`sanitizer`

The tracking issues for this feature are:

- #39699.
- #89653.

This feature allows for use of one of following sanitizers:

- AddressSanitizer a fast memory error detector.
- ControlFlowIntegrity LLVM Control Flow Integrity (CFI) provides forward-edge control flow protection.
- HWAddressSanitizer a memory error detector similar to AddressSanitizer, but based on partial hardware assistance.
- LeakSanitizer a run-time memory leak detector.
- MemorySanitizer a detector of uninitialized reads.
- [MemTagSanitizer][clang-memtag] fast memory error detector based on Armv8.5-A Memory Tagging Extension.
- ThreadSanitizer a fast data race detector.

To enable a sanitizer compile with `-Zsanitizer=address` , `-Zsanitizer=cfi` , `-Zsanitizer=hwaddress` , `-Zsanitizer=leak` , `-Zsanitizer=memory` , `-Zsanitizer=memtag` , or `-Zsanitizer=thread` .

AddressSanitizer

AddressSanitizer is a memory error detector. It can detect the following types of bugs:

- Out of bound accesses to heap, stack and globals
- Use after free
- Use after return (runtime flag `ASAN_OPTIONS=detect_stack_use_after_return=1`)
- Use after scope
- Double-free, invalid free
- Memory leaks

The memory leak detection is enabled by default on Linux, and can be enabled with runtime flag `ASAN_OPTIONS=detect_leaks=1` on macOS.

AddressSanitizer is supported on the following targets:

- `aarch64-apple-darwin`
- `aarch64-fuchsia`
- `aarch64-unknown-linux-gnu`
- `x86_64-apple-darwin`
- `x86_64-fuchsia`
- `x86_64-unknown-freebsd`
- `x86_64-unknown-linux-gnu`

AddressSanitizer works with non-instrumented code although it will impede its ability to detect some bugs. It is not expected to produce false positive reports.

Examples

Stack buffer overflow:

```rust
fn main() {
    let xs = [0, 1, 2, 3];
    let _y = unsafe { *xs.as_ptr().offset(4) };
}
```

```
$ export RUSTFLAGS=-Zsanitizer=address RUSTDOCFLAGS=-Zsanitizer=address
$ cargo run -Zbuild-std --target x86_64-unknown-linux-gnu
==37882==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffe400e6250 at
pc 0x5609a841fb20 bp 0x7ffe400e6210 sp 0x7ffe400e6208
READ of size 4 at 0x7ffe400e6250 thread T0
    #0 0x5609a841fb1f in example::main::h628ffc6626ed85b2 /.../src/main.rs:3:23
    ...

Address 0x7ffe400e6250 is located in stack of thread T0 at offset 48 in frame
    #0 0x5609a841f8af in example::main::h628ffc6626ed85b2 /.../src/main.rs:1

  This frame has 1 object(s):
    [32, 48) 'xs' (line 2) <== Memory access at offset 48 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind
mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /.../src/main.rs:3:23 in
example::main::h628ffc6626ed85b2
Shadow bytes around the buggy address:
  0x100048014bf0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100048014c00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100048014c10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100048014c20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100048014c30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x100048014c40: 00 00 00 00 f1 f1 f1 f1 00 00[f3]f3 00 00 00 00
  0x100048014c50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100048014c60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100048014c70: f1 f1 f1 f1 00 00 f3 f3 00 00 00 00 00 00 00 00
  0x100048014c80: 00 00 00 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1
  0x100048014c90: 00 00 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
```

```
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==37882==ABORTING
```

Use of a stack object after its scope has already ended:

```rust
static mut P: *mut usize = std::ptr::null_mut();

fn main() {
    unsafe {
        {
            let mut x = 0;
            P = &mut x;
        }
        std::ptr::write_volatile(P, 123);
    }
}
```

```
$ export RUSTFLAGS=-Zsanitizer=address RUSTDOCFLAGS=-Zsanitizer=address
$ cargo run -Zbuild-std --target x86_64-unknown-linux-gnu
=================================================================
==39249==ERROR: AddressSanitizer: stack-use-after-scope on address 0x7ffc7ed3e1a0 at
pc 0x55c98b262a8e bp 0x7ffc7ed3e050 sp 0x7ffc7ed3e048
WRITE of size 8 at 0x7ffc7ed3e1a0 thread T0
    #0 0x55c98b262a8d in core::ptr::write_volatile::he21f1df5a82f329a
/.../src/rust/src/libcore/ptr/mod.rs:1048:5
    #1 0x55c98b262cd2 in example::main::h628ffc6626ed85b2 /.../src/main.rs:9:9
    ...

Address 0x7ffc7ed3e1a0 is located in stack of thread T0 at offset 32 in frame
    #0 0x55c98b262bdf in example::main::h628ffc6626ed85b2 /.../src/main.rs:3

  This frame has 1 object(s):
    [32, 40) 'x' (line 6) <== Memory access at offset 32 is inside this variable
HINT: this may be a false positive if your program uses some custom stack unwind
mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-use-after-scope
/.../src/rust/src/libcore/ptr/mod.rs:1048:5 in
core::ptr::write_volatile::he21f1df5a82f329a
Shadow bytes around the buggy address:
  0x10000fd9fbe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000fd9fbf0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000fd9fc00: 00 00 00 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1
  0x10000fd9fc10: f8 f8 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000fd9fc20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x10000fd9fc30: f1 f1 f1 f1[f8]f3 f3 f3 00 00 00 00 00 00 00 00
```

```
  0x10000fd9fc40: 00 00 00 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1
  0x10000fd9fc50: 00 00 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000fd9fc60: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00 00 f3 f3
  0x10000fd9fc70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000fd9fc80: 00 00 00 00 f1 f1 f1 f1 00 00 f3 f3 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==39249==ABORTING
```

