Linux Kernel Modules in Rust

Alex Gaynor & Geoffrey Thomas

Alex & Geoff

Vulnerabilities due to memory unsafety are common, and preventable

Memory unsafety

- Use after free, double free, wild free
- Buffer overflow, buffer underflow, wild pointer
- Use of uninitialized memory
- Data races (often leading to one of the above)
- etc

49% - Chrome

Estimated 49% of Chrome security vulnerabilities in 2019 had memory unsafety as a root cause

72% - Firefox

Estimated 72% of Firefox security vulnerabilities in 2019 had memory unsafety as a root cause

81% - 0days

Estimated 81% of in the wild 0days (as tracked by Google Project Zero) since 2014 have memory unsafety as a root cause.

But what about kernel space?

88% - macOS

Estimated 88% of macOS kernel space vulnerabilities in the 10.14 series had memory unsafety as a root cause

70% - Microsoft

Estimated 70% of Microsoft vulnerabilities since 2006 had memory unsafety as a root cause

65% - Ubuntu

Estimated 65% of kernel CVEs in Ubuntu USNs in the last six months had memory unsafety as a root cause

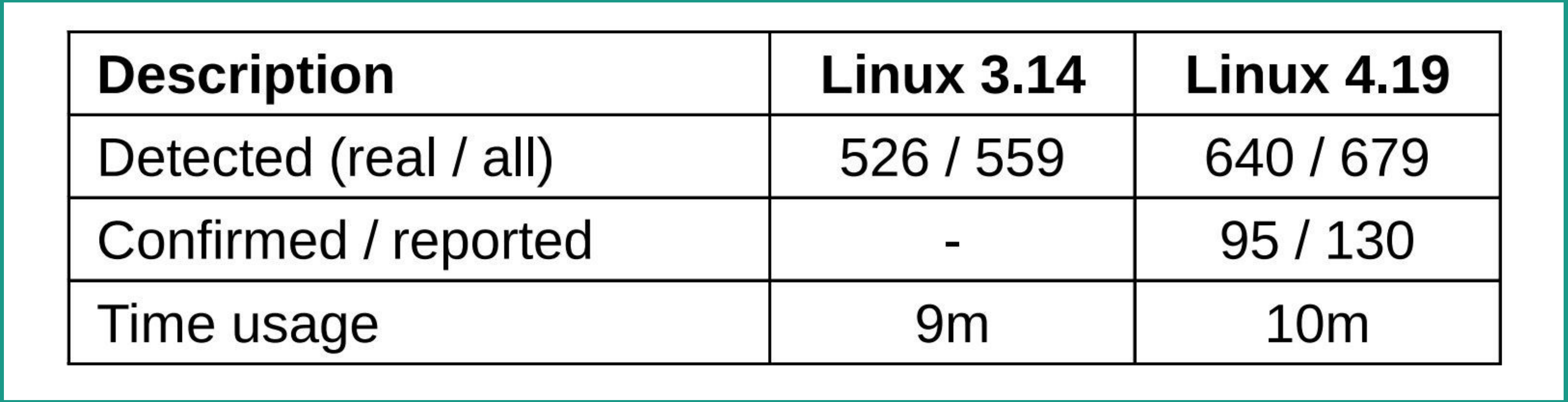
65% - Android

Estimated 65% of CVEs in Android from May 2017 to May 2018 had memory unsafety as a root cause

225 - Syzkaller

```
curl 'https://syzkaller.appspot.com/upstream' | \
    grep "K[AM]SAN:" | wc -l
```

UAF Static Analysis



Description	**Linux 3.14**	**Linux 4.19**
Detected (real / all)	526 / 559	640 / 679
Confirmed / reported	-	95 / 130
Time usage	9m	10m

These vulnerabilities have the same root cause: C and C++

So what are our options?

(or, why Rust?)

Hardening C

- ASLR
- Stack canaries
- Control flow integrity / Intel CET
- STACKLEAK
- sparse
- Coverity

Isolation

- WebAssembly
- eBPF
- ring 1
- microkernels

… at what cost?

From: Ingo Molnar <mingo@kernel.org>
Subject: Re: [RFC PATCH 2/7] x86/sci: add core implementation for system call isolation

To phrase the argument in a bit more controversial form:

If the price of Linux using an insecure C runtime is to slow down system calls with immense PTI-alike runtime costs, then wouldn't it be the right technical decision to write the kernel in a language runtime that doesn't allow stack overflows and such?

