Applying GCC-based Address Sanitizer to Tizen OS

Vycheslav Barinov

2017-12-01

Outline

1. **Introduction**
   - Address Sanitizer
   - Tizen
   - Problem
2. **Building sanitized Tizen**
   - Integration approach
   - Compiler
   - Build system
   - Build acceleration
3. **Running sanitized Tizen**
   - Running on device
   - Binary size
   - RAM usage
   - UX improvements
4. **Other issues**
5. **Results**

Introduction

Address Sanitizer

Definition

Address Sanitizer (ASan) is a fast memory error detector. It finds use-after-free and heap,stack,global-buffer overflow bugs in C/C++ programs.

Structure

Address Sanitizer consists of two major parts

- Compiler internal part
- Run-time support library `libasan.so`

Pros and cons

- Requires recompilation
- Much faster than Valgrind ($\times 2\ -\ \times 3$ overhead)

Tizen

OS

Tizen is an open source operating system based on the Linux kernel and the GNU C Library implementing the Linux API.

Toolchain

- Linaro GCC
- GNU Glibc
- GNU Binutils

Package management

Tizen uses `rpm` as package manager and `OBS` to build packages.

Applicable software

Tizen OS contains lots of code in C/C++ languages which are known to have issues which can't be detected during compilation stage but can affect resulting application. Sanitizer is the tool for issues detection during runtime.

Problem statement

Background

Address Sanitizer investigation in SRR started in 2013, after several years of technology stabilizing it was applied to several applications.
The investigation of ASan applicability to Tizen apps started in 2015 (the Article [2] has been prepared).
After we ensured that ASan could be applied to Tizen applications the idea extended into full firmware sanitization.

Project target

Build every Tizen binary with Address Sanitizer and create a "mirror" firmware fully equal to plain Tizen, but fully sanitized.

Additional targets

- Check existing code and report to maintainers
- Create tooling for ASan in Tizen toolchain
- Prepare full documentation set

Building sanitized Tizen

Integration steps

1. Build GCC with ASan support
2. Integrate with OBS build
3. Prevent build from failing all the time
4. Enable build acceleration with ASan
5. Run on device
6. Add UI

GCC Build

Introducing compiler feature

Tizen is locked to certain toolchain build, merge windows are tied to release time frames. Minor updates are possible only if they do not require any changes to non-sanitized builds.

Approach

- Enable build of ASan infrastructure
- Create new `rpm`-package with `libasan.so`
- Test compiler with and without the package
- Make full build of staging project

GCC sanitization

Should we perform sanitized bootstrap `--with-build-config=bootstrap-asan` or not?
Our decision: both.

- Sanitized GCC in internal staging project
- Non-sanitized GCC in external "release" project

OBS Integration

Build procedure

- Each build is performed in isolated container (`qemu-kvm`)
- Each package build results into `rpm` (or several ones)
- `rpm`'s joined into *projects*
- Each project is configured separately and has common metadata

Naive approach

Project has special configuration macro `Optflags` which contains compiler flags applied to every build in scope of this project.

Issues

- Not all packages are built using GNU Autotools or CMake
- Not all packages do honor `Optflags`
- `libtool`

Working approach

Custom compiler wrapper with fine tuning possibility we named `gcc-force-options`

OBS Integration

ASan environment in container

Each build container for `rpm` package is create right before the build start so the ASan environment must be set up together with it. At least the following steps are needed:

- Install `libasan.so`
  Done via project config and `Preinstall`
- Add `libasan.so` into `LD_PRELOAD`
  Done via creation of aux package with right post-install script
- Provide `ASAN_OPTIONS`
  Done via run-time part patch. Our `libasan.so` reads an *option file* `/ASAN_OPTIONS`
- Collect ASan logs after build is finished
  Done via additional `rpm` scripts

ASan build influence

Running all the tools under Address Sanitizer usually causes two main issues:

- Memory errors caught by ASan with following failure
  Resolved via *recovery mode*
- `configure`/`cmake` failures due to unexpected ASan output
  Resolved via output redirection

Build acceleration

Reason

There is a lot of hardware targets for Tizen, we build at least `i586`, `x86_64`, `armv7l`, `aarch64` and `mips` and `qemu-user` is used to run the target binaries during build.
Since `qemu-user` is rather slow, it's better to use cross-compiler and set of other cross-tools.

