Bug detecting software in the NetBSD userland: MKSANITIZER

MeetBSDCa 2018

Author: Kamil Rytarowski

E-mail: kamil@netbsd.org

Date: October 19th 2018

Place: Intel Santa Clara Campus, California, USA

Bio

Kamil Rytarowski (born 1987)

Krakow, Poland

NetBSD user since 6.1.

The NetBSD Foundation member since 2015.

Work areas: kernel, userland, pkgsrc.

Interest: NetBSD on desktop and in particular NetBSD as a workstation.

The current activity in 3rd party software:

- LLVM committer.
- GDB & binutils committer.
- NetBSD maintainer in qemu.

Topics

- What are sanitizers?
- Types of sanitizers
- Characteristics of sanitizers
- NetBSD compatibility challenges
- Interceptors
- MKSANITIZER
- A selection of fixed programs

What are sanitizers?

Sanitizer is a programming tool that detects computer program bugs such as:

- buffer overflows,
- signed integer overflow,
- uninitialized memory read,
- data races etc.

Types of sanitizers

The fundamental four types of sanitizers.

- Address Sanitizer (Asan) - Finds invalid address usage bugs.
- Undefined Behavior Sanitizer (UBSan) - Finds unspecified code semantics bugs.
- Thread Sanitizer (TSan) - Finds threading bugs.
- Memory Sanitizer (MSan) - Finds uninitialized memory read.

All of them are supported on NetBSD.

Characteristics of sanitizers

- Checks are performed dynamically in runtime.
- Compiler (Clang, GCC) emits checks inlined into the generated code.
- Runtime handles non-trivial validation and reporting of bugs.

Sanitizers vs Valgrind

Sanitizers.

- Compile-time instrumentation
- Slowdown 2x
- Decent portability
- Detects: out-of-bounds heap, out-of-bounds stack, out-of-bounds globals, use-after-free, use-after-return, uninitialized-memory-read, leaks, undefined-behavior, data races

Valgrind.

- Dynamic-binary instrumentation
- Slowdown 20x
- Difficult porting to new platforms and OSes
- Detects: out-of-bounds heap, use-after-free, uninitialized-memory-read, leaks, data races

Sanitizers in the NetBSD userland

The base distribution (HEAD version).

- GCC-style distribution: ASan, UBSan, LSan (scratch)
- LLVM-style distribution: shipping with the distribution coming soon

Externally prebuilt standalone toolchain.

- GCC: downstream for LLVM, ignored right now
- LLVM: occasional prebuilt snapshots for public consumption
  http://cdn.netbsd.org/pub/NetBSD/misc/kamil/

Design of LLVM Sanitizers

Design choices.

- Sanitizers are designed to run on top of libc, libpthread, libm, librt
- The c, pthread, m and rt libraries are not instrumented (no compiler flags during build)
- Most sanitizers keep track of allocations from unistrumented libraries with malloc(3) and mmap(2) interceptors
- Written in C++ and ship with native support of sanitization of C++ code
- Requirement of low-level C++ runtime features (LLVM libc++ and GNU libstdc++ supported)
- Most sanitizers require large address space for meta-buffers

Environment setup

Code sanitized with one sanitizer flags shall not be linked with code sanitized with other options.

Implication.

- All the dependencies and a program itself must be built with the same set of compiler flags for sanitizers.

Example of Address Sanitizer (ASan).

```
1 cc -fsanitize=address main.c -o main
```

Interoperability with PaX ASLR

Problems generated by the design of sanitizers: *ASLR*.

- Incompatiblity with NetBSD PaX ASLR (as of now) due to requirement of mapping allocated buffers into meta-buffers of sanitizers
- NetBSD's Address Space Layour Randomization is too aggressive
- ASan randomly work with ASLR; TSan & MSan break all the time
- Crashes caused by ASLR tend to be hard to understand

Workaround.

- Disable PaX ASLR either globally or per-application
- Included runtime detector of PaX ASLR for NetBSD applications and bail out with an error message

Interceptors

Interceptor is a wrapper for a library function.

Original function in a library.

```
1 ret_type function(int a0, arg_type1 *a1, arg_type2 *a2);
```

Inteceptor wraps it (pseudocode).

```
1 ret_type wrapper_function(int a0, arg_type1 *a1, arg_type2 *a2) {
2    INITIALIZE_INTERCEPTOR(); // sanitizer-specific initialization
3    PRE_READ(a1);   // sanitizer-specific pre-read operations
4    PRE_WRITE(a2);  // sanitizer-specific pre-write operations
5    ret_type rv = REAL(function)(a0, a1, a2);
6    POST_READ(a1);  // sanitizer-specific post-read operations
7    POST_WRITE(a2); // sanitizer-specific post-write operations
8    return rv;
9 }
```

Sanitizers replace references of the real *function()* with references of *wrapper_function()*.

