Security in Zephyr and Fuchsia

Stephen Smalley and James Carter
Trust Mechanisms
Information Assurance (IA) Research
National Security Agency

August 27, 2018

About Us

- Perform R&D in support of NSA's Information Assurance (IA) mission to protect National Security Information and Information Systems.
- Research and develop hardware and software security architectures and mechanisms to facilitate trust.
- 25+ years of operating system security R&D
  - DTMach, DTOS, Flask, ...
- First at NSA to create and release open source software (SELinux, Dec 22 2000).
- Long history of open source contribution and collaboration.
  - Linux, Xen, FreeBSD, Darwin, Android

Zephyr and Fuchsia

- Two emerging open source operating systems
- Targeting very different use cases
- With very different OS architectures
  - Both from each other and from Linux
- We'll be examining:
  - their OS architectures and security mechanisms
  - prior and ongoing work to advance their security
  - how they compare with Linux-based systems

What is Zephyr?

- Cross-architecture, vendor-neutral RTOS for IoT devices
- Sponsored by Linux Foundation
- Targeting devices where Linux is not considered viable
  - 32-bit microcontrollers ranging from 8kB RAM to several MB.
  - Seeking to be a new “Linux” for little devices
- Security as a stated goal and focus
- https://www.zephyrproject.org

Zephyr: In the beginning

- Single executable, single address space OS
- Kernel as library linked into application
- All threads running in supervisor mode
- No memory protection, no virtual memory
- Typical for many RTOSes
- Focused on minimizing footprint, overhead
- Security efforts focused on development process, code auditing, static analysis, update, crypto, etc not OS protection mechanisms.

Zephyr: Motivation for OS protections

- Increase difficulty of exploitation of software flaws.
- Limit the damage from a single flaw.
- Sandbox untrusted components.
- Protect integrity of critical processing and data.
- Enforce desired information flows.
- Prevent leakage of sensitive data/keys.
- Improve robustness.

Zephyr: Credit Where Credit is Due

- Most of the Zephyr protection work has been done by the core Zephyr developers, particularly from Intel, Linaro and Synopsys.
- We'll call out some of our own specific contributions along the way.

Zephyr: Hardware Limitations

- Most microcontrollers lack a MMU.
  - No virtual memory support
- Some have a Memory Protection Unit (MPU).
  - Limited number of discretely protected physical regions.
    - Often as few as 8 distinct regions supported
  - MPUs are very limited in their flexibility (pre-ARMv8-M).
    - ARMv7-M: Power-of-2 size, aligned to size
    - NXP MPU only imposes modulo 32-byte restrictions

Zephyr: Protection Design Constraints

- Initial focus on supporting typical microcontrollers.
  - Can use a MMU if present, but must also work on MPU-only boards.
- Minimize changes to kernel APIs.
  - Can't rewrite to use handles/file descriptors.
- Minimize and bound memory and runtime overheads.
  - Do as much at build time as possible, preserve real-time guarantees.
- No impact on low end boards.
  - Fully configurable, no overheads if disabled.

Zephyr: Basic Memory Protections

- First appearing in v1.8, official in v1.9
- Depends on hardware MPU or MMU support
- Enforces RO/NX, stack depth overflow protections
- Most work done at build and boot time only (runtime support for stack depth overflow protections)
- Our contribution: protection tests
  - Modeled after subset of lkdtm tests in Linux from KSPP
  - Detected bugs and regressions in Zephyr MPU drivers
  - Now used as part of regression testing

Zephyr: Userspace Support

- Introduced for x86 in v1.10, for ARM and ARC in v1.11
- Builds on memory protection support, requires MPU/MMU
- Supports user mode threads with isolated memory
- Our contribution: userspace tests
  - Verifies (some) security-relevant properties for user mode threads
  - Confirmed correctness of x86 implementation (wrt to properties)
  - Used to validate initial ARM and ARC userspace implementations
  - Now used as part of regression testing

Zephyr: Userspace Memory Model

- Single executable and address space OS (still)
- User threads, not full processes
  - Explicitly launched by application code as user threads
  - RX/RO to text / read-only data, RW to per-thread stack
  - Memory domain abstraction for programmer-defined explicit shared memory regions among user threads.
  - Optional application memory feature to allow user threads to access application global variables.