I.e. if having Linux in C ends up being slower than having it in Java, then what's the performance argument in favor of using C to begin with? ;-)

And no, I'm not arguing for Java or C#, but I am arguing for a saner version of C.

"a saner version of C"

From: Linus Torvalds
Subject: Re: Compiling C++ kernel module + Makefile
Date: Mon, 19 Jan 2004 22:46:23 -0800 (PST)

It sucks. Trust me - writing kernel code in C++ is a BLOODY STUPID IDEA.

- the whole C++ exception handling thing is fundamentally broken. It's _*especially*_ broken for kernels.

- any compiler or language that likes to hide things like memory allocations behind your back just isn't a good choice for a kernel.

- you can write object-oriented code (useful for filesystems etc) in C, _*without*_ the crap that is C++.

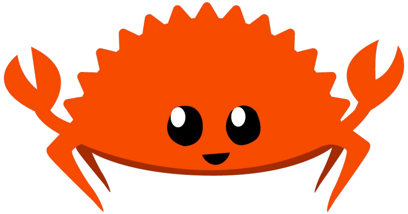
What do we want out of our language?

- Memory safety
- No unwind-based exception handling
- Simpler OO
- Don't "hide things like memory allocations behind your back"
- No garbage collector
- No runtime / thread manager
- Performant FFI to C / assembly

Good but unsuitable safe languages:

- Haskell: GC + runtime
- Go: GC + runtime + overhead for C calls
- D: GC
- Ada: static memory allocations

Rust



- Compiled language intended for systems programming
- Sponsored by Mozilla as a better / more secure language for Firefox (C++)
- Drop-in replacement for C for incremental rewrites
- Memory safety and thread safety
- No GC
- OS threading
- C-compatible calling convention

A whirlwind tour of Rust, focusing on safety

Hello world!

```rust
fn main() {
  let x: i32 = 10;
  println!("Hello world! x = {}", x);
}
```

Variables

```rust
fn main() {
  let x: i32 = 10;
  x = 5;
  println!("Hello world! x = {}", x);
}
```

Variables

```rust
fn main() {
  let mut x: i32 = 10;
  x = 5;
  println!("Hello world! x = {}", x);
}
```

Uninitialized variables

```
fn main() {
  let mut x: i32;
  println!("Hello world! x = {}", x);
  x = 5;
}
```

```
error[E0381]: borrow of possibly uninitialized variable: `x`
 --> src/main.rs:3:35
  |
3 |   println!("Hello world! x = {}", x);
  |                                   ^ use of possibly uninitialized `x`
```

Structs

```
struct Rectangle {
  length: f64,
  width: f64,
}

impl Rectangle {
  fn area(&self) -> f64 {
    self.length * self.width
  }
}
```

Traits

```
trait Shape {
  fn area(&self) -> f64;
  fn perimeter(&self) -> f64;
}

impl Shape for Rectangle {
  fn area(&self) -> f64 { self.length * self.width }
  fn perimeter(&self) -> f64 { 2.0 * self.length + 2.0 * self.width }
}
```

Generics and polymorphism

```
fn describe<T: Shape>(shape: &T) {
  println!("Area:      {}", shape.area());
  println!("Perimeter: {}", shape.perimeter());
}
```

Trait objects and runtime polymorphism

```
fn describe(shape: &dyn Shape) {
  println!("Area:      {}", shape.area());
  println!("Perimeter: {}", shape.perimeter());
}
```

Enums

```
enum OvercommitPolicy {
  Heuristic,
  Always,
  Never,
}

let overcommit_okay = match policy {
  OvercommitPolicy::Heuristic => size < heuristic_limit(),
  OvercommitPolicy::Always => true,
  OvercommitPolicy::Never => size < remaining_memory(),
}
```

Enums with data

```
enum Address {
  IP { host: IPAddress, port: u32 },
  UNIX { name: String },
  Raw,
}

match address {
  Address::IP { host, port } => ...,
  Address::UNIX { name } => ...,
  Address::Raw => ...,
}
```

Option and Result

```
enum Option<T> {
  None,
  Some<T>
}
if let Some(x) = potential_x
{
  ..
}
```

