Rewriting Linux's Key Management Subsystem in Rust

Matthew Denton, Stanford University

Rust is a newer language that describes itself as "a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety." With the number of major security bugs affecting the Linux kernel, including remote code execution and local privilege escalation bugs, as well as the even larger number of usability bugs, Rust looks like a good candidate to replace C as the language of the kernel. In this report, we examine the possibility of rewriting the Linux key management subsystem in Rust, in order to explore the potential safety gained by using Rust.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Implementation*

General Terms: Rust

Additional Key Words and Phrases: Rust, local privilege escalation, remote code execution, key management, bounds checking, thread-safety

1. INTRODUCTION

Linux is the most prevalent operating system kernel in the world; in particular, the large majority of the server market runs Linux, including servers run by Amazon, Facebook, Google, Akamai, Twitter, and many more. Even Android, the most popular phone operating system in the world, is based on Linux.

However, Linux continues to run into a host of security issues. This is in large part due to the language Linux is written in, C, which was created in 1972 and has largely remained unchanged since. C was designed to be a thin wrapper around assembly, and as such allows common programming errors such as use-after-free, double-free (a special type of use-after-free), writing to buffers out of bounds, and race conditions. Each of these can often be exploited as a security hole, as described in future sections.

Furthermore, C suffers from a lack of type annotations and enforced safety from the type system. As a simple example, programmers can access unions using any of the available fields, and may accidentally use one field, when a variable of a different type was assigned to a different field in that union. As another example, userspace pointers carry no additional information in their type to notify the kernel developers that they are dereferencing a userspace pointer, which may be dangerous within the kernel depending on the context.

Work has been done on tooling, such as the static analysis tool Sparse. For the aforementioned userspace pointer dereferencing problem, much of the C code contains the ___user annotation for userspace pointers, which Sparse can understand and output possible programming errors. Unfortunately, most static analysis tools cannot completely reliably detect errors, due to the unconstrained nature of programs written in C, and thus must choose between a policy of false positives and a policy of false negatives. Unfortunately, displaying false positives discourages the programmers from running the tool, let alone looking through the error messages, and thus tools will often choose false negatives, which allows security errors to remain in the code. In addition, Sparse works specifically for the Linux kernel, and as such cannot benefit from improvements to static analysis tools meant for general programming work.

Rust promises to solve many of the issues, and restrict possible security vulnerabilities to logic errors, and errors in Rust's language implementation and code generation–a much smaller attack surface, tested regularly by far more users than just kernel developers. Rust has a powerful type system that can be used to encode far more specific constraints than the typical C constraints, and the borrow checker promises to avoid use-after-free and race conditions. In fact, many static analysis and runtime sanitizers can be completely eliminated in favor of Rust's borrow system.

With this in mind, it would seem advantageous to write an entirely new operating system in Rust, with the same system call interface, and replace Linux with it. And in fact, many projects have been started in order to do just that. These include Redox, a full-fledged operating system with graphics, windows, filesystems, mouse and keyboard support, x86_64 support, and more. However, these operating system could not possibly replace Linux for at least a decade: Linux has a vast number of drivers which will simply never be rewritten in Rust. At the moment, despite the excitement of the Rust community and a lot of active development, Redox cannot boot on most real hardware due to support for disks with multiple partitions. Not to mention that there are thousands of kernel developers very intimately educated in the workings of the Linux system as well as the C programming language–moving them to a new operating system in a new language would be a waste of their expertise, and companies would not see the rapid improvements they want in their kernels.

Thus, the only way to gain any of Rust's advantages is to incorporate Rust into the Linux kernel, piece by piece. It remains to be seen whether this is truly plausible, and whether Rust's abstractions and type systems can provide any extra security in a system that still involves mostly C code.

In this project I attempted to rewrite a (small subset) of Linux's key management subsystem. The key management subsystem provides other subsystems, as well as user space, the ability to store encryption keys, or any blobs of binary data, within kernel memory. This aids with implementing distributed or encrypted filesystems within the kernel, as well as many other systems.

2. SYSTEM SECURITY BACKGROUND

2.1 Common Linux Kernel Exploitation Techniques

In this project I primarily focused on the local privilege escalation threat model, though many of the techniques described apply to the remote exploitation threat model as well, and indeed many successful remote exploits begin with a userspace exploit, and end with a local privilege escalation to finally gain full privileges.

Common exploitation techniques of regular C programs include

(1) stack buffer overflows, which allow attackers to overwrite important stack data such as saved return addresses stored on the stack
(2) heap overflows, which may allow an attacker to overwrite adjacent heap objects, which may include user credentials, access permissions, or function pointers (in order to jump to attacker-controlled code).
(3) use-after-free, which allows an attacker to coax the heap allocator into allocating an object in the same position as the dangling pointer, and writing data (like access permissions or code pointers) into the object so that vulnerable code that accesses the dangling pointer will be fooled into using attacker-controlled data
(4) race conditions, which often allow an attacker to trigger one of the above vulnerabilities

Typical exploits work by by hijacking the instruction pointer, and pointing it towards desired code that will perform the final exploit. Thus, there are normally two focuses for defenses: (1) prevent hijacking of the instruction pointer, and (2) preventing the attacker from directing the instruction pointer to useful exploit code.

