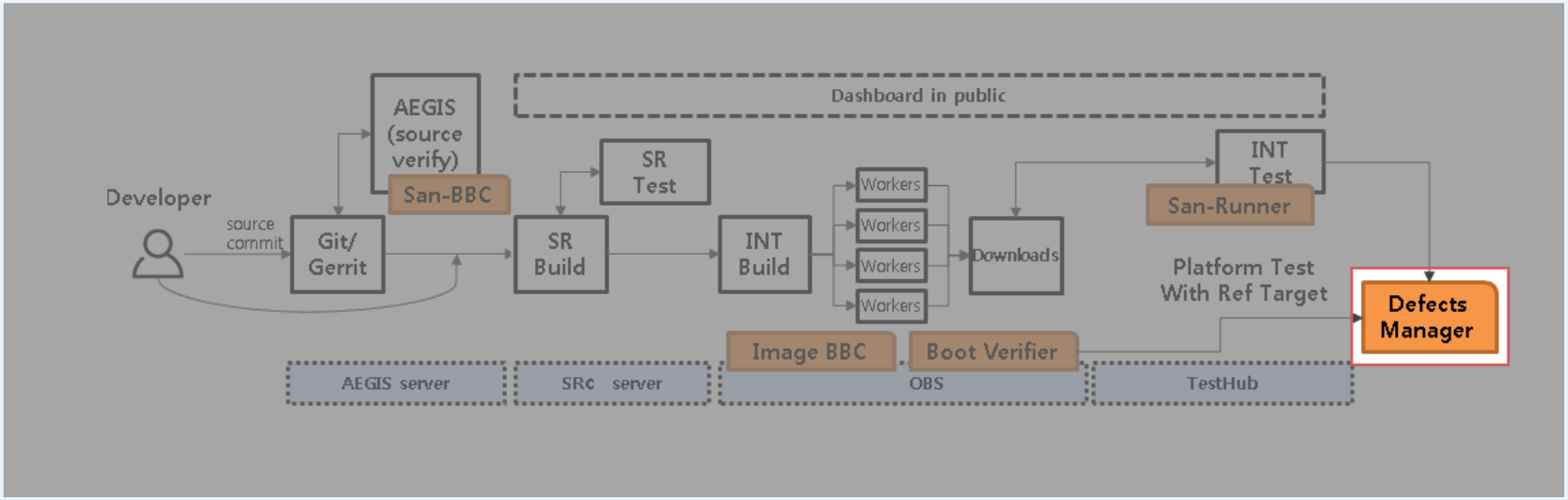


Process

- Запуск тестов на платформе с санитизированной прошивкой
- Сборка логов от санитайзеров
- Сборка необходимой дебажной информации приложений (debuginfo)
- Отправка логов в Defect Manager для дальнейшего анализа