Zephyr: Userspace Kernel Interface

- Kernel object references
  - Addresses as “handles” to avoid API rewrite
  - Kernel validates addresses via perfect hash for static objects, red-black tree for dynamic.
- Object permissions model
  - User threads must first be granted permissions to an object.
  - Optionally inherited from parent to child.
  - All-or-none, no per-operation or read/write distinctions.
- System calls
  - Transparent build-time and runtime redirection of API calls.
  - Only a select subset of kernel APIs exposed as system calls, vetted for trust.
  - Helpers for argument validation.

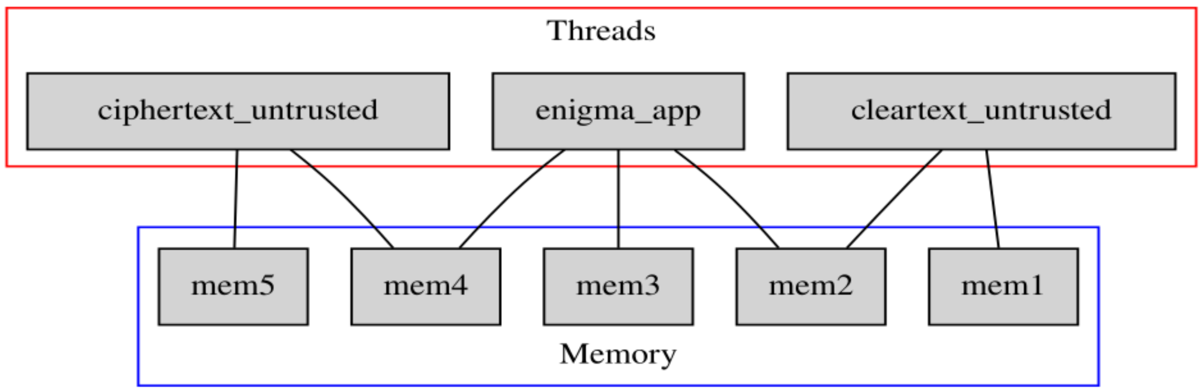
Zephyr: Application Memory

- Original application memory feature limited to all-or-nothing access.
  - All user threads can access all application global variables.
- High burden on application developers to leverage memory domain mechanism.
  - Manually organize application global variable memory layout to meet (MPU-specific) size/alignment restrictions.
  - Manually define and assign memory partitions and domains.

Zephyr: App Shared Memory

- New feature coming in v1.13, contributed by us.
- Provides a (more) developer-friendly way of grouping application globals based on desired protections.
- Automatically generates linker script, section markings, memory partition/domain structures.
- Provides helpers to ease application coding.
- No panacea, but a step forward.

Zephyr: App Shared Memory Example



Notes:

mem1 and mem5 are untrusted thread local memories.
mem2 and mem4 provide a common data buffer between threads.
mem3 provides a secure location for the enigma state information.

Zephyr: Areas for Future Work

- MPU virtualization
- Compartmentalization of program text and rodata
- Full support for multiple applications and program loading
- Kernel self-protection features ala KSPP
- Leveraging ARMv8-M features (more flexible MPU configuration, TrustZone-M support) to increase security
- Some form of MAC suited to RTOSes (e.g. build-time application partitioning/pipelining based on config).

Zephyr vs Linux OS security

- RO/NX memory protections
- Stack depth overflow prevention
- Stack buffer overflow detection
- No ASLR
- Kernel code considered trusted
- Userspace threads, not processes
- Kernel/user boundary still being fully fleshed out
- (Generally) Single application
- Highly dependent on particular SoC, config, application developer

- RO/NX memory protecitons
- Stack depth overflow prevention
- Stack buffer overflow detection
- Kernel and userspace ASLR
- Mitigations for many kernel vulnerabilities via KSPP
- Process isolation
- Mature kernel/user boundary
- Multi-application/user/tenant
- Generally independent of particular arch/SoC and application

Zephyr Security: Other Resources

- ELC / OpenIoT NA 2018 presentation by Andrew Boie, https://schd.ws/hosted_files/elciotna18/db/Boie%20-%20Retrofitting%20Zephyr%20Memory%20Protection.pdf
- Zephyr usermode docs, http://docs.zephyrproject.org/kernel/usermode/usermode.html

What is Fuchsia?