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

Error handling

```
foo?

Ok(foo)? ⇒
foo

Err(bar)? ⇒
{ return Err(From::from(bar)); }
```

```
fn read_data() -> Result<Data, Error> {
  let file = open("data.txt")?;
  let msg = file.read_to_string(...)?;
  let data = parse(msg)?;
  Ok(data)
}
```

Panics and unwinding

```
1/0

[3, 4, 5][10]

[3, 4, 5].get(10) == None

panic!("everything went wrong")
```

References, lifetimes, and the borrow checker

References

```rust
fn main() {
  let x: i32 = 10;
  let y: &i32 = &x;
  println!("y = {}", *y);
}
```

References

```rust
fn print(a: &i32) {
    println!("The value is {}", a);
}

fn main() {
  let x: i32 = 10;
  print(&x);
}
```

Dangling references

```
fn main() {
  let mut y: &i32;
  for i in 1..5 {
      y = &i;
  }
  println!("{}", y);
}
```

```
error[E0597]: `i` does not live long enough
 --> src/main.rs:4:11
  |
4 |       y = &i;
  |           ^^ borrowed value does not live long enough
5 |   }
  |   - `i` dropped here while still borrowed
6 |   println!("{}", y);
  |                  - borrow later used here
```

Mutable references

```rust
fn main() {
  let mut x: i32 = 5;
  let y: &i32 = &x;
  *y = 10;
}
```

```
error[E0594]: cannot assign to `*y` which is behind
a `&` reference
 --> src/main.rs:4:3
  |
3 |   let y: &i32 = &x;
  |                 -- help: consider changing this
to be a mutable reference: `&mut x`
4 |   *y = 10;
  |   ^^^^^^^ `y` is a `&` reference, so the data
it refers to cannot be written
```

Mutable references are unique references

```
fn main() {
  let mut x: i32 = 5;
  let y: &mut i32 = &mut x;
  let z: &i32 = &x;
  *y = 10;
}
```

```
error[E0502]: cannot borrow `x` as immutable
because it is also borrowed as mutable
 --> src/main.rs:4:17
  |
3 |   let y: &mut i32 = &mut x;
  |                     ------ mutable borrow
occurs here
4 |   let z: &i32 = &x;
  |                 ^^ immutable borrow occurs
here
5 |   *y = 10;
  |   ------- mutable borrow later used here
```

Safe abstractions for unsafe code

Atomics

```rust
use std::sync::atomic::*;

let x = AtomicU32::new(1);
let y = &x;
let z = &x;
y.store(3, Ordering::SeqCst);
println!("{}",
  z.load(Ordering::SeqCst));
```

```rust
struct AtomicU32 {
  v: UnsafeCell<u32>
}

impl AtomicU32 {
  fn store(&self,
           val: u32,
           order: Ordering) {
    unsafe { atomic_store(self.v.get(),
      val, order) }
  }
}
```

Safe and unsafe Rust

```rust
fn zero(x: *mut u8) {
  unsafe { *x = 0; }
}

unsafe fn zero(x: *mut u8) {
  *x = 0;
}
```

```rust
fn main() {
  let mut x = vec![3u8, 4, 5];
  let p = &mut x[0];
  unsafe { zero(p); }
  println!("{:?}", x);
}
```

FFI: calling C from Rust

```
extern {
  fn readlink(path: *const u8, buf: *const u8, bufsize: usize) -> i64;
}

fn rs_readlink(path: &str) -> Result<String, ...> {
  let mut r = vec![0u8; 100];
  if unsafe { readlink(path.as_ptr(), r.as_mut_ptr(), 100) } < 0 {
    Err(...)
  } else {
    Ok(String::from_utf8(r)?)
  }
}
```

FFI: calling Rust from C

```rust
#![no_mangle]
extern fn add(x: u32, y: u32) -> u32 {
  x + y
}
```

```c
uint32_t add(uint32_x, uint32_y);

int main(void) {
  printf("%d\n", add(10, 20));
}
```

FFI: types

```
#[repr(C)]
struct Sigaction {
  sa_handler: extern fn(c_int),
  sa_flags: c_int,
  ...
}
extern {
  fn sigaction(signum: c_int,
    act: *const Sigaction,
    oldact: *mut Sigaction);
}
```

```
extern fn handler(signal: c_int) {...}

let act = Sigaction {
  sa_handler: handler,
  ... }
unsafe {
  sigaction(SIGINT, &act, ptr::null_mut())
}
```

