seL4 Overview and Tutorial

Nicholas Evancich

nick@trustedst.com

Trusted Science and Technology, Inc.

Agenda

OVERVIEW OF seL4

- History
- Capabilities
- Threads
- Booting

TUTORIAL

- Code walk through
- seL4Test
- Hello SecDev
- Threads
- Capabilities

What is seL4

seL4 = Secure Embedded L4 Microkernel

- seL4 is the world's first microkernel that brings with it a formal proof of correctness

- The kernel was designed against a specification and the translation of the source code to binary representation hold correctness proofs with them as well

- What does seL4 prove?
  - The binary code of the ARM version of the seL4 microkernel correctly implements the behavior described in its abstract specification and nothing more. Furthermore, the specification and the seL4 binary satisfy the classic security properties called *integrity* and *confidentiality*. (https://sel4.systems/Info/FAQ/proof.pml)
  - Integrity and Confidentiality derive from the CIA definition
    - Confidentiality refers to a system's efforts to keep their data private
    - Integrity ensures that data has not been tampered with and, therefore, can be trusted

What is a microkernel

Have been around since the early 70’s
- Being formally recognized/coined in 1981 (Accent from Carnegie-Mellon)

Near minimum amount of software that can provide mechanisms needed for an operating system.

Some mechanisms (Bare-minimum)
- Low level address space management
- Thread Management
- Inter Process Communication
- Interrupt management/routing

Generally the kernel is the most privileged, everything else contained in user space

Microkernel (Pros & Cons)

ADVANTAGES

Modularity

- Servers can be plugged, swapped, and restarted easily

Security

- Separation of address spaces managed by the kernel

Robustness

- The limited requirements of a microkernel generally lead to a simpler code base
- This is easier to test/peer review
- Driver cannot directly crash the kernel
- A system can be resurrected by monitor servers

DISADVANTAGES

Performance

- More context switches
- Because of crossing from User space to kernel space
  - A user space driver now communicates via IPC messages to another user space process
- In a monolithic kernel a driver would be in kernel space and already be at a high privilege

Complex process management

Generally not as many application/services for developers to build on

- Usually need to implement your own key components

Microkernel (cont)

- **Common Characteristics**
  - User space drivers and applications
  - IPC communication
  - Small kernel because of small set of responsibilities
- **Other Microkernels**
  - L4 Family (seL4, OKL4, Fiasco.OC)
  - GNU Hurd
  - MINIX (Andrew Tanenbaum’s academic microkernel)
  - Magenta (Google’s microkernel for Fuchsia OS)
  - Redox (A Rust microkernel)
  - QNX (Blackberry’s microkernel)

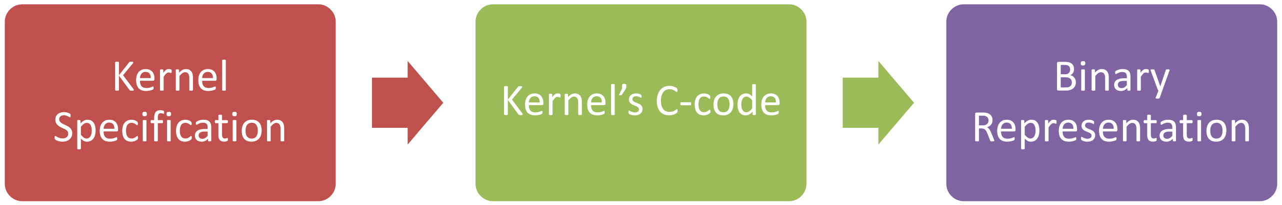
seL4 History

- In the L4 Family of Microkernels
  - Stems from Jochen Liedtke 1993 i386 assembly version
  - Has been generalized because the different L4’s don’t necessarily use the original ABI
  - L4 Kernels have a branching history of implementations
    - The UNSW (Univ New South Wales) group at NICTA’s L4-embedded version was the first to focus on embedded systems favoring
      - smaller memory footprints
      - lower complexity
    - OKLabs spun out from UNSW/NICTA with a commercial offering of L4, OKL4 (2006-2008)
      - This had capability based security

seL4 History (cont)

- Share basic mechanism concepts
  - Address Spaces
  - Threads
  - Scheduling
  - Thin IPC implementation between isolated servers
- seL4 was originally created at OKLabs from scratch and implemented in a Haskell specification
  - Resource access is determined by a capability model which enables formal reasoning about an object's permissions
  - Formal proof of functional correctness was completed in 2009
    - Proves implementation is correct against the Haskell specification
    - Proves the kernel is free of deadlocks, livelocks, buffer overflows, arithmetic exceptions or use of uninitialized variables
  - Formal proof of C to binary translation

What does seL4 prove?