ControlFlowIntegrity

The LLVM Control Flow Integrity (CFI) support in the Rust compiler initially provides forward-edge control flow protection for Rust-compiled code only by aggregating function pointers in groups identified by their number of arguments.

Forward-edge control flow protection for C or C++ and Rust -compiled code "mixed binaries" (i.e., for when C or C++ and Rust -compiled code share the same virtual address space) will be provided in later work by defining and using compatible type identifiers (see Type metadata in the design document in the tracking issue #89653).

LLVM CFI can be enabled with -Zsanitizer=cfi and requires LTO (i.e., -Clto).

Example

```
#![feature(naked_functions)]

use std::arch::asm;
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}
```

```
#[naked]
pub extern "C" fn add_two(x: i32) {
    // x + 2 preceded by a landing pad/nop block
    unsafe {
        asm!(
            "
             nop
             nop
             nop
             nop
             nop
             nop
             nop
             nop
             nop
             lea rax, [rdi+2]
             ret
        ",
            options(noreturn)
        );
    }
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {answer}");

    println!("With CFI enabled, you should not see the next answer");
    let f: fn(i32) -> i32 = unsafe {
        // Offsets 0-8 make it land in the landing pad/nop block, and offsets 1-8
are
        // invalid branch/call destinations (i.e., within the body of the function).
        mem::transmute::<*const u8, fn(i32) -> i32>((add_two as *const
u8).offset(5))
    };
    let next_answer = do_twice(f, 5);

    println!("The next answer is: {next_answer}");
}
```

Fig. 1. Modified example from the Advanced Functions and Closures chapter of the The Rust Programming Language book.

```
$ rustc rust_cfi.rs -o rust_cfi
$ ./rust_cfi
The answer is: 12
```

```
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Fig. 2. Build and execution of the modified example with LLVM CFI disabled.

```
$ rustc -Clto -Zsanitizer=cfi rust_cfi.rs -o rust_cfi
$ ./rust_cfi
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Fig. 3. Build and execution of the modified example with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to change/hijack control flow using an indirect branch/call to an invalid destination, the execution is terminated (see Fig. 3).

```
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

fn add_two(x: i32, _y: i32) -> i32 {
    x + 2
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {answer}");

    println!("With CFI enabled, you should not see the next answer");
    let f: fn(i32) -> i32 =
        unsafe { mem::transmute::<*const u8, fn(i32) -> i32>(add_two as *const u8)
};
    let next_answer = do_twice(f, 5);

    println!("The next answer is: {next_answer}");
}
```

Fig. 4. Another modified example from the Advanced Functions and Closures chapter of the The Rust Programming Language book.

```
$ rustc rust_cfi.rs -o rust_cfi
$ ./rust_cfi
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Fig. 5. Build and execution of the modified example with LLVM CFI disabled.

```
$ rustc -Clto -Zsanitizer=cfi rust_cfi.rs -o rust_cfi
$ ./rust_cfi
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Fig. 6. Build and execution of the modified example with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to change/hijack control flow using an indirect branch/call to a function with different number of arguments than intended/passed in the call/branch site, the execution is also terminated (see Fig. 6).

Forward-edge control flow protection not only by aggregating function pointers in groups identified by their number of arguments, but also their argument types, will also be provided in later work by defining and using compatible type identifiers (see Type metadata in the design document in the tracking issue #89653).

HWAddressSanitizer

HWAddressSanitizer is a newer variant of AddressSanitizer that consumes much less memory.