Rust excels at (1) by preventing out-of-bounds access and use-after-free vulnerabilities. (2) is partially solved in that attackers can almost never inject arbitrary code in the program due to the NX (no-execute) bit present in most CPU page tables. Thus, attackers are normally constrained to running existing code in the program, by repurposing collections of short instruction sequences (often called gadgets) into exploit code. For example, if an attacker controls the stack due to an out-of-bounds overflow, they may write a sequence of return addresses to the stack, and point each return address at a useful gadget that is followed by a "ret" instruction. Therefore, due to the stack setup, the gadgets will return from one to the other, chaining into useful exploit code. A common defense against this so-called Returned Oriented Programming

(ROP) is known as Address Space Layout Randomization (ASLR), which randomizes the starting address of the code so that the attacker cannot insert correct return addresses onto the stack. In the kernel, this defense is known as KASLR. There are other defenses–such as control flow integrity (CFI), which generally work by ensuring that returns cannot jump to code that isn't specified by the original control flow graph, and thus the attacker cannot execute code out of the intended order.

Unfortunately, CFI is generally too much of a performance hit for most distributions to enable it in their kernel configurations, especially when security isn't of the utmost importance (such as offline data-crunching machines or high performance computing). Thus, KASLR is often the only defense against ROP and other code-reuse attacks.

A method of breaking KASLR is to leak kernel pointers to userspace. With access to a known kernel pointer, an attacker can calculate the starting address of the kernel, and with knowledge of the kernel's code layout (easily obtained by examining the kernel version and the Linux distribution's configuration), one can calculate offsets to any code.

Common methods of obtaining these kernel pointers include:

(1) Leaking uninitialized data to userspace. Oftentimes this uninitialized data (on either the stack or the heap) will contain a kernel pointer, especially if the attacker has managed to "groom" the stack or the heap to contain relevant kernel pointers with high probability. In addition, if the uninitialized data does not contain anything that resembles a kernel pointer, the attacker can simply rerun the exploit and try again. Rust effectively prevents this attack by disallowing all uninitialized values–though in interoperation with C code, uninitialized data from buggy C code can still be present in Rust data.
(2) Forgetting to check pointers passed from userspace through system calls. For example, if a user program passes a kernel pointer to a system call, and the kernel reads the data, and copies it back to userspace in some way, then the userspace program has obtained leaked kernel data. Thus, the kernel must call access_ok() on every userspace pointer to ensure it is actually within the bounds of the user program's address space, and not the kernel's.

2.2 Userspace Pointers in System Calls

As mentioned above, pointers passed by user programs must be checked as valid userspace pointers, lest they be used to leak KASLR information. However, it is even more essential to check user pointers in the case that we are *writing* to the address: if we give user space programs the ability to instruct the kernel to write to an arbitrary kernel address, this could break the entire security of the kernel. For example, the attacker could write to their "credentials" struct and set their credentials to "root."

In addition, the Linux kernel may be vulnerable to race conditions if the kernel reads multiple times from a user pointer. For example, consider the following system call pseudocode:

(1) Read file descriptor number from user pointer ptr.
(2) Determine file descriptor number is within the bounds of the file descriptor table
(3) Dereference ptr (without access checks as those are now unnecessary, as they already occurred), and use it to index into the file descriptor table.

If another user thread gets scheduled and changes the memory pointed to by ptr in between steps 1 and 3, the kernel will access the file descriptor table out of bounds in step 3, despite the bounds check in step 1 passing. These types of bugs are called Time of Check vs. Time of Use (TOCTOU) bugs.

Thus the kernel must copy data from user space, and use its local copy for the rest of the system call, and in general it should not dereference the same userspace pointer more than once per system call.

This procedure causes even more of a performance hit on accessing user data–and thus, many kernel developers will try to regain performance by omitting redundant userspace pointer checks. For example,

developers will often run a single access_ok() check on an entire range, and call unsafe_get_user and unsafe_put_user functions that do not perform the access_ok() check. This can often lead to developers forgetting to call access_ok() on certain sections of memory they dereference.

2.3 Recent Linux Vulnerabilities

CVE-2017-5123 is an example of missing access_ok() checks in the waitid() system call. The code needed to put many different integer values into a userspace struct, so the goal was to use access_ok() checks on the entire struct and then calling unsafe_put_user to copy all of the relevant integer values. However, the access_ok() check was forgotten, and thus userspace programs were able to write arbtirary values to the kernel's address space.

CVE-2017-15951 is an example of a race condition within the key management subsystem–proper synchronization was not enforced to ensure that when reading an error message from the key's payload, that the key did not become instantiated with arbitrary data, allowing an attacker to read uninitialized data and potentially leak information to userspace.

CVE-2017-14954 is also an example of a leak of uninitialized memory in the waitid system call, exposing many kernel pointers.

CVE-2017-7308 is a bug in the bounds checking of raw packet sockets in Linux that allows an attackers to overwrite data into adjacent heap structs.

3. RUST AND LINUX INTEROPERABILITY

3.0.1 *No Standard Library: Building for Bare Metal.* The first difficulty with compiling Rust code into Linux is the reliance of Rust code on the standard library. The standard library relies on compiling for a target–including an operating system. It does not work without a library of system calls, and often a system specific library such as libc.

Luckily, the Rust developers have tackled this issue, creating the "core" library, which does not depend on any pre-existing components on the target system, but provides basic types and traits. This still raises a couple issues, however:

(1) We cannot use any crates that rely on anything in std, signifcantly constraining our usage of the Rust ecosystem.

(2) No allocation–which means no Box, Arc, Vector, or String, which are often considered a basic part of Rust.

(3) No collections, due to the lack of allocation support. This makes it difficult to write useful programs.