Implementation

It's possible to create and use cross-x86_64-to-armv7l toolchain, but its not possible e.g. for `m4` or `grep` which are widely used in build. The solution is to replace `armv7l` tools to `x86_64` ones right inside the container.

1. Pack `x86_64` binaries into `qemu-accel.rpm`
2. Move them into special `/emul` dir
3. Use `patchelf` to update library paths
4. Add `qemu-arm-binfmt` wrapper
5. Install package to every buildroot

Structure

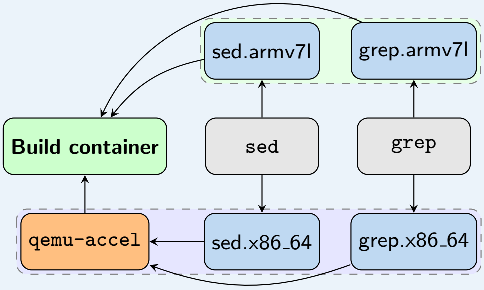

Figure: `qemu-accel` structure

Build acceleration

Pros

- Huge speed-up (up to ×6 times)
  glibc configure time for armv7l

Mode	Time
No qemu-accel	1m25s
qemu-accel	0m13s

- Easy to switch off (remove /emul)
- Ease to maintain (single rpm)

Cons

- Rather large (had to separate)
  Added:
  - python-accel
  - clang-accel
- Requires efforts for *hacks*
- Not very clear for understanding
  We regularly get questions on qemu-accel internals in a form "Suddenly everything got broken!".

Structure

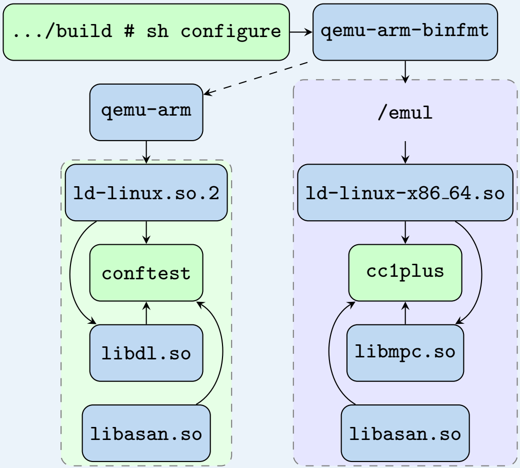

Figure: Accelerated buildroot structure

ASan build setup

OBS setup

To enable ASan build:

- Add ASan to project config

```
Preinstall: asan-force-options
%define asan 1
Macros:
%asan 1
:Macros
```

- Switch off Thumb for `armv7l` build (recommended)[a]

```
Optflags: armv7l ... -marm -fno-omit-frame-pointer
```

- Wait for until packages are built

[a] ASan fast unwinder (fp-based) doesn't work in Thumb mode

Structure

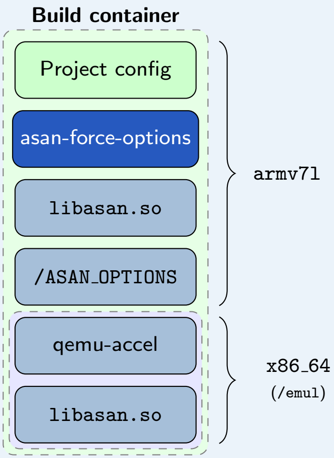

Figure: Build container in Tizen OBS

Results

- Build firmware after package build is finished
- Scan build logs for errors found by ASan

