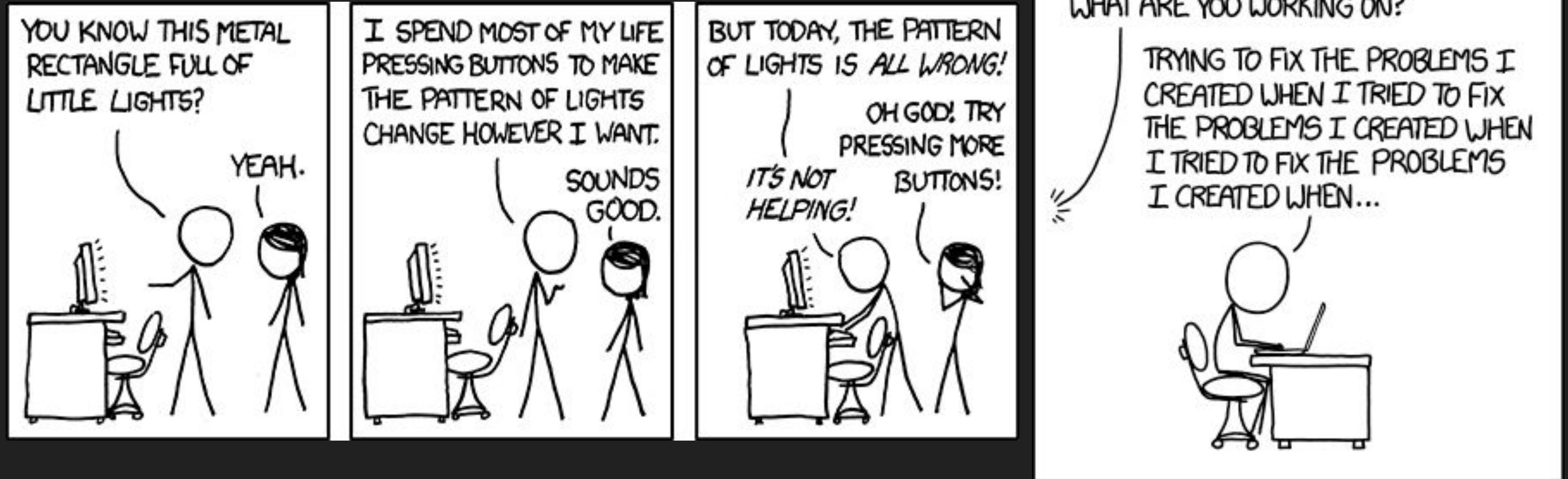
Address/MemorySanitizer, valgrind

and their integration into non-trivial C projects and CI

Jakub Jelen

C programming



C: Insecure programming language

- heap/stack overflows/underflows
- uninitialized memory
- double free
- memory leaks
- …

→ CVEs (DoS, RCE, …)

Hundreds of thousands of lines of C code …
… in hundreds of crucial projects …
… dozens of years of history …
… and people still create new projects

- Rewrite all to some safer language?

or

- Introduce static and dynamic analyzers

Dynamic analyzers: prerequisites & common features

Prerequisites:

- C project
- The test suite with reasonable code coverage

Common features:

- Provide address/memory sanitization
  - With CPU and memory overhead!
- Detect address/memory issues in executed code

What is the difference?

AddressSanitizer:

- build with instrumentation
- run (tests) **normally**

Valgrind:

- build **normally**
- run code (tests) under valgrind

AddressSanitizer:

- gcc **-fsanitize=address** hello.c
- ./a.out

Valgrind:

- gcc hello.c
- **valgrind** ./a.out

Easy?

Dynamically loaded libraries

- PKCS#11 modules
- OpenSSL providers/engines
  - Internal: default/fips
  - Third-party: pkcs11
- …

Example:

```
h = dlopen(argv[1], RTLD_NOW | RTLD_LOCAL | RTLD_DEEPBIND);
…
dlclose(h);
```

What could go wrong?