What properties does this imply?

- This implies that the seL4 kernel lacks the following:
    - Buffer overflows
    - Null point dereferences
    - Memory leaks
    - Arithmetic overflows and exceptions
    - Undefined behavior
    - General pointer errors
- Assuming the assumptions are correct

Source: https://sel4.systems/Info/FAQ/proof.pml

Assumptions

- Assembly
  - Small amount of asm written and is assumes to be good
- Hardware
  - HW is working within its spec
- HW Management
  - Cache, TLB, etc are implemented correctly
- Boot code
  - Boot code puts the kernel into a good state
- DMA
  - Only the CPU & MMU can directly access memory
- Info side-channels
  - Does not cover timing channels

Capability Security Model

- Capability Security Model (CSM) is a design pattern in trustworthy computing
- A capability is a key
- It provides authority that references an object and provides certain access rights
- seL4 provide an implementation of this model with unique features

seL4 & CSM

- Capability spaces are implemented as a directed graph of kernel-managed capability nodes (CNodes)
- The reason why capabilities form the basis of security in seL4 is the fact that the kernel keeps track of everything in the capability derivation tree and a capability is required for any operation on a kernel object
- This prevents bad actor threads from gaining access to a resource in a trusted thread that they aren’t supposed to have access to
- Capabilities are unforgeable, transferable, and extensible
- If a thread is monitored and malicious behavior is detected, the capabilities that thread has can be revoked by a thread that owns a capability further up the capability derivation tree

Cspaces & CNodes

- In seL4, there are a few related structures to be familiar with when talking about capabilities:
  - Cspace
    - Directed graph of Cnodes
    - All capabilities reachable from the root of the Cspace
  - Capability Pointer (CPTR)
    - The address or pointer to a capability
  - CSpace Path (seL4_libs/libsel4vka/include/vka/cspacepath_t.h)
    - A structure for describing the capability in the graph
  - Croot
    - The beginning of a Cspace
  - CSpace Node (CNode)
    - Table of slots
    - Each slot may contain a capability

VSpaces

- Virtual address spaces are chunks of kernel managed memory that threads can use to run in, while being isolated from the rest of the system
- A thread must poses a capability to a chunk of untyped memory, which it can retype as virtual memory
- This virtual memory can be associated with a thread object and when the thread is started, it will execute inside of that memory

VSpaces (cont)

- The memory objects are hardware dependent. For ARM, some important objects are:

- Page Directory (PD)
- Page Directory Entry (PDE)
  - Entry in the page directory
  - Contains a pointer to a page table, page, or it can be invalid
- Page Global Directory (PGD)
- Page Upper Directory (PUD)
- Page Table (PT)
  - Contains page table entries
- Page Table Entry (PTE)
  - An entry in a page table
  - Contains a pointer to a page or it can be invalid
- Page
  - Actual physical memory, usually a 4k region
  - This can be mapped into a VSpace.

Untyped Memory

- Untyped memory is the default classification of memory in seL4
- Untyped memory is required to re-type into another kernel object
- This means that the thread that invokes the untyped memory needs to have access to that to change it to something else
- When untyped memory is retyped, the kernel knows about it and can keep track of how it is used

Kernel Overview

- The seL4 kernel isolates applications and provides primitive mechanisms for these isolated instances to communicate with one another
- It is up to the userspace libraries to abstract these primitives and provide easier to use API’s
- The kernel provides objects for threads, address spaces, inter-process communication, notifications, device primitives, and capability spaces
- These objects can be used in user space to interact with the kernel and other objects
- These services are invoked with system calls, which consist of the kernel object to be operated on and the capability to that object, some of which can take more arguments

Root Thread

- The most important chunk of data to worry about once the root thread's main function starts to execute is the bootinfo structure. This is what other objects are built from.
- Some important information that the bootinfo struct contains is the root thread's TCB object, the initial CSpace, VSpace, and IPC buffer.
- The bootinfo struct can be found in kernel/libsel4/include/sel4/bootinfo_types.h
- To acquire the original boot information, the platsupport_get_bootinfo function will initialize a user space boot info variable
- From here, memory can be allocated, capabilities minted and passed, and a whole slew of other exciting seL4 activities.