Integration with services

Overview



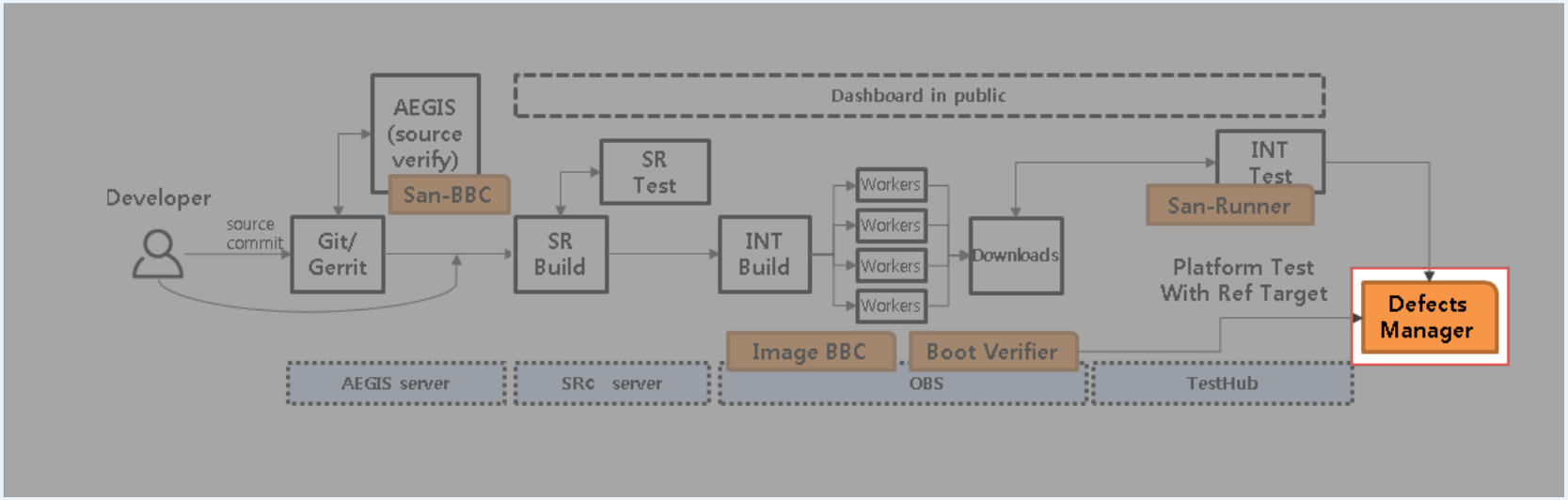
Символизация

Символизация логов с помощью собранной дебажной информации

- `#0 0x10781 in main (/opt/var/tmp/hk0110/a.out+0x5e38b)`
- `#0 0x10781 in main /root/test.c:5`

Integration with services

Overview



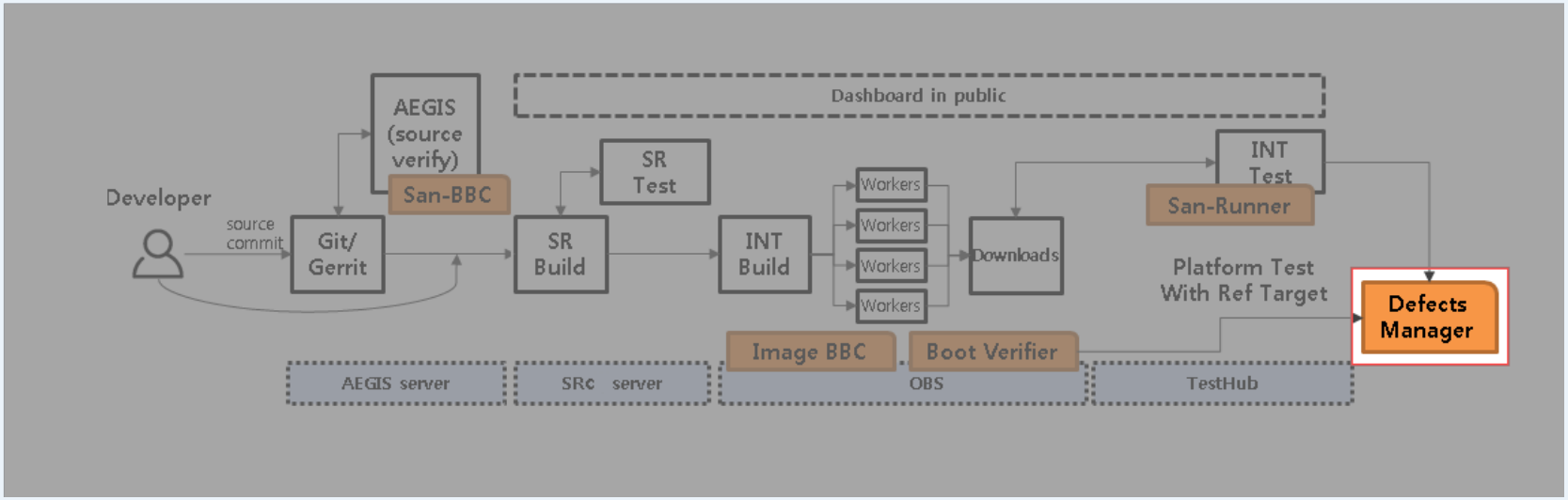
Анализ

Определение проблемного приложения по логу

- `==31423==ERROR: AddressSanitizer: SEGV on unknown address ...`
- `#0 0x7f44a249ee96 in foo /build/glibc/.../nptl-signals.h:80`
- `#1 0x7f449f3c2f27 in bar /home/Git/coreclr/.../process.cpp:3101`
- `#3 0x7f440c6189a3 in baz (/lib/precompiled.so+0x9a3)`