HWAddressSanitizer is supported on the following targets:

- `aarch64-linux-android`
- `aarch64-unknown-linux-gnu`

HWAddressSanitizer requires `tagged-globals` target feature to instrument globals. To enable this target feature compile with `-C target-feature=+tagged-globals`

Example

Heap buffer overflow:

```rust
fn main() {
    let xs = vec![0, 1, 2, 3];
    let _y = unsafe { *xs.as_ptr().offset(4) };
}
```

```
$ rustc main.rs -Zsanitizer=hwaddress -C target-feature=+tagged-globals -C
linker=aarch64-linux-gnu-gcc -C link-arg=-fuse-ld=lld --target
```

```
aarch64-unknown-linux-gnu
```

```
$ ./main
==241==ERROR: HWAddressSanitizer: tag-mismatch on address 0xefdeffff0050 at pc
0xaaaae0ae4a98
READ of size 4 at 0xefdeffff0050 tags: 2c/00 (ptr/mem) in thread T0
    #0 0xaaaae0ae4a94  (/.../main+0x54a94)
    ...

[0xefdeffff0040,0xefdeffff0060) is a small allocated heap chunk; size: 32 offset: 16
0xefdeffff0050 is located 0 bytes to the right of 16-byte region
[0xefdeffff0040,0xefdeffff0050)
allocated here:
    #0 0xaaaae0acb80c  (/.../main+0x3b80c)
    ...

Thread: T0 0xeffe00002000 stack: [0xffffc28ad000,0xffffc30ad000) sz: 8388608 tls:
[0xffffaa10a020,0xffffaa10a7d0)
Memory tags around the buggy address (one tag corresponds to 16 bytes):
  0xefceffffef80: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffffef90: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffffefa0: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffffefb0: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffffefc0: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffffefd0: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffffefe0: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffffeff0: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
=>0xefceffff000: d7  d7  05  00  2c [00] 00  00  00  00  00  00  00  00  00  00
  0xefceffff010: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffff020: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffff030: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffff040: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffff050: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffff060: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffff070: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
  0xefceffff080: 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
Tags for short granules around the buggy address (one tag corresponds to 16 bytes):
  0xefceffffeff0: ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..
=>0xefceffff000: ..  ..  8c  ..  .. [..] ..  ..  ..  ..  ..  ..  ..  ..  ..  ..
  0xefceffff010: ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..
See https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html#short-
granules for a description of short granule tags
Registers where the failure occurred (pc 0xaaaae0ae4a98):
    x0  2c00efdeffff0050  x1  0000000000000004  x2  0000000000000004  x3
0000000000000000
    x4  0000fffefc30ac37  x5  000000000000005d  x6  00000ffffc30ac37  x7
0000efff00000000
    x8  2c00efdeffff0050  x9  0200efff00000000  x10 0000000000000000  x11
0200efff00000000
    x12 0200effe00000310  x13 0200effe00000310  x14 0000000000000008  x15
5d00ffffc30ac360
```

```
    x16 0000aaaae0ad062c  x17 0000000000000003  x18 0000000000000001  x19
0000ffffc30ac658
    x20 4e00ffffc30ac6e0  x21 0000aaaae0ac5e10  x22 0000000000000000  x23
0000000000000000
    x24 0000000000000000  x25 0000000000000000  x26 0000000000000000  x27
0000000000000000
    x28 0000000000000000  x29 0000ffffc30ac5a0  x30 0000aaaae0ae4a98
SUMMARY: HWAddressSanitizer: tag-mismatch (/.../main+0x54a94)
```

LeakSanitizer

LeakSanitizer is run-time memory leak detector.

LeakSanitizer is supported on the following targets:

- `aarch64-apple-darwin`
- `aarch64-unknown-linux-gnu`
- `x86_64-apple-darwin`
- `x86_64-unknown-linux-gnu`

MemorySanitizer

MemorySanitizer is detector of uninitialized reads.

MemorySanitizer is supported on the following targets:

- `aarch64-unknown-linux-gnu`
- `x86_64-unknown-freebsd`
- `x86_64-unknown-linux-gnu`

MemorySanitizer requires all program code to be instrumented. C/C++ dependencies need to be recompiled using Clang with `-fsanitize=memory` option. Failing to achieve that will result in false positive reports.