Thread Control Blocks

- seL4 uses threads to group executing contexts together
- Threads are contained by Thread Control Block tcb_tstructures
- As of version 8.0 of the seL4 code base, these can be seen in kernel/include/object/structures.h
- Each TCB has an associated CSpace and Vspace
- Like other objects, TCBs are created with the seL4_Untyped_Retype() method
- A newly created thread is initially inactive
- It is configured by setting
- its CSpace and VSpace with the seL4_TCB_SetSpace() or seL4_TCB_Configure() methods and then calling seL4_TCB_WriteRegisters() with an initial stack pointer
- and instruction pointer
- The thread can then be activated either by setting the resume_target parameter in the seL4_TCB_WriteRegisters()

Endpoints

- Allow small amounts of data and capabilities to be transferred between two threads
- Invoked with seL4 system calls:
  - seL4_Send
  - seL4_Call
  - seL4_Recv
  - seL4_ReplyRecv

IPC

- IPC is the mechanism for thread-to-thread and thread-to-kernel communication
- IPC messages can be sent to either an “Endpoint” or other kernel objects
- The seL4 IPC model consists of the thread having the capability to access the kernel object of interest and then loading up the thread’s message register
- IPC message registers have 3 descriptive registers that are backed by CPU registers and anywhere from 1-4 “Message Registers” that are backed by the thread’s IPC buffer (& thread_t.tcbIPCBuffer)

IPC (cont)

- Capability Register
- Badge Register
- Message tag (seL4_MessageInfo_t)
  - Has four fields
    - Label
      - Kernel does not do anything with this
      - First data payload of the message
    - Message Length
    - Number of capabilities
    - Capabilities Unwrapped
      - Used only on the receive side
      - Indicates the manner in which capabilities were received
- Message Registers

Notification

- Notifications are the non-blocking signaling mechanism that applications running on seL4 can use as binary semaphores
- Related sel4 system calls:
  - seL4_Signal
    - Updates the notification object's word
    - Unblocks the first thread waiting on the notification, while all other threads that are waiting on the notification object keep waiting until the next time the notification is signaled.
    - If invoked with an un-badged capability the first queued thread is unblocked
  - seL4_Wait
    - If notification object's word is zero the invoker blocks
    - If notification object's word is not zero then the call returns and sets the notification object's word to zero
  - seL4_Poll
    - If notification object's word is zero the call will return immediately without blocking
    - If notification object's word is not zero then the call returns and sets the notification object's word to zero
  - seL4_TCB_BindNotification
    - Binds notification objects and TCBs together

Kernel Boot Sequence

- After the board is set up with the bootloader and execution is handed off to the elfloader, where it then branches to the kernel
- The kernel boot sequence can be traced in src/arch/arm/kernel/boot.c
- The first function to be called is init_kernel
- Then try_init_kernel is called and depending on the number of CPUs, this function branches to try_init_kernel_secondary_core

Kernel Boot Sequence (cont)

- In try_init_kernel several important things happen:
- Virtual memory for the kernel is set up in map_kernel_window.
  - This also calls map_kernel_devices
    - Where the platform dependent kernel_devices gets mapped
    - This is 32/64 bit dependent and in the src/arch/arm/$ARCH/machine/hardware.c source file.
- CPU is initialized in init_cpu
  - The activate_global_pd is a macro that is dependent on whether it is running in 32 or 64 bit mode
    - In 64 bit mode this actually calls activate_kernel_vspace
      - This function is in src/arch/arm/64/kernel/vspace.c
  - The kernel stack is set up
  - The timer is initialized