Warning: in reality a large amount of interceptors contains various special cases.

Interceptors

Interceptor for strnlen(3).

```
 1 #if SANITIZER_INTERCEPT_STRNLEN
 2 INTERCEPTOR(SIZE_T, strnlen, const char *s, SIZE_T maxlen) {
 3   void *ctx;
 4   COMMON_INTERCEPTOR_ENTER(ctx, strnlen, s, maxlen);
 5   SIZE_T length = REAL(strnlen)(s, maxlen);
 6   if (common_flags()->intercept_strlen)
 7     COMMON_INTERCEPTOR_READ_RANGE(ctx, s, Min(length + 1, maxlen));
 8   return length;
 9 }
10 #define INIT_STRNLEN COMMON_INTERCEPT_FUNCTION(strnlen)
11 #else
12 #define INIT_STRNLEN
13 #endif
```

Interceptors

On NetBSD (ELF program file format) interceptors use dynamic loader functionality to install interceptors for routines from a dynamically loaded library.

Sanitizer's runtime inlines a local copy of a certain interceptor into the body of a program, and this symbol is resolved before resolving a symbol from a library.

Interceptors

- Sanitizers (usually) do not know whether arbitrary data region is initialized
- Sanitizers (usually) do not know whether base library will either read or write data referenced by passed pointer
- Sanitizers (usually) do not know what happens inside unsanitized libraries (c, rt, etc)

Solution.

- Introduce an interceptor for every libc, libm, librt, libpthread API call that passes data (in any direction: in / out) over a pointer
- Handle special cases on per-sanitizer, per-API and per-OS (& per-libc) basis

Interceptors

Interceptors can be triggered when code is executing inside a base library and sanitizers usually do not know the context of the surrounding source code

Example (libedit).

```
 1 /* el_resize():
 2  *      Called from program when terminal is resized
 3  */
 4 void
 5 el_resize(EditLine *el)
 6 {
 7     int lins, cols;
 8     sigset_t oset, nset;
 9
10     (void) sigemptyset(&nset);
11     (void) sigaddset(&nset, SIGWINCH);
12     (void) sigprocmask(SIG_BLOCK, &nset, &oset);
13     ...
```

Memory Sanitizer might report a false positive on an interceptor for *sigprocmask(2)*, because it does not know whether *nset* is initialized, in case of inlined/not-intercepted version of *sigemptyset(3)*.

Interceptors

Solution.

- Disable recursive interceptors in sensitive sanitizers (MSan)
- Recursive interception is disabled on per-interceptor and per-sanitizer basis
- In narrow cases there is a need to install interceptors for functions that do not return or take any arguments (example: *void tzset(void)*) only for the purpose of disabling calls of interceptors inside uninstrumented library
- Cover more function calls in base libraries with interceptors
- Keep adding interceptors until your application starts to execute correctly

Interceptors

GNU world.

- Fragmentation of userland, programs such as GNU grep(1) are developed separately with GNU patch(1) and GNU libc
- Alternative versions of libc (musl, eglibc, newlib, ...)
- Developers restrict themselves to POSIX functions and reinvent utility functions in their programs

BSD world.

- Entire basesystem is developed inside a single source tree
- Single version of libc, unless somone is overly-adventurous
- Developers push common utility routines to either utility libraries (libutil) or directly into base libraries (libc, ...)

Interceptors

The GNU vs BSD implications in sanitizers.

- The BSD world must install many more interceptors than the GNU world (at least 100% more)

Interceptors

- Adding interceptors for C++ libraries is practically undoable due to C++ symbol name mangling (and might be incompatible between compiler versions)
- Adding interceptors for a certain C library requires us to link every sanitized program with that library (such as libutil symbols will require us to link every sanitized program with -lutil)
- Covering symbols from more libraries with interceptors enlarges the runtime, requires a lot of manual work and does not scale
- Adding interceptors for a certain library assumes that we will skip potential bugs inside its code

Interceptors

Certain symbols require a lot of knowledge about API usage.

```
1 int
2 mount(const char *type, const char *dir, int flags, void *data,
3     size_t data_len);
```

The mount(2) call contains per-filesystem structures, e.g.

```
 1 MOUNT_FFS
 2 struct ufs_args {
 3     char      *fspec;             /* block special file to mount */
 4 };
 5
 6 MOUNT_MFS
 7 struct mfs_args {
 8     char      *fspec;             /* name to export for statfs */
 9     struct    export_args30 pad;  /* unused */
10     caddr_t   base;               /* base of file system in mem */
11     u_long    size;               /* size of file system */
12 };
13
14 ... /* over two dozens of filesystems more */
```

Interceptors

In narrow cases interceptors introduce compatiblity issues between 32 and 64-bit code (such as libkvm - interacting with kernel memory).

