Retrofitting Zephyr Memory Protection

Andrew Boie, Software Engineer, Intel Corporation
What is Zephyr?

• Apache 2.0 licensed RTOS
• https://www.zephyrproject.org/
• Targeting a wide range of devices
  • Lowest end: Quark D2000, Nucleo boards with 8K of RAM
  • Highest end: MCUs with RAM on order of megabytes but can't run Linux
    • Insufficient RAM
    • Real-time constraints
    • Safety Certification
Memory Protection Hardware

MPU - Memory Protection Unit
- Fixed number of configurable **regions** each with their own access policy
- No virtualization, physical memory addresses
- Typically have constraints on region specification
  - Common: region sizes must be power of two, aligned to their size

MMU - Memory Management Unit
- Address space divided into equal sized **pages** (typically 4K)
- Configuration for caching and access policy for each page set in **page tables**
- Optional support for virtual memory
- MPU-like behavior with identity page table
Memory Protection in Zephyr

- Zephyr had no means of preventing unwanted memory access
- Initial efforts targeting MPU-based systems
- Joint effort with most contributions from Linaro, Synopsys, and Intel
- Milestones:
  - 1.9 release (7/2017): MMU/MPU enabled, stack overflow protection on ARM/x86
  - 1.10 release (11/2017): User mode support on x86 MMU
  - 1.11 release (3/2018): User mode support on ARC/ARM MPU
- Further work
  - Leverage the capabilities of MMU-based systems
  - Flesh out APIs and iterative refinement based on user feedback
  - Support for additional CPU architectures
Use-Cases for Memory Protection

• Protect against unintentional programming errors
  • Stack overflows, writing to bad memory, data corruption
• Sandbox complex data parsers and interpreters
  • Network protocols
  • Filesystems
  • Reduce likelihood of third-party data compromising the system
• Support the notion of multiple logical isolated applications
Comparison With Other RTOSes

- **FreeRTOS-MPU**
  - Supports ARM MPU hardware
  - Unprivileged "User" threads with configurable memory access, system calls for privileged operations
  - Fork of main FreeRTOS codebase, not well maintained, often doesn't compile

- **NuttX Protected Build**
  - Supports ARM MPU and MMU (with identity page table)
  - Unprivileged threads similar to FreeRTOS-MPU
  - Separately loaded applications
  - Many features proposed but still WIP

- **ThreadX Modules**
  - MPU or MMU Virtualized address spaces for separately loaded modules with thread-level memory protection features
  - Support for lots of different CPUs
  - Not free. Royalty-free license with significant upfront cost, modules feature costs extra
Layered Approach to Implementation

• Layer 1: Boot-time MMU/MPU configuration
  • Enforce no-execute for non-text, comprehensible exceptions for accessing/executing nonsense addresses, NULL pointer dereferences, etc.

• Layer 2: Supervisor-mode stack overflow detection
  • Stack overflow errors can be truly baffling if the cause is unknown

• Layer 3: User-mode threads
  • User threads with CPU running in un-privileged mode, system calls, thread-level kernel object/driver permissions and memory policy

• Layer 4: Virtual Memory
  • Zephyr "processes" in their own VM
  • Implementation in progress
Threat Model

- User mode threads are considered untrusted and are isolated from the kernel and each other.
- A flawed or malicious user thread cannot:
  - Leak or modify private data of another thread unless specifically granted permission.
  - Interfere with or control another thread except through designed thread communication APIs (pipes, semaphores, etc.)
Assumptions

- Kernel and any threads running as supervisor are trusted
- Toolchain and all headers/source code part of kernel build assumed trusted
- No prevention of Denial of Service (DoS) through thread CPU starvation
  - Zephyr does not have priority aging
  - Arranging thread priority levels is important!
High-Level Policy Decisions