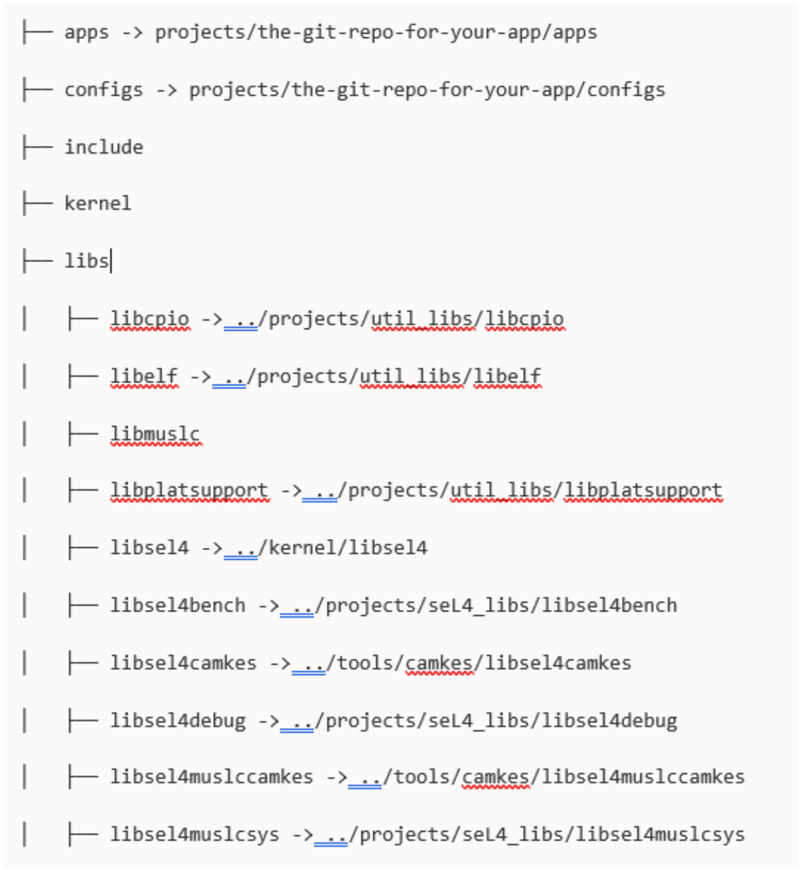
Kernel Boot Sequence (cont)

- Platform is initialized in init_plat
  - The IRQ Controller is initialized with initIRQController
    - The implementation of this is found in src/plat/$PLAT/machine/ or src/arch/arm/machine/gic_$GIC_VERSION.c
  - The L2 Cache is initialized
  - In the future other platform dependent initializations may take place here
- Capabilities are created for the root environment
- The boot info frame is allocated and initialized
- The initial address space is created
- The boot info frame capability is created
- The initial threads IPC buffer is created
- This is all brought together and the initial thread control block of type tcb_t is created with a call to create_initial_thread
- L1 cache is invalidated
- Execution is returned to the init_kernel function