In the libkvm case it worked to rebuild the library with a sanitizer natively.

Interceptors

Rebuilding manually all the dependencies of complex C++ applications such as the LLVM Debugger is difficult and laborious.

```
 1 $ ldd /usr/local/bin/lldb
 2 /usr/local/bin/lldb:
 3 -lpthread.1 => /usr/lib/libpthread.so.1
 4 -lc.12 => /usr/lib/libc.so.12
 5 -llldb.7 => /usr/local/bin/../lib/liblldb.so.7
 6 -lkvm.6 => /usr/lib/libkvm.so.6
 7 -ledit.3 => /usr/lib/libedit.so.3
 8 -lterminfo.1 => /usr/lib/libterminfo.so.1
 9 -lexecinfo.0 => /usr/lib/libexecinfo.so.0
10 -lelf.2 => /usr/lib/libelf.so.2
11 -lgcc_s.1 => /usr/lib/libgcc_s.so.1
12 -lpython2.7.1.0 => /usr/pkg/lib/libpython2.7.so.1.0
13 -lutil.7 => /usr/lib/libutil.so.7
14 -lm.0 => /usr/lib/libm.so.0
15 -lcurses.7 => /usr/lib/libcurses.so.7
16 -lform.6 => /usr/lib/libform.so.6
17 -lpanel.1 => /usr/lib/libpanel.so.1
18 -lxml2.2 => /usr/pkg/lib/libxml2.so.2
19 -lz.1 => /usr/lib/libz.so.1
20 -llzma.2 => /usr/lib/liblzma.so.2
21 -lrt.1 => /usr/lib/librt.so.1
22 -lstdc++.8 => /usr/lib/libstdc++.so.8
```

Interceptors

Last but not least, if there are already symbols inside an application duplicated with a basesystem - they will conflict with an interceptor.

Workaround.

- Patch the code to rename a symbol to remove name clash (Symbol name clash in a program with UNIX interfaces is in some cases a POSIX spec violation)
- Refer to preprocessor magic and rename the symbol in the fly for the use of sanitization (e.g. -*Dregex=__renamed_regex*)

Interceptors

Summary.

- libc, libm, librt, libposix remain unsanitized
- The BSD world requires more interceptors than the GNU world
- All the other libraries shall be prebuilt with native sanitization
- Processing the userland to ship sanitized libraries manually is impractically difficult

Solution.

- Automatization of preprocessing the entire userland with a selected set of sanitizers with a distribution build flag (NetBSD's MKSANITIZER)

MKSANITIZER

Build and use almost all of the userland with a selected sanitizer.

```
./build.sh \
    -V MKLLVM=yes \
    -V MKGCC=no \
    -V HAVE_LLVM=yes \
    -V MKSANITIZER=yes \
    -V USE_SANITIZER="address,undefined" \
    distribution
```

Unsanitized exceptions: kernel, loadable kernel modules, ramdisks, static libraries, static programs, base libraries (libc, libm, libpthread, librt).

Functional chroot environment: ASan, UBSan, MSan.

Bootable (& installable) distribution into a functional shell: ASan, UBSan.

As of now the MKSANITIZER flag requires external and patched Clang/LLVM toolchain.

A selection of fixed programs

ASan: sh(1), sysinst(8), heimdal krb5, libutil(3), man(1), installboot(8), passwd(8), ...

UBSan: tmux(1), expr(1), ksh(1), ifconfig(8), libc, [gnu]grep(1), gzip(1), [n]awk(1), [n]vi(1), disklabel(8), ...

MSan: sh(1), top(1), ...

... and others that were forgotten to mention.

Sanitizers on NetBSD

Further reading

- Monthly reports on https://blog.netbsd.org/
- Current TODO maintained in the NetBSD sources src/doc/TODO.sanitizers
- Wiki page overview https://wiki.netbsd.org/users/kamil/sanitizers/
- MKLIBCSANITIZER with a homegrown sanitizer runtime inside libc
- NetBSD Kernel Sanitizers

Action needed

- Validate your code with a sanitizer
- Check the list of a reported NetBSD issues and submit patches with fixes!
  http://netbsd.org/~kamil/mksanitizer-reports

Future directions

- kcov(4) and syzkaller - multithreaded coverage-guided kernel fuzzer
- rumpkernel sanitizing and fuzzing - research and innovations