• User mode threads are by default granted only:
  • read/write access to their own stack memory
  • read-only/execute access to program text and ROM
  • Memory Domain APIs to configure access to additional regions with child thread inheritance
• Threads cannot use device drivers or kernel objects without being granted permission
  • Permission granted by other threads with sufficient permission or inherited
• System call API parameters are rigorously checked
• User mode stack overflows are safely caught
Constraints

• All kernel objects must be defined statically at build time if they are to be used from user mode
• Application code loaded separately from the kernel cannot directly instantiate their own kernel objects
• Forthcoming APIs to allocate kernel objects from pools or a runtime-managed heap in progress
Permission Model

- Each kernel object has a bitfield indicating what threads have access to it
- Supervisor threads can grant object access to any thread
- User threads may grant object object access to another thread if the calling thread has permissions on both the object and the target thread
- Newly created threads may optionally inherit object permissions of the parent thread.
Summary: Overview

- Threads may run in either supervisor (privileged) or user (non-privileged) mode
- Kernel object permissions are managed on per-object, per-thread basis
- Memory domain APIs exist to allow threads to access globals and share memory regions with each other
- Privilege elevation handled via system calls
- End goal is to aid debugging and improve the security of the system
Implementation Details
Boot Time Memory Configuration (MPU)

• Simplest policy: configure the MPU at boot and leave it alone
• Dedicated MPU regions reserved for address ranges whose policy will not change while the system is up
  • Program text
  • ROM
  • RAM
• No complex associated data structures, just program the registers at boot
Boot Time Memory Configuration (MMU)

- MMU_BOOT_REGION() macros to set address range permissions
  - Some set in core x86 code based on linker-defined regions for text/ROM/RAM regions defined by linker variables and CPU MMIOs
  - Additional definitions in SOC/board code for peripheral MMIO ranges
  - All other memory ranges will be non-present pages
- Populates a special MMU_LIST section containing region policy
- gen_x86_mmu.py
  - Post-build script consuming MMU_LIST to generate page directory/tables
  - MMU_LIST omitted from final binary
Stack Overflow Protection

- Typically requires extra space for guard areas
  - k_thread_stack_t typedef to distinguish between entire stack memory area, and char * for thread-usable area
  - K_THREAD_STACK_DEFINE(), K_THREAD_STACK_BUFFER() and related macros implemented on arch-specific basis
- Architecture-specific implementation
  - x86 MMU: Thread stacks are page-aligned have guard page reserved preceding stack area that is marked non-present in MMU at thread creation time
  - ARM MPU: Dedicated MPU region, programmed on context switch to 32-byte guard area preceding stack space
  - ARC: Optional ARCv2 CPU support for directly detecting stack overflows; no guard
- Stack overflows in kernel mode are fatal - can't guarantee against corruption

Example: x86

Guard Page

Stack Buffer

Initial stack pointer

Provided stack size

k_thread_stack_t pointer
User Mode

• Control access to kernel objects and device drivers
  • Per-object and per-thread basis
• Maintain compatibility with existing Zephyr APIs
• Implement system calls for privilege elevation
• Arch-specific code to enter user mode
• Validate system call parameters including kernel object pointers
• Do not require changes to individual drivers
• Manage user mode access to memory
Kernel Objects

• Three main types of kernel-private data structures
  • Kernel API data structures - k_thread, k_sem, k_mutex, k_pipe, etc.
  • All device driver instances
  • All thread stacks
    • Instead of individual structs, these are arrays of a special typedef to character data

• To preserve Zephyr API compatibility, all are referenced by memory address
  • Acts as a handle for user threads, object memory not accessible
  • Need a system for validating object pointers passed to system calls
Creating New Kernel Object Types

• Creating new kernel object types is easy!
  • Add the name of the associated data structure to the build
    • Struct name itself for new kernel APIs
    • API struct name for new device driver subsystem types
  • Small modifications to some lists in two C files
    • Could eventually be automated

• Recognizing instances of kernel objects and providing a validation function for them is all handled automatically at build time
Kernel Objects: Placement Constraints