Running sanitized Tizen

Running on device

Goals

- **Short term**
  Boot target device with fully sanitized image and make sure that it works properly:
  - Launch each application manually and make sure that they work properly (OK, just don't fail due to ASan)
- **Long term**
  Establish regular builds and perform regular testing of sanitized images:
  - Run automated Tizen testsuite on sanitized image regulary
  - Integrate ASan into Tizen images verification process

Challenges

- **Sanitized image size**
  Size of sanitized image is much bigger then size of regular one
- **Memory consumption**
  Sanitized image consumes much more physical memory then regular one
- **Bugs in ASan itself**
  ASan is pretty stable nowadays, we still hit on bugs in some corner cases

Sanitized image size

Firmware size

Sanitized image is much bigger then regular one:

- Original size (compressed tarball): **327.6 MB**
- Sanitized size (compressed tarball): **456.4 MB**
- Difference: **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%
`.data`	1.8	9.7	8.1	450%
...	...	...	...	...
Total:	**39**	**146**	**107**	**274%**

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

Image size reduction

Recipes

- Optimize for code size
  `CFLAGS+="-Os"`
- Do not instrument global variables
  `CFLAGS+="--param asan-globals=0"`
- Use outline instrumentation
  `CFLAGS+="--param asan-instrumentation-with-call-threshold=0"`

Mode	.text(MB)	.rodata (MB)	.data (MB)	Total (MB)
Normal code	29.3	4.4	1.8	39
Inline instrumentation	108.1	19.0	9.7	146
Increase vs normal	**268%**	**332%**	**450%**	**274%**
Without globals	98.6	4.4	1.8	109
Increase vs normal	**248%**	**0%**	**0%**	**179%**
Outline instrumentation	62.9	19.0	9.7	101
Increase vs normal	**116%**	**332%**	**420%**	**158%**

Table: Binary size comparison with different options for **libchromium.so**

Memory consumption

Memory overhead sources

- **Allocator quarantine**
  Can be tuned by `quarantine_size_mb` runtime option
- **ASan redzones**
  Can be reduced by not instrumenting some parts of applications (e.g. global variables)
- **ASan shadow**
  Can be reduced by using more compact shadow (e.g. 16:1)
- **ASan fake stacks**
  Can be eliminated by disabling `stack-use-after-return` detection
- **Code and data bloating**
  Can be reduced by optimizing for code size (`-Os`, `--param asan-instrumentation-with-call-threshold=0`)
- **Allocator implementation**
  Sanitizer allocator is tuned for speed and scalability. Yet it can be tuned to be less memory consuming

Memory consumption

Mitigating OOM killer in Tizen

- **Reduce quarantine size to minimally possible value (1MB in our case)**
  In theory this can lead to missing some use-after-free bugs, but we haven't seen this in practice
- **Disable stack-use-after-return detection**
  Use-after-return mode is very memory consuming (up to $\times 2$ additional memory overhead)
- **Tweak ASan allocator**
  - ASan's primary allocator divides memory chunks in **size classes** (52 of them)
  - For each size class except the largest one ASan mmaps **1MB** of memory on demand (these regions are called **memory regions**) and uses it as banks of chunks. For the last class ASan just uses **mmap**.
  - For most applications **1MB** for each memory region is too wasteful so region size was reduced to **128KB**. This gave additional **100MB** of memory footprint reduction.
  - We also thougth about tweaking size classes number (reduce from 52 to, say, 40) but this didn't give us any noticable improvement
  - NOTE: all these tweaks were performed for `SanitizerAllocator32`. The 64-bit `SanitizerAllocator64` uses completely different allocation strategy.
- **Enable swapping**
  - Allows to run heavy applications like Tizen browser
  - Makes sanitized image more stable