AddressSanitizer: RTLD_DEEPBIND

- common dlopen flag
- Does not work with ASAN

- Change code – not crucial flag
- Override library call using LD_PRELOAD to remove the flags from each call if outside of reach

```
$ ./hello_dl `realpath hello_lib.so`
==37678==You are trying to dlopen a hello_lib.so shared library with
RTLD_DEEPBIND flag which is incompatible with sanitizer runtime (see
https://github.com/google/sanitizers/issues/611 for details). If you want
to run hello_lib.so library under sanitizers please remove RTLD_DEEPBIND
from dlopen flags.
```

AddressSanitizer: First invoked binary instrumented

Loading the instrumented library from third-party application

- Example: provider for openssl

- Rebuild whole OpenSSL with asan?
- Or use dynamic address sanitizer through LD_PRELOAD

```
$ ./hello_dl_noninstrumented `realpath hello_lib.so`
==39102==ASan runtime does not come first in initial library list; you
should either link runtime to your application or manually preload it with
LD_PRELOAD.

$ LD_PRELOAD=/usr/lib64/libasan.so.8.0.0 \
  ./hello_dl_noninstrumented `realpath hello_lib.so`
Hello world
```

AddressSanitizer: Useful backtraces

- Unloading the library with `dlclose()`, removes debug symbols
- Backtraces are useless

```
$ LD_PRELOAD=/usr/lib64/libasan.so.8.0.0 \
  ./hello_dl_noninstrumented `realpath hello_lib_leaks.so`
Hello world
=================================================================
==40473==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 12 byte(s) in 1 object(s) allocated from:
     #0 0x7fe9a6c814a8 in strdup (/usr/lib64/libasan.so.8.0.0+0x814a8) (BuildId:
542ad02088f38edfdba9d4bfa465b2299f512d3e)
     #1 0x7fe9a7314177  (<unknown module>)
     #2 0x401218 in main (.../asan_talk/hello_dl_noninstrumented+0x401218)
(BuildId: 05540ebc82700da10d571f4b09db6d19372b2b82)
[...]
SUMMARY: AddressSanitizer: 12 byte(s) leaked in 1 allocation(s).
```

AddressSanitizer: Useful backtraces

- Remove `dlclose()` library calls
- Or override them with LD_PRELOAD if outside of reach (after asan!)

```
$ LD_PRELOAD=/usr/lib64/libasan.so.8.0.0:`realpath fake_dlclose.so` \
  ./hello_dl_noninstrumented `realpath hello_lib_leaks.so`
Hello world
=================================================================
==40473==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 12 byte(s) in 1 object(s) allocated from:
    #0 0x7fe9a6c814a8 in strdup (/usr/lib64/libasan.so.8.0.0+0x814a8) (BuildId:
542ad02088f38edfdba9d4bfa465b2299f512d3e)
    #1 0x7f291fc7e137 in write_hello (.../hello_lib_leaks.so+0x1137) (BuildId:
22ffde08890b5e02463407bc1e1cc7ac2a21a26c)
    #2 0x401218 in main (.../asan_talk/hello_dl_noninstrumented+0x401218)
(BuildId: 05540ebc82700da10d571f4b09db6d19372b2b82)
```

AddressSanitizer: Tests using other LD_PRELOAD libs

Wrappers allow running testsuite of complicated application in user-space:

- socket_wrapper – simulate network communication
- uid_wrapper – simulate root user and user switching
- nss_wrapper – emulate users, groups …
- pam_wrapper – emulate PAM conversation
- priv_wrapper – emulate privilege separation/seccomp

There is a bug in glibc

- Sorry, won’t not work now!