(4) Core includes floating point code, which is incompatible with the kernel.

(5) Core actually has some dependencies. Specifically, it relies on there being existing memcpy, memset, and memmove functions. These are normally provided by libc, which is not available. However, this was solved by including the rlibc crate, which implements these functions with pure-Rust code.

We disable std and re-enable core by including these annotations at the top of our main lib.rs module:

```
#![no_std]
#![feature(core)]
```

Note that "no_std" is standardized, but "core" is still unstable, and thus is enabled above as an "unstable feature". This requires the Rust nightly compiler, which can be installed with rustup and enabled globally or on a per-directory basis.

3.0.2 *Generating C Bindings for Rust.* In this project, calling Rust code from C was simple enough, as our Rust code was far shorter and less established than our C code.

```
#[no_mangle]
pub extern fn rust_hello() {
        println!("Hello from Rust!!!!");
}
```

We simply enable the no_mangle attribute to avoid Rust's name mangling, and declare the function as pub and extern, which allows the symbol to be called by C. Then the C code

```
extern void rust_hello(void);
int main() {
    rust_hello();
}
```

works as desired when linked with the Rust library.

Calling C code from Rust was more challenging.

Each C structure and function must have an equivalent type signature in Rust, so that the Rust compiler can generate appropriate accesses to struct and union fields, and can check function argument types and generate the correct function calls

As the kernel is vast, we needed to autogenerate the signatures of the function calls and structures. And since this C code may change between kernel releases (there are no ABI guarantees for the kernel), we must incorporate this binding generation into the build system.

To this end, we used rust-bindgen, which utilizes clang's parser to build the equivalent Rust definitions for the C definitions, and a Cargo build script to autogenerate these bindings, but only when our list of header files has changed (as binding generation is very slow).

This is an excerpt of our build script build.rs, kept in our Cargo project's root directory:

```
extern crate bindgen;
extern crate shlex;

use std::env;
use std::path::PathBuf;

fn main() {
    // tell cargo to only rerun the build script if linux_headers.h changes...
    println!("cargo:rerun-if-changed=linux_headers.h");

    // change to kernel working directory, save current wd first
    let curr_wd = env::current_dir().unwrap();
    let k_dir = PathBuf::from(env::var("STD_KERNEL_PATH").unwrap());
    assert!(env::set_current_dir(&k_dir).is_ok());

    let clang_args = match std::env::var("STD_CLANG_ARGS") {
        // code omitted for brevity
    };
    let bindings = bindgen::Builder::default()
        .rust_target(bindgen::RustTarget::Nightly)
        .derive_copy(false) // used because deriving copy for things like spinlock would be unsafe.
```

```
        .derive_debug(false)
        .use_core()
        .ctypes_prefix("c_types")
        .clang_arg("-Dfalse=__false")
        .clang_arg("-Dtrue=__true")
        .clang_arg("-Du64=__u64")
        .clang_args(clang_args.iter())
        .opaque_type("timex")
        .header(curr_wd.join("linux_headers.h").to_string_lossy())
        // Finish the builder and generate the bindings.
        .generate()
        // Unwrap the Result and panic on failure.
        .expect("Unable to generate bindings");

    // change back to old working directory (rust directory)
    assert!(env::set_current_dir(&curr_wd).is_ok());

    // Write the bindings to the $OUT_DIR/bindings.rs file.
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't write bindings!");
}
```

There are a number of important aspects of this build script. The first is that it must run in the kernel's root directory in order to correctly generate bindings, but must write the bindings in the Cargo directory.

The list of headers to generate bindings for is kept in "linux_headers.h". This build script only runs when that file changes to avoid long compilation times. In addition, the kernel compiler's command line arguments are passed to rust-bindgen, so it can invoke clang with those flags. Among other things, these flags will specify "include" directories for the compiler to look for headers in. This is crucial to allowing clang to find headers referenced recursively by the headers we are generating bindings for–we can hardcode relative header paths in "linux_headers.h", but those headers will in-turn include other headers which will then not have the correct relative paths, so we must tell the compiler the correct directories to look for headers.

We also instruct rust-bindgen to use core instead of std, avoid deriving Copy and Debug for types that can't support it (like spinlocks), and to generate for Rust nightly. We also list "timex" as an opaque type (no binding generated), as a rust-bindgen bug prevent the autogenerated binding from compiling due to a strange use of C's bitfields. Finally, we instruct bindgen to use "c_types" as the prefix for c types, as the default uses std. What follows is a short excerpt of "c_types.rs":

```
pub type c_short      = i16;
pub type c_ushort     = u16;

pub type c_int        = i32;
pub type c_uint       = u32;
```

The unfortunate downfall of generating bindings, and calling C code in general, is the kernel's rampant use of complicated macros and inlined functions. As these constructs normally will not have symbols in the final object files, the Rust code cannot link against these functions, and rust-bindgen does not create bindings for these functions. This makes interoperation a little more difficult than was have been expected.

Luckily, however, rust-bindgen does generate constants for define'd constants, which often makes it possible to reimplement some of the simpler macro's in Rust.