- Microkernel-based operating system
- Primarily developed by Google, but open source
  - Rumored to be replacement for Android and/or ChromeOS
- Targets modern hardware (phones, laptops)
  - 64-bit Intel and ARM application processors
- (Object) Capability-based security
- Work in progress

Fuchsia: The Zircon Microkernel

- Initially derived from Little Kernel (LK)
  - Embedded kernel / RTOS similar to FreeRTOS
  - Used in Android bootloader, Trusty TEE
- Extended/rewritten to be a microkernel
  - Support for 64-bit, user mode / process model, object capabilities, IPC, virtualization, ...
- The only part of Fuchsia that runs in supervisor mode
  - Drivers, filesystem, network run in user mode!

Fuchsia Security Mechanisms

- Microkernel security primitives
  - (regular) Handles
  - Resource handles
  - Job policy
  - vDSO enforcement
- Userspace mechanisms
  - Namespaces
  - Sandboxing

Fuchsia: (Regular) Handles

- Only way (usually) that userspace can access kernel objects
  - They are object capabilities
  - Uses a push model where client creates handle and passes it to a server
- Per-process (like file descriptors) and unforgeable
- Identify both the object and a set of access rights to the object
  - duplicate, transfer, read, write, execute, map, get_property, set_property, enumerate, destroy, …
- Can be duplicated with equal or lesser rights (if allowed duplicate)
- Can be passed across IPC (if allowed transfer)
- Can be used to obtain handles to "child" objects (object_get_child) with equal or lesser rights (if allowed enumerate)

Fuchsia: (Regular) Handles

- Good:
  - Separate rights for propagation vs use
  - Separate rights for different operations
  - Ability to reduce rights through handle duplication
- Of concern:
  - object_get_child()
  - Leak of root job handle (e.g. /dev/misc/sysinfo)
  - Refining default rights down to least privilege
  - Handle propagation and revocation
  - Operations that do not check rights
  - Unimplemented rights

Fuchsia: Resource Handles

- Variant of handles for platform resources
  - memory mapped I/O, I/O port, IRQ, hypervisor guests
  - specify allowed resource kind and optionally range
  - “root” resource handle allows access to all resources
- Can be used to obtain more restrictive resource handles
- root resource handle provided to initial process (userboot)

Fuchsia: Resource Handles

- Good:
  - Supports fine-grained, hierarchical resource restrictions
- Of concern:
  - Coarse granularity of root resource checks
  - Leak of root resource handle (e.g. /dev/misc/sysinfo)
  - Handle propagation and revocation
  - Refining root resource down to least privilege

Fuchsia: Job Policy

- Every process is part of a job
- Jobs can have child jobs (nesting)
  - Root job contains all other jobs/processes
- Job policy applied to all processes within the job
  - But can only be set on an empty job (no processes yet)
- Policies inherited from parent and can only be made more restrictive
- Policies include error handling behavior, object creation, and mapping of WX memory

Fuchsia: Job Policy

- Good:
  - Fine-grained object creation policies (per type)
  - Supports hierarchical job policies
- Of concern:
  - WX policy: not yet implemented and may pose problems for hierarchy
  - Inflexible mechanism
  - Refining job policies down to least privilege
    - Currently only used for device drivers and fuchsia job

Fuchsia: vDSO Enforcement

- Goal: vDSO is the only means for invoking system calls
- vDSO is fully read-only
- vDSO mapping constrained by the kernel
  - Can only occur once per process
  - Must cover entire vDSO
  - Can't be modified/removed/overwritten
- System call entry must occur from expected location in vDSO
- vDSO variants can expose a subset of the system call interface

Fuchsia: vDSO Enforcement

- Good:
  - Limits kernel attack surface
  - Enforces the use of the public ABI
  - Supports per-process system call restrictions
  - vDSO code is NOT trusted by kernel which fully validates system call arguments
- Of concern:
  - Potential for tampering with or bypassing the vDSO
    - process_write_memory()
  - limited flexibility, e.g. as compared to seccomp

Fuchsia: Namespaces and Sandboxing

- Namespace is a collection of objects that can be enumerated and accessed by name.
  - Composite hierarchy of services, files, devices
- Per component, not global
- Constructed by environment which instantiates a component
- Used and extended by components
- Sandbox is the configuration of a process's namespace created based on its manifest

Fuchsia: Namespace/Sandboxing

- Good:
  - No global namespace
  - Object reachability determined by initial namespace
- Of concern:
  - Sandbox only for application packages (and not system services)
  - Namespace and sandbox granularity
  - No independent validation of sandbox configuration
  - Currently uses global /data and /tmp
    - Docs do mention per-package /data and /tmp (future?)