AddressSanitizer: shell scripts test drivers

- Some binary tests run from shell scripts
- Do not change environment for them!
  - They might break
  - They will be slow
  - There will be false positives
- Create helper variable
  - ex. $CHECKER
- Prefix each test program infocation with it

```
$ cat fake_dlclose.c
#include <stdio.h>
int dlclose(void *h){
    return 0;
}
$ gcc -c fake_dlclose.c -o \
    fake_dlclose.o
$ gcc -shared fake_dlclose.o -o \
    fake_dlclose.so
$ ASAN=/usr/lib64/libasan.so.8.0.0
$ DLCLOSE=$(realpath fake_dlclose.so)
$ CHECKER=”env
LD_PRELOAD=$ASAN:$DLCLOSE”

$ $CHECKER test_cmd
```

AddressSanitizer: tweaks and related

- Incomplete backtraces?
  - ASAN_OPTIONS='fast_unwind_on_malloc=0'
- Third-party library issues can be suppressed:
  - LSAN_OPTIONS=”suppressions=`realpath lsan.supp`”;
- Other configuration through environment
- Other sanitizers work similarly
  - Memory Sanitizer – uninitialized memory, clang only, more tweaks
  - Leak Sanitizer (integrated in Address Sanitizer)
  - Undefined Behavior Sanitizers (clang only)

AddressSanitizer: Summary

- So not so normal invocation …

Demo?

valgrind (mostly memcheck)

Similar issues:

- dlclose() removes debuginfo
  - → Unusable backtraces
  - LD_PRELOAD library to make it no-op
  - Remove the function call
  - Use `--keep-debuginfo=yes`
- Avoid running shell scripts under valgrind
- Suppression file:
  - `--suppressions=proj.supp`

Different:

- It's much slower (10x)
  - Might need add longer sleeps/waits
- Its noisy: use `-q`
- Change exit code on error:
  - `--error-exitcode=1`

Invocations:

- Code/test modifications
- Run test command under valgrind

```
$ CHECKER="valgrind -q
--keep-debuginfo=yes"

$ $CHECKER test_cmd
```

It's getting similar now …

valgrind: more information

Valgrind is not only memcheck:

- Other tools for other use cases
- cachegrind, callgrind, helgrind, drd, massif, dhat, lackey, exp-bbv, …

If interested in some, please, let me know

Putting it all together

Different build systems

- Autoconf should be enough for everyone

unless it is not

- CMake
- Meson
- …

Autoconf, automake and autotools

valgrind

- There are macros in autoconf-archive:
  - https://www.gnu.org/software/autoconf-archive/ax_valgrind_check.html
  - `./configure --enable-valgrind`
  - `make check-valgrind-memcheck`
  - Exports $VALGRIND environment
  - → change to $CHECKER

AddressSanitizer

- Update CFLAGS and LDFLAGS:

```
CFLAGS="-fsanitize=address $CFLAGS"
LDFLAGS="-fsanitize=address $LDFLAGS"
```

- Prepare $CHECKER environment variable
- Run with `make check`

Common:

- Modify all tools invocation from shell scripts to be prefixed with $CHECKER
- Do not use both valgrind and asan!

Examples:

- https://github.com/OpenSC/OpenSC/pull/2756/files
- https://github.com/latchset/pkcs11-provider/pull/243/files

CMake: valgrind

- There is memcheck support in ctest:
  - https://cmake.org/cmake/help/latest/manual/ctest.1.html#ctest-memcheck-step
  - `--test-action memcheck`
- Detection of memory issues
  - less obvious as the `ctest` returns 0 even with errors
  - writes separate files with logs
  - Non-obvious way to provide suppression file

Example:

- https://gitlab.com/libssh/libssh-mirror/-/merge_requests/365/diffs

CMake: AddressSanitizer

- Build and link flags configured by CMAKE_BUILD_TYPE:
  - https://gitlab.com/libssh/libssh-mirror/-/blob/master/cmake/Modules/DefineCompilerFlags.cmake
  - Running the tests as usually
- ASAN not tested
  - Issues with LD_PRELOAD in libssh

What next?

- Have upstream projects?
- Static analyzers
  - Fast, but simple: can not imagine all the possible code paths
  - Coverity scan integrated in upstream CI
- Combine with fuzzing to find new inputs/code paths
  - Review code coverage regularly
  - Implement new fuzzers
  - Oss-fuzz infrastructure
  - Interested in more information? Let me know!
- Resolve the glibc issue
  - And extend the coverage

Thanks!