And finally, here is an excerpt of "os.rs", which is to contain our final bindings. Note that many warnings have to be disabled to tolerate the generated bindings, including improper_ctypes, due to kernel configs causing the generation of empty structs:

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
#![allow(dead_code)]

// allow improper c_types because due to the kernel config there can
// be empty structs, like lock_class_key if lock_dep is disabled.
#![allow(improper_ctypes)]

// This is so the bindings can generate unions with "non-Copyable" structs in them
#![feature(untagged_unions)]

use c_types;

include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

3.0.3 *Allocation.* The first step to regaining many of Rust's useful features is by implementing our own allocator and setting it as the global default. This not only regains Box, Arc, Vector, and String, but also regains the rest of Rust's collection as well! We enable these (unstable) language features as such

```
#![feature(alloc)]
#![feature(global_allocator)]
#![feature(allocator_api)]
```

What follows is our implementation of the Rust allocator:

```
use alloc::heap::{Alloc, AllocErr, Layout};
use os;
use c_types;

pub struct LinuxAllocator {}

unsafe impl<'a> Alloc for &'a LinuxAllocator {
        unsafe fn alloc(&mut self, layout: Layout) -> Result<*mut u8, AllocErr> {
                let ptr = os::krealloc(0 as *const c_types::c_void, layout.size(), os::GFP_KERNEL)
                    as *mut c_types::c_uchar;
                if ptr.is_null() {
                        return Err(AllocErr::Exhausted{request:layout});
                }
                return Ok(ptr);
        }

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

```
        }
}
```

In the kernel, the function to allocate memory is kmalloc. However, kmalloc is an inlined function, and thus is not available in our generated bindings. As the function is short, the functionality could be reimplemented in our alloc function, but reimplementation leads to issues when one of the implementations changes. Here, we take advantage of the fact that krealloc is not inlined, and calling krealloc on a NULL ptr is defined to perform the same function as kmalloc. This is of course not a general workaround for the issue of calling inlined C functions.

In any case, our implementation of the allocator above successfully wraps the kernel's slab allocator for usage in Rust by collections and other allocating structures.

Another possibility for implementing this allocator included using a Rust-only allocator. We could use ralloc, a Rust-only allocator from the Redox project, with thread local caches and more. However, this would require us to give ralloc access to either the kernel's page allocator, or kmalloc'ing a massive part of memory to give to ralloc to control, but this may be an inefficient use of physical memory. We could vmalloc a range, which allows the kernel to allocate a virtually contiguous range and allocate physical pages on demand, but this comes with a performance impact.

Even more importantly–with a pure-Rust allocator, any kmalloc'd objects passed to Rust by a C function (thus giving the Rust code "ownership") must be manually freed, or have custom Drop traits implemented in order to call kfree on that object rather than our pure-Rust free. Wrapping the kernel allocator allows us to obtain kmalloc'd objects from the C code and wrap a Vec or String around them, allowing the Vec to take full ownership of the object and automatically free the allocated memory when the Vec goes out of scope. The power granted by this mechanism cannot be understated. It allows Rust's safety to reach further than just its internal code, and allows us to use standard containers with objects allocated by C.

In addition, allocators are not common sources of vulnerabilities–most vulnerabilities have already been triggered by the time the code has reached the allocator. Allocators do not directly take in any "user data" from external sources (network packets) or from userspace (system call arguments), and if they do take in user data, the caller is typically in charge of properly sanitizing the data anyway. Thus, the allocator is actually a good candidate for remaining as C code for its speed advantages.

We enable this allocator as the global allocator by inserting this code (using unstable feature global_allocator) in our main lib.rs code:

```
#[global_allocator]
static MY_ALLOCATOR: LinuxAllocator = LinuxAllocator {};
```

3.1 Rust Compilation

There are a number of additional challenges in compiling Rust code for the kernel context. Both stem from the possibility of hardware interrupts in kernel mode.

First, floating point cannot be used in the Linux kernel. The reason for this that during hardware interrupts, the kernel interrupt handler does not want to save floating point registers to the stack and restore them on interrupt end (to avoid them being clobbered), because there are a huge number of large floating point registers and this would be a significant performance cost. Thus a restriction imposed on the kernel is that no code can use the floating point registers or instructions.

Rust's core library makes extensive use of the floating point operations.

In addition, the System-V x86_64 ABI specifies a red-zone of 128 bytes on the stack, specifying that nothing in the system (such as signal handler stacks or runtime mechanisms) will clobber anything up to 128 bytes above the top of the stack, which allows programs (especially leaf functions) to use the stack without incrementing %rsp, as an optimization.

However, the CPU's interrupts will push directly to the %rsp, and thus we cannot rely on the red-zone remaining untouched, and so we must instruct Rust not to generate code that uses the red-zone.

We solve most of these issues with a new target specification for "bare-metal", "x86_64-unknown-none-gnu.json" (the default for linux x86_64 is "x64_64-unknown-linux-gnu.json").

```
{
  "llvm-target": "x86_64-unknown-none",
  "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
  "linker-flavor": "gcc",
  "target-endian": "little",
  "target-pointer-width": "64",
  "target-c-int-width": "32",
  "arch": "x86_64",
  "os": "none",
  "features": "-mmx,-sse,-sse2,-sse3,-ssse3,-sse4.1,-sse4.2,-3dnow,-3dnowa,-avx,-avx2,+soft-float",
  "disable-redzone":         true,
  "eliminate-frame-pointer": true,
  "linker-is-gnu":           true,
  "no-compiler-rt":          true,
  "panic-strategy":          "abort",
  "archive-format":          "gnu",
  "code-model":              "kernel",
  "relocation-model":        "static"
}
```

Importantly, the "features" section removes Rust auto-vectorization options that utilize the floating point registers for SIMD operations, and enable soft-float, which emulates any floating point code with normally integer operations. And, of course, "disable-redzone" disables the compiler's use of the redzone. The "panic-strategy" being set to "abort" allows us to avoid linker errors due to a lack of stack unwinding support on our target.

Rust's "core" library only comes in precompiled format, so we need to install the Rust source and switch to the "Xargo" tool, which is a Cargo wrapper that will automatically "cross-compile" the core library to our new target.

There are a few more changes necessary to resolve linker errors. One is to enable the unstable compiler_builtins_lib feature and include the pure-Rust compiler_builtins crate, to include most of the intrinsics that LLVM relies on to compile code. Some of the floating point intrinsics are not available, so we provide stub implementations of these function in our main lib.rs, because we don't use floating pointer anyway. Finally, we enable the unstable lang_items feature and include the following in order to provide stub implementations of functions that are expected by the Rust compiler:

```
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] #[no_mangle] pub extern fn panic_fmt() -> ! {loop{}}
```

3.2 Linux Build System

The Linux build system is known as KBuild, and is comprised of a series of recursive Makefiles and special Makefile conventions.

The following is a security/keys/Makefile:

```
obj-y := \
        gc.o \
        key.o \
        keyring.o \
        keyctl.o \
        permission.o \
        process_keys.o \
        request_key.o \
        request_key_auth.o \
        user_defined.o

obj-$(CONFIG_PROC_FS) += proc.o
obj-$(CONFIG_SYSCTL) += sysctl.o
obj-$(CONFIG_PERSISTENT_KEYRINGS) += persistent.o
obj-$(CONFIG_ENCRYPTED_KEYS) += encrypted-keys/

# Some lines omitted for brevity

obj-y += keys_rs/
```

Each CONFIG_* constant is defined in the Linux configuration file, and has values "y" (include feature), "n" (don't include), or "m" (include as an optional module, added on demand at runtime). For our purposes, we set all optional configs to "n" in order to limit the scope of the project and make it manageable.

Anything listed under the variable obj-y will be compiled according to the specified compilation rule for the object. Since most of the object files have no specified compilation rules, they are assumed to be composed of only their corresponding .c files.

The only edit made to this file is the last line, which instructs the Makefile to recursively invoke the Makefile within keys_rs, which holds our Rust files.

The following is our custom security/keys/key_rs/Makefile:

```
# adapted from from https://github.com/tsgates/rust.ko/blob/master/kbuild.mk

# Tell kbuild which files to build
obj-y                 := keys-rust.o
keys-rust-objs        := libkeys.a

# Strip unused symbols from the input object file
EXTRA_LDFLAGS += --gc-sections --entry=init_module --undefined=cleanup_module
EXTRA_LDFLAGS += $(if ${RELEASE},--strip-all)
CARGO = `which xargo`

RUST_FILES := $(wildcard src/*.rs)
RCFLAGS =

# Determine target directory of cargo's module build
CARGO_MOD_DIR := $(src)/target/${UTS_MACHINE}-unknown-none-gnu/$(if ${RELEASE},release,debug)
# Determine target directory of cargo's build script build
CARGO_BLD_DIR := $(src)/target/$(if ${RELEASE},release,debug)
```

```
$(obj)/libkeys.a: ${RUST_FILES} FORCE
	cd $(src) && env STD_CLANG_ARGS='${c_flags}' STD_KERNEL_PATH='${CURDIR}' \
	    STD_CLANG_FILES='${KERNEL_INCLUDE}' "${CARGO}" \
	    rustc $(if ${RELEASE},--release) $(if ${V},--verbose) ${CARGOFLAGS} \
	    --target="${UTS_MACHINE}-unknown-none-gnu" -- ${RCFLAGS}
	cp "${CARGO_MOD_DIR}/libkeys.a" $(obj)
```

Note that the "obj-y" variable contains keys-rust.o. We also define the variable keys-rust-objs, which contains "libkeys.a" to instruct KBuild that keys-rust.o should contain all the symbols from the static library libkeys.a. The custom build command for libkeys.a is included at the bottom.

EXTRA_LDFLAGS includes the –gc-sections option, which is supposed to optimize away unused code, including code that may be invalid in the kernel, such as floating point code. However, it seems not to be obeyed by the kernel's linking system.

The custom build target for libkeys.a has a number of important parameters. For example, STD_CLANG_ARGS contains the compiler flags in use by the Makefile, from the c_flags variable. This allows our build script, and rust-bindgen, to generate correct bindings for the C functions. It also includes the correct bare-metal target. Finally, it copies the final static library generated by rustc into the directory that KBuild expects the compiled objects to reside.

This Makefile setup allows us to compile our Rust code as part of the kernel build, and link it successfully into the final kernel.

4. USER POINTER DESIGN

Recall that incorrect usage of user pointers has caused multiple security issues. So, we leverage the Rust type system to provide safe alternatives to user pointers.

4.1 Parameterized Structs for User Pointers

Here is the user pointer design for primitives types:

```
pub struct UserRWPtr<T>(*mut T);
pub struct UserWPtr<T>(*mut T);

impl<T> UserRWPtr<T> {
	pub fn new(x : *mut T) -> UserRWPtr<T> {
		UserRWPtr(x)
	}
	pub fn read(self) -> Result<(T, UserWPtr<T>), isize> {
		unsafe {
			let mut ret : T = uninitialized();
			let res = os::copy_from_user(&mut ret as *mut T as *mut c_types::c_void,
				self.0 as *mut c_types::c_void, size_of::<T>());
			if res > 0 {
				Err(-os::err_no(os::EFAULT))
			} else {
				Ok((ret, UserWPtr::<T>(self.0)))
			}
		}
	}
}
```