Fuchsia: Bootstrap / Process Creation

- userboot creates devmgr and exits (not like init)
- devmgr creates zircon drivers and services, including svchost.
- devmgr creates fuchsia job and appmgr.
- svchost provides process creation facility for fuchsia processes
  - But caller must supply all kernel handles for new process.
- appmgr provides component creation facility
  - But appmgr is not allowed to create processes (because of the job policy of fuchsia job)
  - Caller identifies component, appmgr constructs namespace based on sandbox, uses svchost to create the actual Zircon process.

Fuchsia: A Case for MAC

- A MAC framework could address gaps left by Fuchsia's existing mechanisms, e.g.
  - Control propagation, support revocation, apply least privilege
  - Support finer-grained resource checks, generalize job policy
  - Validate namespace/sandbox, support finer granularity
- It could also provide a unified framework for defining, enforcing, and validating security goals for Fuchsia.
  - As it has in Android.

Fuchsia: Back to the Future

- Our early work was in the context of capability-based microkernel operating systems.
  - Mach (DTMach/DTOS) and Fluke (Flask)
- We've revisited MAC & capabilities repeatedly.
  - SELinux & Unix file descriptors
  - SE Darwin & Mach ports
  - Android & Binder

Fuchsia & MAC: Design Options

- Entirely userspace, no microkernel support
  - Build on top of existing capability-based mechanism
- Mostly userspace, limited microkernel support
  - Minimalist extensions to capability-based mechanism
- Security policy logic in userspace, full microkernel enforcement for its objects
  - As in our prior work (DTMach, DTOS, Flask, SE Darwin)

Full Kernel Support for MAC

- The Flask security architecture, http://www.cs.utah.edu/flux/flask
- Userspace security server provides labeling and access decisions.
- Object managers bind labels to objects, enforce security server decisions
  - Both microkernel and userspace servers
- Microkernel provides peer labeling, fine-grained control over transfer and use.

Flask Approach to MAC: Benefits

- Assurable implementation
  - Direct support for labeling and access control in microkernel
  - Capability leak by userspace component can be mitigated by microkernel checks
  - Reduced assurance burden on userspace components
  - Disaggregated TCB - userspace object managers, limited trust in each
- Centralized security policy
  - Amenable to analysis, audit, management
- Support for flexible, fine-grained access control

Current Work

- Investigating creation and flow of handles among Fuchsia components
- Analyzing reachability of security-critical handles/objects in the system
- Assessing effectiveness of existing mechanisms
- Exploring options for providing MAC-like properties

Current Work - Examples

- VMO
  - [vdso/full]|userboot|*|bin/devmgr|+|bin/devmgr|*|svchost|+|svchost|*|sh
- Resource
  - root-resource|userboot|*|bin/devmgr|+|bin/devmgr|*|devhost:sys
- Channel
  - <2407-2408>|bin/devmgr|*|devhost:pci#3:8086:100e
  - <2407-2408>|bin/devmgr|*|svchost

Fuchsia vs Linux OS security

- RO/NX memory protections
- Stack depth overflow prevention
- Stack buffer overflow detection
- Kernel and userspace ASLR
- Process isolation
- Self-protection not examined yet
- Small, decomposed TCB
- Object capabilities

- RO/NX memory protections
- Stack depth overflow prevention
- Stack buffer overflow detection
- Kernel and userspace ASLR
- Process isolation
- Mitigations for many kernel vulnerabilities
- Large, monolithic TCB
- DAC, MAC

Wrap Up

- Zephyr and Fuchsia are each seeking to advance the state of OS security for their respective domains.
- Much work remains to be done for the security of both of them.
- Get Involved!

Questions?

- Stephen: sds@tycho.nsa.gov
- James: jwcart2@tycho.nsa.gov