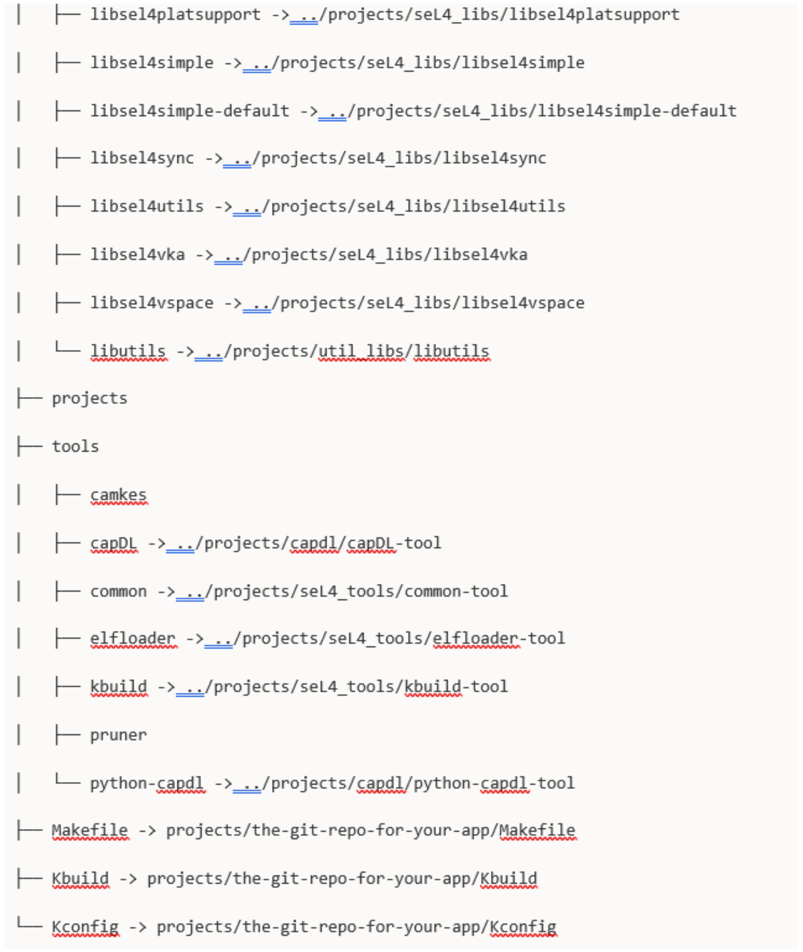
Navigating the Code Base

- The directory structure of the seL4 kernel, libraries, and tools are organized based on a `repo` manifest file, which pulls the correct `git` repositories and places them in a certain place
- The `repo` tool places everything such that the root directory is not tracked by `git` but the sub directories are still tracked by their respective repositories
- This makes it relatively easy for build tool files to reference other projects and not muddy up the source trees with generated and compiled code

Navigating the Code Base (cont)



```
├── apps -> projects/the-git-repo-for-your-app/apps
├── configs -> projects/the-git-repo-for-your-app/configs
├── include
├── kernel
├── libs
│   ├── libcpio ->../projects/util_libs/libcpio
│   ├── libelf ->../projects/util_libs/libelf
│   ├── libmuslc
│   ├── libplatsupport ->../projects/util_libs/libplatsupport
│   ├── libsel4 ->../kernel/libsel4
│   ├── libsel4bench ->../projects/seL4_libs/libsel4bench
│   ├── libsel4camkes ->../tools/camkes/libsel4camkes
│   ├── libsel4debug ->../projects/seL4_libs/libsel4debug
│   ├── libsel4muslccamkes ->../tools/camkes/libsel4muslccamkes
│   ├── libsel4muslcsys ->../projects/seL4_libs/libsel4muslcsys
```



```
│   ├── libsel4platsupport ->../projects/seL4_libs/libsel4platsupport
│   ├── libsel4simple ->../projects/seL4_libs/libsel4simple
│   ├── libsel4simple-default ->../projects/seL4_libs/libsel4simple-default
│   ├── libsel4sync ->../projects/seL4_libs/libsel4sync
│   ├── libsel4utils ->../projects/seL4_libs/libsel4utils
│   ├── libsel4vka ->../projects/seL4_libs/libsel4vka
│   ├── libsel4vspace ->../projects/seL4_libs/libsel4vspace
│   └── libutils ->../projects/util_libs/libutils
├── projects
├── tools
│   ├── camkes
│   ├── capDL ->../projects/capdl/capDL-tool
│   ├── common ->../projects/seL4_tools/common-tool
│   ├── elfloader ->../projects/seL4_tools/elfloader-tool
│   ├── kbuild ->../projects/seL4_tools/kbuild-tool
│   ├── pruner
│   └── python-capdl ->../projects/capdl/python-capdl-tool
├── Makefile -> projects/the-git-repo-for-your-app/Makefile
├── Kbuild -> projects/the-git-repo-for-your-app/Kbuild
└── Kconfig -> projects/the-git-repo-for-your-app/Kconfig
```

Navigating the Code Base (cont)

- At first glance the directory structure can be somewhat intimidating
- If you are ever confused where certain components might be, the main directories where source code is located are `kernel` and `projects`
- The `kernel` directory is, obviously enough, where all the kernel code is. However, as can be seen in the previous listing, the `apps` directory is really a link to a directory under `projects` and so are some library directories under `libs` and some tools directories under `tools`
- You can see where all of the directories are placed and linked to in the project manifest file
- You can run the command `repo manifest` at the command line and it will print the contents of the currently used manifest file in `.repo/manifest.xml`
- Each entry has a `path`, which is where the directory for that git repository is placed