Incrementally "oxidizing" C

What we've built so far

Kernel modules

```
struct HelloWorldModule;
impl KernelModule for HelloWorldModule {
    fn init() -> KernelResult<Self> {
        println!("Hello world!");
        Ok(HelloWorldModule)
    }
}
kernel_module!(HelloWorldModule, license: "GPL");
```

Compiling

```
$ cargo xbuild --target x86_64-linux-kernel-module.json
$ make
```

```
obj-m := helloworld.o
helloworld-objs :=
target/x86_64-linux-kernel-module/debug/libhello_world.a
KDIR ?= /lib/modules/$(shell uname -r)/build
all:
        $(MAKE) -C $(KDIR) M=$(CURDIR)
```

Bindings

- printk
- error types
- kmalloc/kfree
- register_sysctl
- register_filesystem
- alloc_chrdev_region
- copy_from_user / access_ok

Mapping kernel APIs to Safe Rust

Box/Vec/String

- Box: Basically std::unique_ptr
- Vec: Heap-based growable linear array
- String: Linear sequence of utf-8 encoded code points

GlobalAlloc

```
pub struct KernelAllocator;

unsafe impl GlobalAlloc for KernelAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // krealloc is used instead of kmalloc because kmalloc is an inline function and can't be
        // bound to as a result
        return bindings::krealloc(ptr::null(), layout.size(), bindings::GFP_KERNEL) as *mut u8;
    }

    unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
        bindings::kfree(ptr as *const c_types::c_void);
    }
}
```

Heap allocations just work

```
struct HelloWorldModule {
    message: String,
}

impl linux_kernel_module::KernelModule for HelloWorldModule {
    fn init() -> linux_kernel_module::KernelResult<Self> {
        println!("Hello kernel module!");
        Ok(HelloWorldModule {
            message: "on the heap!".to_owned(),
        })
    }
}
```

What about __user pointers?

Desired goals:

- Type safe
- Always bounds checked
- No double fetches

UserSlicePtr

```rust
impl UserSlicePtr {
    pub fn read_all(self) -> error::KernelResult<Vec<u8>>

    pub fn reader(self) -> UserSlicePtrReader

    pub fn write_all(self, data: &[u8]) -> error::KernelResult<()>

    pub fn writer(self) -> UserSlicePtrWriter
}
```

```
fn read(
    &self,
    buf: &mut UserSlicePtrWriter,
) -> KernelResult<()> {
    for c in b"123456789".iter().cycle().take(buf.len()) {
        buf.write(&[*c])?;
    }
    return Ok(());
}
```

Concurrency!

Rust models concurrency with two traits: **Sync** & **Send**:

- **Sync**: Multiple threads may have references to values of this type
- **Send**: Type may transfer ownership to a different thread

Lots of kernel types need safe concurrent access!

FileOperations must be Sync!

```
pub trait FileOperations: Sync + Sized {
    const VTABLE: FileOperationsVtable;

    fn open() -> KernelResult<Self>;
    fn read(&self, buf: &mut UserSlicePtrWriter) -> KernelResult<()>;
}
```

bindgen and libclang

Architecture support

- x86
- arm/arm64
- mips
- powerpc
- riscv
- s390
- sparc
- um?

LLVM backend

minimal Rust support

mrustc / LLVM CBE

https://github.com/fishinabarrel/linux-kernel-module-rust/issues/112

Future directions!

The future is very bright!

- More kernel APIs
- Support existing out of tree module authors (upstream kernel developers: insert boos here!)
- Better kbuild integration

More kernel APIs

Expand beyond

- chrdevs
- sysctls

Exciting targets:

- Filesystems
- Drivers for particular device classes

Real world out-of-tree module usage?

- What would it take for you to use this?
- We’d love to find a way to support you!

Better kbuild integration

```
$ cargo xbuild --target $(pwd)/../x86_64-linux-kernel-module.json
$ make
$ sudo insmod helloworld.ko
```

What would it take to have first-class support for writing modules in Rust in-tree?

Q & A

https://github.com/fishinabarrel/linux-kernel-module-rust

Follow us for vulnerability statistics: @LazyFishBarrel