Integration with services

Overview

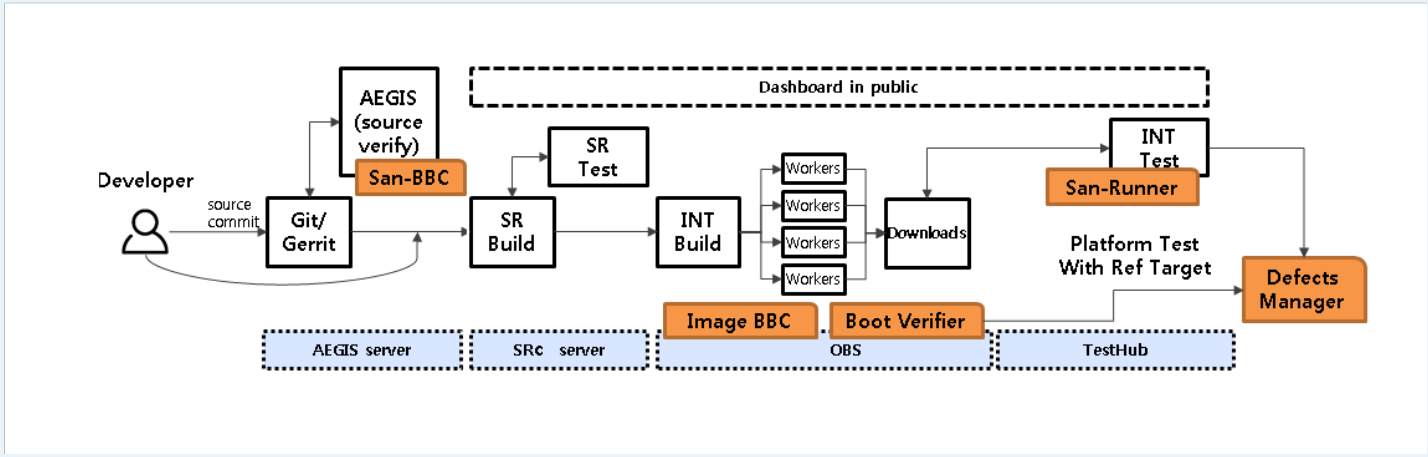


Создание Jira tickets

В результате анализа логов создаются тикеты в проектах команд, ответственных за затронутое приложение со всей необходимой информацией для воспроизведения.

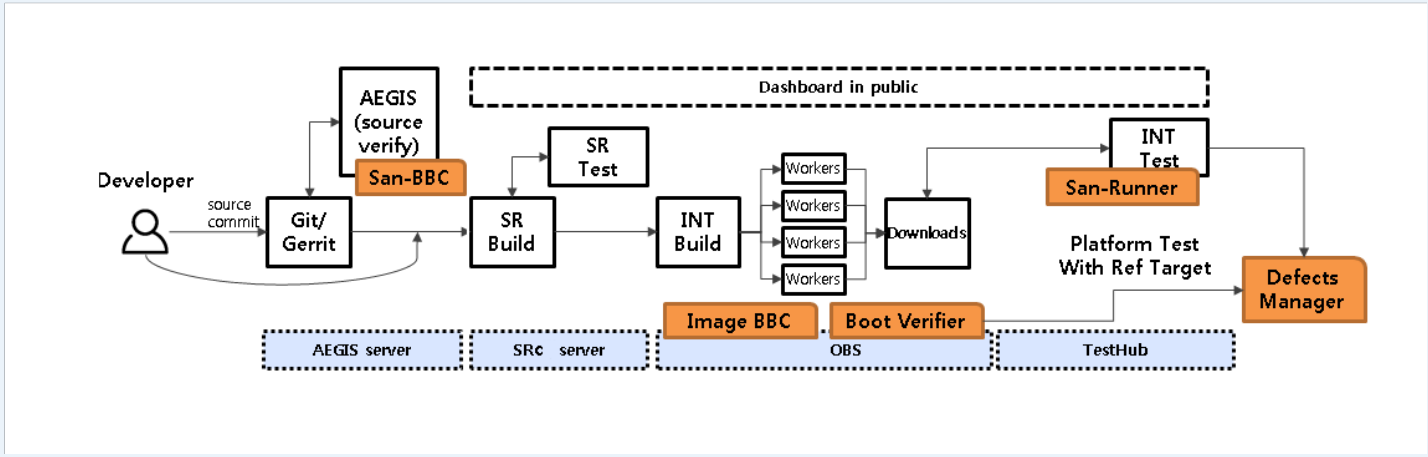
Integration with services

Example pipeline



Integration with services

Example pipeline



Integration with CI

Infra	Input	Output
Build break checker	Sources, project	Build ok/fail
Fully ASaned image creator	Triggered build	Build results, images
Boot verifier	ASan'ed image	Set of logs
Testsuite runner	ASan'ed image	Set of logs
Log analyzer	ASan logs	Analyzed reports
Reporter	Analyzed asan logs	Jira tickets, dashboard

Thank You!

References

[1] GCC Bugzilla. Bug 81697 - incorrect asan global variables alignment on arm, 2017. URL https://gcc.gnu.org/bugzilla/show_bug.cgi?id=81697#add_comment.

[2] GCC Maillist. sanitizer/77631 - support separate debug info in libbacktrace, 2017. URL https://gcc.gnu.org/ml/gcc-patches/2017-07/msg01958.html.

[3] GNU Bison Maillist. grammar: fix memory access bug, 2017. URL http://lists.gnu.org/archive/html/bison-patches/2017-07/msg00001.html.