```
        pub fn write(&self, x : T) -> Result<(), isize> {
                write_primitive(self.0, x)
        }
}
impl<T> UserWPtr<T> {
        pub fn write(&self, x : T) -> Result<(), isize> {
                write_primitive(self.0, x)
        }
}
#[inline]
fn write_primitive<T>(dest : *mut T, x : T) -> Result<(), isize>  {
        unsafe {
                let res = os::_copy_to_user(dest as  *mut c_types::c_void,
                        &x as *const T as *const c_types::c_void, size_of::<T>());
                if res == 0 {
                        Ok(())
                } else {
                        Err(-os::err_no(os::EFAULT)) // number of bytes left to copy
                }
        }
}
```

The implementation utilizes Rust generics to implement a user pointer type for every primitive types and in fact this implementation works for any type known at compile time. I originally implemented these types using macros, but using macros to auto-generated structs and signatures has disadvantages–for example, certain syntax features don't work, the structs may not be easily searchable within the code base, and just don't have as much stability and support and type-checking as functions that take generic parameters.

The advantages are numerous–the unsafe code is contained to this module (including mem::uninitialized), the finicky C-interoperation is contained, and the error checking is contained. It is impossible to dereference this pointer manually, so one cannot accidentally dereference the pointer without checking. In addition, the read function consumes the UserRWPtr (making it unusable after the first read call) and only returns a UserWPtr, which has no read function–this prevents TOCTOU bugs caused by reading multiple times from user space.

We also provide an implementation of a user pointer to a string:

```
pub struct UserRPtrStr(*const c_types::c_char);
impl UserRPtrStr {
        pub fn new(x : *const c_types::c_char) -> UserRPtrStr {
                UserRPtrStr(x)
        }
        pub fn read(self, n: usize) -> Result<Vec<i8>, isize> {
                unsafe {
                        let kptr = os::strndup_user(self.0, n as c_types::c_long);
                        // equivalent to IS_ERR fn in Linux
                        if (kptr as c_types::c_ulong) > ((-(os::MAX_ERRNO as c_types::c_long))
                            as c_types::c_ulong) {
                                Err(kptr as isize)
                        } else {
```

```
                                let str_length = os::strnlen(kptr, n as c_types::c_size_t);
                                Ok(Vec::from_raw_parts(kptr as *mut i8,
                                                str_length as usize,
                                                str_length as usize))
                        }
                }
        }
        // unsafe fn for reading into stack array
        // caller must ensure stack array is large enough for n bytes.
        pub unsafe fn read_preallocated(self, n: usize, dest: *mut c_types::c_char,
                null_term : bool) -> Result<(), isize> {
                unsafe {
                        let res = os::strncpy_from_user(dest, self.0, n as isize);
                        if res < 0 {
                                return Err(res);
                        }

                        if null_term {
                                *dest.offset((n as isize)-1) = 0; // null terminate the last byte
                        }
                        return Ok(());
                }
        }
}
```

We don't provided write functions as it normally does not make sense to write to a userspace string.

The return type of the read function is the most problematic. Despite being a UserRPtrStr, the read function returns (on success) a Vec<i8>. The right type would really be std::ffi:CString, or at least std::ffi:OsString, but we do not have access to std and thus cannot use these. Unfortunately, String is not the right type. Rust Strings are encoded with UTF8, and thus all of the available library functions assume that the Strings are valid UTF8–which means that passing invalid UTF8 may result in undefined behavior (AKA bugs and security vulnerabilities), and of course we cannot rely on userspace to pass valid UTF8. So, our only option is to return a sequence of bytes, which leaves some of the manual legwork to the caller of the read function, which is undesired but currently necessary. We need this sequence to be a dynamically allocated container, so we can give it ownership of the strdup'd (kmalloc'd) string returned to us by the C function. As mentioned above, because Rust and the C code use the same allocator, we can give ownership of this dynamically allocated string to a Vec, and have Rust's scoping rules generate the "free" in the correct place!

Technically, the standard does not quite mark this as safe, but stipulates (in from_raw_parts documentation): "ptr needs to have been previously allocated via String/Vec<T>(at least, it's highly likely to be incorrect if it wasn't)." But since String and Vec use the same allocator as the kernel, this shouldn't be a problem, and using from_raw_parts with anything allocated by the global allocator will likely be stabilized sometime in the future.

We also provide a read_preallocated function for the UserRPtrStr, which reads into a pre-existing buffer. Currently the function is unsafe because the buffer is taken by raw pointer and must be large enough to hold the number of bytes to be copied. In the future, I hope to use the type system to codify this constraint, and mark the function as safe. In addition, this does not necessarily null terminate, like the read function, so some of the complexity of dealing with C strings is still present in the Rust implementation.

There is also a user pointer type for variable length blobs of binary data–essentially like pointers to string, except without the null-terminator problem.

An implementation of a type that performs "access_ok on a range" and then is capable of copying values unsafely (using unsafe_put_user, etc.) into that range (as a performance optimization) could also likely be included, and would avoid CVEs like the recent lack of an access_ok check due to performance optimizations. This wasn't necessary just yet, however, so I have no such implementation.

4.2 Syscall Macro

The user pointer types are useful, but only if they are used correctly. When writing a function that takes syscall arguments passed from C, the writer of the syscall function must create the appropriate user pointer structs, only create them once, and neglect them using the raw pointers anywhere else. This is arduous and can be somewhat error-prone. Thus, we implement a syscall macro, much like Linux's SYSCALL_DEFINE macro, that will automatically create two functions: one that takes only user pointer wrapper structs and provides the actual implementation of the syscall, and another that takes the raw C pointers, creates the user pointer structs, and calls the former function.

The desired macro is:

```
syscall_define(rust_add_key, add_key(type_k : *mut c_types::c_char : UserRStrPtr,
    len : c_types::c_int) -> c_types::c_long {
 // impl here
})
```

Where the first token is the name of the function callable from C, while the second token is the name of the pure-Rust function that takes only user pointer wrappers. Also note the extra type annotation on the mutable pointer that determines the wrapper struct to be created. Thus the desired result is:

```
#[no_mangle]
pub extern fn rust_add_key(type_k : *mut c_types::c_char,
    len : c_types::c_int) -> c_types::c_long {
  add_key(UserRStrPtr(type_k, len))
}
#[inline]
fn add_key(type_k : UserRStrPtr, len : c_types::c_int) {
         // impl here
}
```

The actual macro is surprisingly difficult to write, given the difficulty of implementing thing like optional parameters, and due to the hygienic macro system, which differs from C's text substitution system. For example, capturing something as a type means you cannot use it as an expression or identifier, and you cannot append any strings to identifiers. Thus, some constructs seem unnecessarily difficult to implement with macros. This is the implementation of the macro listed above:

```
macro_rules! syscall_define {
    ($c_fn_name:ident, $fn_name:ident($( $i:ident : $t:ty $(: $ptr_t:tt)* ),*) ->
    $ret:ty $impl:block) => {
            #[no_mangle]
            pub extern fn $c_fn_name($( $i : $t ),*) -> $ret  {
                    $fn_name($( gen_arg!($i , $($ptr_t),* $($ptr_t),*) ),*)
            }
            #[inline]
```

```
            fn $fn_name($( $i : gen_param_type!($t, $($ptr_t),*) ),*) -> $ret $impl
    }
}

macro_rules! gen_arg {
    ($i:ident, $ptr_t:tt) => ({let x : $ptr_t = $ptr_t ($i, ); x});
    ($i:ident, ) => ($i);
}

macro_rules! gen_param_type {
    ($t:ty, $ptr_t:ty) => ($ptr_t);
    ($t:ty, ) => ($t);
}
```

Note that each syscall argument takes an optional wrapper type. However, it is impossible to exactly express this, so we must use the Kleene star and pass the result to a second macro that can recognize either 0 or 1 arguments (gen_arg and gen_param_type) with appropriate matching clauses.

However, this macro does not work, as $ptr_t (which represents the type of the user pointer wrapper we want to use), when taken as a type "ty", the code cannot use it to initialize a struct: $ptr_t ($i,), the compiler complains that it expected an expression rather than a user pointer wrapper. However, taking it as an "ident" or a "tt" does not match against the user pointer wrapper type, which I believe is a Rust macro bug as "tt" is supposed to match against everything (and the Rust documentation/tutorials for the macro subsystem are woefully inadequate), but cannot match against our type. Thus, our final macro is a little more tedious, as it has to include the construction of our wrapper itself:

```
syscall_define(rust_add_key, add_key(
    type_k : *mut c_types::c_char : UserRStrPtr : UserRPtrStr::new(type_k),
    len : c_types::c_int) -> c_types::c_long {
 // impl here
})
```

And our final macro recognizes this optional expression and inserts it as initialization code:

```
macro_rules! syscall_define {
    ($c_fn_name:ident, $fn_name:ident($( $i:ident : $t:ty $(: $ptr_t:ty : $ptr_e:expr)* ),*) -> $ret
            #[no_mangle]
            pub extern fn $c_fn_name($( $i : $t ),*) -> $ret  {
                    $fn_name($( gen_arg!($i, $($ptr_t : $ptr_e),*) ),*)
            }
            #[inline]
            fn $fn_name($( $i : gen_param_type!($t, $($ptr_t),*) ),*) -> $ret $impl
    }
}

macro_rules! gen_arg {
    ($i:ident, $ptr_t:ty : $ptr_e:expr) => ($ptr_e);
    ($i:ident, ) => ($i);
}
```

```
macro_rules! gen_param_type {
    ($t:ty, $ptr_t:ty) => ($ptr_t);
    ($t:ty, ) => ($t);
}
```

5. KEY MANAGEMENT SUBSYSTEM REWRITE

Unfortunately, I was not able to fully implement a working key management subsystem due to time constraints, build and linker problems, bindgen errors, and strange Rust macro errors.

However, the key management system is a fantastic place to start rewriting Linux:

(1) The key management is heavily used by the kernel
(2) The key management system deals with user data from both external sources (network packets) and local sources (system calls), and thus is a prime target for code execution attacks. Not to mention that any bug or race condition in the implementation can (if it doesn't lead to remote code execution) lead to the exposure of important encryption keys normally reserved for privileged processes and the kernel. Thus Rust is an excellent choice
(3) The key management subsystem, despite being very well-written C code, has had its share of security vulnerabilities–they are seemingly unavoidable in C.
(4) the key management subsystem is being increasingly contributed to by cryptographers, who do not write secure code, ever.
(5) Most importantly, the key management subsystem is very self-contained. The implementation can be replaced without hundreds of functions invocations having to change, and without anyone relying on the particular implementation of the keys subsystem. Many modules use struct key *, but only ever use them as opaque pointers to pass to the key functions. If these were to change to void pointers and point to Rust structures instead, there would be little difference. And, much of the key subsystem's functionality can be disabled by configs, making initial implementation much easier.
(6) The key management subsystem does not need to be extremely performant, making it a good candidate for coarser synchronization and slower Rust code.

Its key states, which aren't normally accessed outside of the key management subsystem, and can easily be represented with an enum:

```
struct KeyLengths {
        quotalen : u16,
        datalen : u16,
}
struct KeyPerms {
        uid : os::kuid_t,
        gid : os::kgid_t,
        perm : u32,
}
enum Key {
        Uninstantiated(Type, Description),
        Instantiated(Type, Description, Payload, KeyLengths, KeyPerms, u64), // last is expiry time
        Negative(Type, Description),
        Expired(Type, Description, u64), // expired time
        Revoked(Type, Description, u64), // revoked time
        Dead(Type, Description)
```

```
}

enum Payload {
        Keyring(alloc::BTreeMap<Description, Key>),
        Data(Vec<u8>),
        RejectError(isize),
}
```

Thus Rust's pattern matching can be used to autogenerate the "state machine" code used to deal with keys, saving us from any errors in implementing a manual state machine/parser.

In addition, keyrings are very easily represented with the collection BTreeMap, which can be used for both quick iteration and quick indexing.

In order to support multithreading, I would have to create a Mutex, as a wrapper (similar to the allocator wrapper for kmalloc and kfree) around kernel semaphores, which would not be very difficult. The resulting wrapper would operate exactly like std::sync::Mutex.

In order to store keyrings, etc., they would likely be wrapped in alloc::Arc<Mutex<Payload»and then passed off as opaque pointers to live in kernel structs. For example, the per-process keyring lives in the process credentials.

However, for each of these places, when they are deallocated, I would have to implement a Rust function that takes the opaque pointer, interprets it as an Arc, and Drop it accordingly. Any C users of struct key might have to also inform the Rust code that they are dropping a pointer to the key.

In any case, the resulting code would mostly be constrained to the Rust implementation, and would likely be much safer than the equivalent C code.

6. WHY NOT REWRITE TCP STACK

I initially began by aiming to rewrite the Linux TCP stack. This was wildly impossible and generally useless:

(1) The TCP stack is integrated with everything: the ipv4 code, the ipv6 code, the lower-level networking code, etc.

(2) Since it has been around since near the beginning, it is massive and sprawling. Code is not rewritten in as much of a modular manner as is desired today, and so many people rely on specific structs and parts of the TCP implementation.

(3) Memory management for packets is managed by the sk_buff's which are passed through the entire networking stack.

(4) Network packets are not good candidates for Rust types, and they have variable length parameters like options, and variable length payloads. Thus bounds checking would be mostly avoided anyway.

There were essentially no self-contained portions of the TCP stack, and thus any rewrite would essentially be entirely unsafe code, and would provide no extra security guarantees. Not to mention the code would mostly be calling out to C functions, especially as each TCP function calls many functions from other subsystems that I wouldn't be rewriting. Not to mention that many of these functions are actually inline functions or macros that I would have to reimpelement in Rust with no extra safety guarantees. The code would end up being a direct line-by-line translation, wrapped in unsafe blocks. There are no core abstractions or memory management tools, except the sk_buff, which could be rewritten.

In addition, the networking stack very specifically needs high performance, so a reimplementation in Rust, and the overhead of calling between one another, would simply be too high to be practical.

7. CONCLUSION

The prospect of rewriting parts of the kernel in Rust is very attractive from a security and stability standpoint. There are a few too many downsides currently.

The first is the lack of good support for compilation within the kernel. Of course, it was possible to get it working, but it required quite a few iterations, a custom target, and about as unstable language features as I could find, along with the nightly compiler. It required a wrapper around Cargo, Xargo, a couple extra crates to satisfy the linker, and a lot of other custom settings. These features are simply too unstable, and have changed a significant amount in even the past year. They clearly have bugs (rust-bindgen in particular has many bugs, and I had to play around with it to get it working at all). Macros are not documented nearly in-depth enough in the Rust Book, and seem to have implementation and usability bugs as well, and macros are certainly necessary for building a large system like the kernel. Support for custom allocators is not quite there and involves too many unstable APIs (and Vec::from_raw_parts doesn't necessarily take things allocated with the custom allocator!), and the std/core split needs more work, as there are many types (especially CString) that were not available. Many crates are not available because they use the standard library, even though they really don't need it. Really, the first thing any programmers should do for Linux is reimplement the Rust standard library using kernel function calls.

In addition, interoperability is difficult as macros and inline functions cannot be called from Rust, causing us to reimplement C functionality in Rust, which can be a source of bugs.

Finally, there is the problem of having duplicated libraries for the C code and Rust code. We could of course implement wrappers for the Rust code over the C code (which leaks some of C's unsafety over to Rust), or vice-versa, but this introduces performance overhead due to extra function calls, and development overhead from having to maintain the wrappers. Thus we end up with two redundant associative array implementations in the same kernel. This is a problem for kernel size, maintainability, memory consumption, and i-cache consumption.

After this project, it seems a good way forward would be to implement the operating system in Rust, while reusing Linux's driver implementations, if at all possible. Some constructs, such as the C allocator, might as well remain implemented in C for performance reasons.