Example

Detecting the use of uninitialized memory. The `-Zbuild-std` flag rebuilds and instruments the standard library, and is strictly necessary for the correct operation of the tool. The `-Zsanitizer-memory-track-origins` enables tracking of the origins of uninitialized memory:

```
use std::mem::MaybeUninit;

fn main() {
    unsafe {
        let a = MaybeUninit::<[usize; 4]>::uninit();
        let a = a.assume_init();
        println!("{}", a[2]);
    }
}
```

```
$ export \
  RUSTFLAGS='-Zsanitizer=memory -Zsanitizer-memory-track-origins' \
  RUSTDOCFLAGS='-Zsanitizer=memory -Zsanitizer-memory-track-origins'
$ cargo clean
$ cargo run -Zbuild-std --target x86_64-unknown-linux-gnu
==9416==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x560c04f7488a in core::fmt::num::imp::fmt_u64::haa293b0b098501ca
$RUST/build/x86_64-unknown-linux-
gnu/stage1/lib/rustlib/src/rust/src/libcore/fmt/num.rs:202:16
...
  Uninitialized value was stored to memory at
    #0 0x560c04ae898a in __msan_memcpy.part.0 $RUST/src/llvm-project/compiler-
rt/lib/msan/msan_interceptors.cc:1558:3
    #1 0x560c04b2bf88 in memory::main::hd2333c1899d997f5 $CWD/src/main.rs:6:16

  Uninitialized value was created by an allocation of 'a' in the stack frame of
function '_ZN6memory4main17hd2333c1899d997f5E'
    #0 0x560c04b2bc50 in memory::main::hd2333c1899d997f5 $CWD/src/main.rs:3
```

MemTagSanitizer

MemTagSanitizer detects a similar class of errors as AddressSanitizer and HardwareAddressSanitizer, but with lower overhead suitable for use as hardening for production binaries.

MemTagSanitizer is supported on the following targets:

- `aarch64-linux-android`
- `aarch64-unknown-linux-gnu`

MemTagSanitizer requires hardware support and the `mte` target feature. To enable this target feature compile with `-C target-feature="+mte"`.

More information can be found in the associated LLVM documentation.

ThreadSanitizer

ThreadSanitizer is a data race detection tool. It is supported on the following targets:

- `aarch64-apple-darwin`
- `aarch64-unknown-linux-gnu`
- `x86_64-apple-darwin`
- `x86_64-unknown-freebsd`
- `x86_64-unknown-linux-gnu`

To work correctly ThreadSanitizer needs to be "aware" of all synchronization operations in a program. It generally achieves that through combination of library interception (for example synchronization performed through `pthread_mutex_lock` / `pthread_mutex_unlock`) and compile time instrumentation (e.g. atomic operations). Using it without instrumenting all the program code can lead to false positive reports.

ThreadSanitizer does not support atomic fences `std::sync::atomic::fence`, nor synchronization performed using inline assembly code.

Example

```
static mut A: usize = 0;

fn main() {
    let t = std::thread::spawn(|| {
        unsafe { A += 1 };
    });
    unsafe { A += 1 };

    t.join().unwrap();
}
```

```
$ export RUSTFLAGS=-Zsanitizer=thread RUSTDOCFLAGS=-Zsanitizer=thread
$ cargo run -Zbuild-std --target x86_64-unknown-linux-gnu
==================
WARNING: ThreadSanitizer: data race (pid=10574)
  Read of size 8 at 0x5632dfe3d030 by thread T1:
    #0 example::main::_$u7b$$u7b$closure$u7d$$u7d$::h23f64b0b2f8c9484
../src/main.rs:5:18 (example+0x86cec)
    ...

  Previous write of size 8 at 0x5632dfe3d030 by main thread:
    #0 example::main::h628ffc6626ed85b2 /.../src/main.rs:7:14 (example+0x868c8)
    ...
    #11 main <null> (example+0x86a1a)

  Location is global 'example::A::h43ac149ddf992709' of size 8 at 0x5632dfe3d030
(example+0x000000bd9030)
```

Instrumentation of external dependencies and std

The sanitizers to varying degrees work correctly with partially instrumented code. On the one extreme is LeakSanitizer that doesn't use any compile time instrumentation, on the other is MemorySanitizer that requires that all program code to be instrumented (failing to achieve that will inevitably result in false positives).

It is strongly recommended to combine sanitizers with recompiled and instrumented standard library, for example using [cargo `-Zbuild-std` functionality].

Build scripts and procedural macros

Use of sanitizers together with build scripts and procedural macros is technically possible, but in almost all cases it would be best avoided. This is especially true for procedural macros which would require an instrumented version of rustc.

In more practical terms when using cargo always remember to pass `--target` flag, so that rustflags will not be applied to build scripts and procedural macros.

Symbolizing the Reports

Sanitizers produce symbolized stacktraces when llvm-symbolizer binary is in `PATH`.

Additional Information

- Sanitizers project page
- AddressSanitizer in Clang
- ControlFlowIntegrity in Clang
- HWAddressSanitizer in Clang
- LeakSanitizer in Clang
- MemorySanitizer in Clang
- ThreadSanitizer in Clang