- Must be declared as a top-level global
  - Needs to appear in the kernel's ELF symbol table
  - OK to declare with static scope
  - May be embedded as members of larger data structures
- Memory for an object must be exclusive to that object
  - Can't be part of a union data type
- Must be in the kernel data section
- Objects that do not meet these constraints will not be accessible from user mode
- Future work: support runtime allocation use-cases from slabs/kernel heap
Kernel Objects: struct _k_object

- Contains metadata for each kernel object in the system
  - Permission bitfield indicating thread permissions for that object
  - Object type information enum
    - K_OBJ_THREAD
    - K_OBJ_UART_DRIVER...
  - Flags - initialization state, public/private, others as needed
  - Extra data in some cases
    - Stack object size
    - Build-time assigned thread ID

```c
struct _k_object {
  char *name;
  u8_t perms[CONFIG_MAX_THREAD_BYTES];
  u8_t type;
  u8_t flags;
  u32_t data;
} __packed;

extern struct _k_object * _k_object_find(void *obj);
```
Kernel Objects: DWARF Scanning

• Problem: need to find all the kernel objects
  • Map object memory addresses to instantiations of struct _k_object containing metadata
  • Validate kernel object pointers passed in from user threads
• gen_kobject_list.py
  • Uses pyelftools to unpack ELF binary and fetch all the DWARP debug information
  • Object identification
    • Kernel API objects - identify by struct type
    • Driver instances - all are struct device, identify by API struct member
Kernel Objects: GPERF table generation

- gperf is a GNU tool for creating perfect hash tables
- gen_kobject_list.py emits gperf configuration file
- output of gperf is C file with hash function and generated data structures
- Further processing on generated C file for efficiency
Kernel Objects: GPERF C File Transformation

```
static struct _k_object wordlist[] =
{
    #line 32 "kobject_hash.gperf"
    "\374\323A\000",{},K_OBJ_DRIVER_UART,0,0),
    #line 14 "kobject_hash.gperf"
    "4\322A\000",{},K_OBJ_MUTEX,0,0),
    {}},
    #line 18 "kobject_hash.gperf"
    "\204\322A\000",{},K_OBJ_MUTEX,0,0),
    {}},
    #line 13 "kobject_hash.gperf"
    " \322A\000",{},K_OBJ_MUTEX,0,0),
    {}},
    #line 17 "kobject_hash.gperf"
    "p\322A\000",{},K_OBJ_MUTEX,0,0),
    {}},
    #line 36 "kobject_hash.gperf"
    " \000@\000",{},K_OBJ_THREAD,0,7),
};
```

```
static struct _k_object wordlist[] =
{
    #line 32 "kobject_hash.gperf"
    "(char *)0x0041d3fc,{},K_OBJ_DRIVER_UART,0,0),
    #line 14 "kobject_hash.gperf"
    "(char *)0x0041d234,{},K_OBJ_MUTEX,0,0),
    {}},
    #line 18 "kobject_hash.gperf"
    "(char *)0x0041d284,{},K_OBJ_MUTEX,0,0),
    {}},
    #line 13 "kobject_hash.gperf"
    "(char *)0x0041d220,{},K_OBJ_MUTEX,0,0),
    {}},
    #line 17 "kobject_hash.gperf"
    "(char *)0x0041d270,{},K_OBJ_MUTEX,0,0),
    {}},
    #line 36 "kobject_hash.gperf"
    "(char *)0x00400020,{},K_OBJ_THREAD,0,7),
};
```
Kernel Object Permissions

- Supervisor threads can access all objects
  - Permissions still tracked
    - Thread drops to user mode
    - Creates child user threads with object permission inheritance enabled
  - May designate some objects as "public" and usable by all threads

- User threads
  - If created with permission inheritance, gain access to all parent thread's permissions except parent thread object
  - `k_object_access_grant()` calls must have permission on both the target thread and the object being granted permission to
System Calls