UX improvements

Points

- Recovery mode
  - Originally was a local patch, contributed upstream by Yuri Gribov
  - Allows to find more bugs during one test cycle
  - Available in GCC 6+
- Automatic `/proc` mounting
  - Needed for `systemd` sanitization
- `print_cmdline` runtime option
  - Can be useful when debugging background processes
- `libbacktrace` separate debuginfo support [3]
  - Greatly improves usability in stripped environment
  - Patch is under review upstream
- Reading `ASAN_OPTIONS` from file
  - Makes ASan runtime setup more flexible in our environment
- SMACK support
  - `setxattr(2)` is called to set SMACK label for logs and make them readable for user
- Various bugfixes e.g. wrong global variables alignment in sanitized binary [1]

Resulting setup

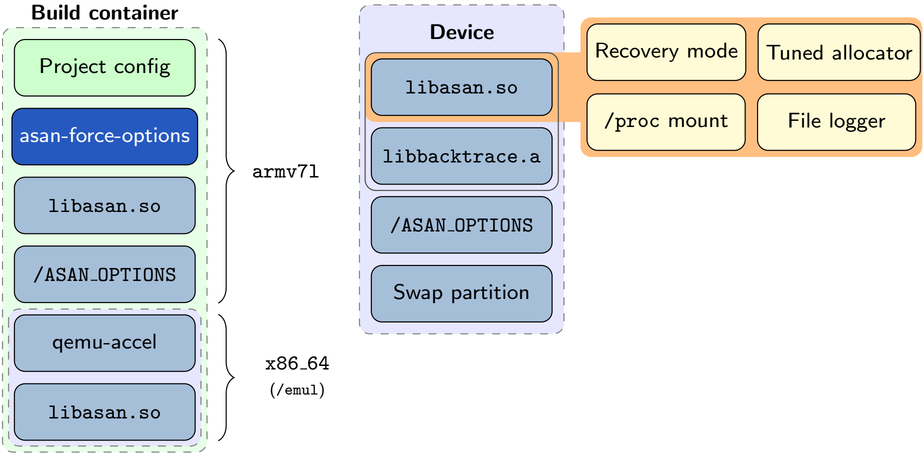

Figure: Resulting ASan setup in Tizen

Other issues

Source code issues

Build issues

- Static builds
  Some binaries must be linked statically (e.g. `initrd` internals), they require additional efforts to build. We either add custom build scripts or use non-sanitized version.
- Support tools like `patchelf`
  Unfortunately `patchelf` is not perfect and sometimes corrupts binaries.
- Rebuild time
  Any `libsanitizer` patch causes rebuilds of compiler, which causes rebuild of the whole OS and takes lots of time.

Open-Source code issues

- Ancient bugs
  Sometimes really old bugs are met, like `bison` bug [4] in unexpected places.
- Unexpected failures
  Sometimes weird things happen, like one found in `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
  ASan gives certain performance penalty and some services get killed by `systemd`.
- `systemd` slice limits
  Most services are in `cgroup` slices and some limits are too small for sanitized binaries.

Corporate issues

- Company size
  Many teams work on Tizen in different ways and with different requirements and development practices.
- Rules and processes
  Corporate limitations do exist, so support different divisions and code open-sourcing is not so easy.

Results

Results

Short-term goals reached

After we enabled Address Sanitizer and were able to boot firmware we found 12 bugs instantly:

- 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

And those were found just after running device and random clicking buttons in apps!
The bugs were fixed quickly by developers after they received ASan reports.

Long-term goals reached

- Regular build procedure of sanitized firmware established
- Sanitized images tested by QA team periodically
- Infrastructure prepared during ASan integration will be reused for other sanitizers

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] Yury Gribov, Maria Guseva, Andrey Ryabinin, JaeOok Kwon, SeungHoon Lee, HakBong Lee, and ChungKi Woo. Fast memory debugger for large software projects, 2015. URL http://injoit.org/index.php/j1/article/viewFile/231/184.

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

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