• Typical OS mechanism for allowing un-privileged threads to perform operations they wouldn't otherwise be able to do
• On all arches, API ID and parameters are marshaled into registers and a software interrupt is triggered
  • Up to six registers used; additional args passed in via struct
• Common landing site for system calls on kernel side
  • Validate API ID, execute the handler function
  • Clean general purpose registers on exit to prevent private data leakage
• Use build-time logic to make adding new system calls as painless as possible
System Call Components

- Very easy for developers to define
- Created by developer for each system call:
  - System call header prototype
    ```c
    __syscall void k_sleep(s32_t duration);
    ```
  - Handler function for argument validation
  - Implementation function
- Auto-generated for each system call:
  - System call ID enumerated type
  - Handler function prototypes
  - `__k_syscall_table` entry mapping ID to handler function
  - `__weak` handler function for system calls excluded from kernel config
  - System call invocation function
System Calls: Handler Functions

- Base kernel APIs and driver subsystem implementations do little or no checking
  - For footprint reasons, core kernel assumes validity and may only have assertions
  - Driver subsystem APIs tend to return -errno on bad parameters but not in all situations
- Invocations of these through system calls go through dedicated handler functions first:
  - Verify caller permissions on provided memory buffers or data passed via pointer
  - Copy any parameter data passed in via pointer to local memory
  - Verify object pointers, permission, initialization state
  - Verify parameter values which are otherwise left to assertions or simply un-checked
- The combination of checks done in handlers and the API itself must prevent any compromise of the system
- Any failed handler function checks will induce a k_oops() and terminate the calling thread
  - Done this way to preserve existing API semantics which generally do not propagate return values
System Calls: Implementation Functions

- When running in supervisor mode, syscall trampoline and handler functions completely skipped
- These are the actual implementation of the API
  - Kernel object API code under kernel/
  - Driver subsystem API functions defined at the subsystem level
- Support for inline functions in headers and C functions
- All prefixed with _impl_<name of API>
- Before memory protection, these all existed and just needed handlers to more rigorously validate arguments
API Call

User Mode?

N

Y

Marshal args, Trigger SW IRQ

Valid Call ID?

Y

N

Lookup handler in dispatch table

Handler Checks

k_oops()

Return to Caller

Implementation Function

Marshal Return Value, exit IRQ

Implementation Function

k_oops()
System Calls: Build-Time Magic

- Limited parsing of kernel header files, looking for function prototypes prefixed with "__syscall"
- Parsing limited to determining return value and argument types to generate additional functions
  - Some minor limitations in parsing with array/function pointer argument types which can be easily worked around
- Generated headers contains implementation of API as an inline function - invokes syscall trap or direct call to implementation as appropriate
- Some generated C code for default handler and dispatch table entry
Memory Domains

- User threads by default can't look at any RAM except their own stacks
- Need a flexible way to designate additional memory areas that a thread has access to
- Limited number of total MPU regions needs to be taken into consideration
- Grant access to top-level data or BSS section globals defined and used by the thread, or application data that needs to be shared between threads
- Memory Domain APIs exist to handle re-programming the MPU for the incoming thread's memory access policy on context switch
- Facilities for grouping related data by the linker in contiguous and appropriately aligned blocks still WIP (see Future Work)
- Implemented for MMU as well (for now)
  - Future work to virtualize address space for MMU-based systems will obsolete these
Memory Domain: Implementation

- Memory Domain APIs are supervisor-access only, no syscalls
- Implemented as an object struct k_mem_domain
  - Contains some number of memory partitions (struct k_mem_partition)
    - Up to the maximum number of regions supported by MPU hardware, no limit for MMU
    - Each partition is a starting address, size, and access policy
    - Hardware dictates alignment and size constraints
  - APIs to add/remove partitions to an initialized memory domain object
- Any thread may be added/removed to a particular Memory Domain to implement an access policy for that thread
- MPU region registers or MMU page tables updated upon context switch to activate policy for incoming thread
Special Case: Application Memory

- CONFIG_APPLICATION_MEMORY in Kconfig
- A stopgap measure
  - Mostly to rapidly enable existing test cases to run in user mode
- All toplevel globals in non-kernel object files (libs, application code) placed in user read/writable section by linker and access policy configured in MMU/MPU at boot
- Additional build phase to determine size of sections and ensure correct power-of-two alignment for MPU-based systems that require it
- Need something better for real-world applications
  - Queued for future work
Summary: Implementation Details

- Adding new system calls or kernel object types is very easy for developers.
- Memory domains are used to partition RAM for groups of threads to share.
- Further work to be done in tying together memory domains and the consolidation of globals into different linker sections.
- Much more work we plan to do, as we shall see...
Future Work
Runtime Kernel Object Allocation

• Not always possible to define all kernel objects used at build time
  • Build-time constraints prevent allocation of kernel objects in separately loaded application code at all

• Two approaches, both under implementation
  • Build-time defined slab pools of kernel objects
    • Pools are build-defined arrays of various objects and validated as normal
  • Kernel-side heap allocation of kernel objects
    • Supplemental runtime hash table for tracking validity of new objects
    • User mode no direct access to this heap!
Kernel API Improvements

• Not all kernel APIs exposed as system calls
  • Many combine user and private kernel data in ways which could be attacked
  • k_mem_pool, k_poll, k_queue

• Need some better heap features
  • k_mem_pool APIs were designed to be ISR-safe and not usable from user mode
  • newlib heap is just a singleton for entire address space since no VM
  • Need a k_mem_pool equivalent that runs entirely in user mode, using memory domains to control access

• User-mode workqueues
  • k_work_q threads currently run as supervisor using k_queue for data buffering
Memory Organization Features

- "Application Memory" feature was useful for getting test cases up but does not work well for real-world uses
- We need a solution which handles both setting up 1..N memory areas for applications
  - Configure memory domains
  - Tie into linker scripts to ensure the data gets where it needs to be
  - Handle alignment constraints
- No design for this yet, under discussion
Virtual Memory

- Will supersede Memory Domains for MMU-based systems
- Zephyr to support one or more VM spaces for user-mode threads to execute in
- Allows for the introduction of Zephyr "processes" which are completely isolated from each other
- Will require design of some new APIs
- Allows notion of separately loaded application modules
Meltdown Mitigation on x86

- Current X86 port of memory protection is vulnerable
- Need to have kernel-only memory pages completely un-mapped when CPU is running in user mode (Ring 3)
- Best solution is to virtualize address space
  - 2G/2G user/kernel memory split
  - Only map those kernel pages needed for exceptions/IRQ handling
- Alternate page directory for running in user mode with non-essential kernel pages unmapped
- Not much we can do for Spectre on the OS side
  - Some mitigation possible in toolchain
  - Ultimately the real fix is in better hardware
Demand Paging

• On some systems the entire firmware image cannot be loaded entirely into available RAM
• Support the notion of swapping in/out ROM pages as necessary
  • Abstraction for transport method: alternate RAM, Flash, etc.
  • Abstraction for page eviction algorithm
• Using NuttX's implementation as a model
Security Features

• On X86, SMEP/SMAP
  • Supervisor Mode Execution/Access Prevention
  • Ring 0 no longer a superset of Ring 3
  • Prevent unintentional execution of code in user memory
  • Prevent unintentional access of user-provided data
• Address Space Layout Randomization (ASLR)
  • Much more difficult to pull off buffer overflow attacks
Call To Action

• Want to learn more? Have some ideas? Get started here:
  • https://www.zephyrproject.org/

• Check out our codebase on GitHub:
  • https://github.com/zephyrproject-rtos/zephyr

• Join our mailing list or hang out in our IRC channel!

• We also have a weekly call for those very interested in these memory protection features