COMP5900T Notes

Formerly a Midterm Study Guide

William Findlay

March 11, 2020

Contents

1 Introduction **1**
1.1 General-Purpose vs Task-Specific OS 1
1.2 Activity: Design a Task-Specific OS 1

2 Secure OS **2**
2.1 Definition of Secure OS 2
2.2 Reference Monitor 2
2.3 Reference Monitor Desirable Properties 3
2.4 Access Control Fundamentals 4
2.5 Activity: Design a Reference Monitor for a Real Life Situation 5

3 Multics **5**
3.1 Segments 6
3.2 Answering Service 6
3.3 Supervisor 6
3.4 Segment Descriptor Word 6
3.5 Multics Access Control 7
3.6 Activity: Using Multics 8
3.7 Activity: Multics Security Evaluation 8

4 UNIX (and UNIX-Like) Security **9**
4.1 Permissions 9
4.2 Problems with UNIX Security 9
4.3 Activity: My Answers to UNIX Quiz 10

5 MAC **13**
5.1 Linux Security Modules 13
5.2 SELinux 13
5.3 Activity: Compare Custom SELinux Policy with Real SELinux Policy 14

6 UNIX Sandboxing **14**
6.1 OpenBSD `pledge(2)` 14
6.2 Linux `seccomp(2)` and `seccomp-bpf(2)` 14
6.3 FreeBSD `capsicum(4)` 15

7 Mobile OS Security **15**

7.1 Crypto Stuff 15
7.2 Android 16
7.3 iOS 17

8 IoT Security **18**
8.1 RIOT OS for Low-End IoT Systems 18
8.1.1 The Problem 18
8.1.2 Goals 18
8.1.3 Architecture 18
8.1.4 RIOT Kernel 19
8.2 IoT Security Talk by ARM CTO 19

1 Introduction

1.1 General-Purpose vs Task-Specific OS

General-Purpose.

- conventional OSes -> run lots of programs on a variety of hardware
- difficult to secure -> verifiability is a problem (and you could argue verifiability problem is getting worse)
- why do we use these? abstract hardware, don't want to reinvent the wheel
  - writing a new task-specific operating system for every machine, use case, is an exercise in futility
  - we trade security guarantees for features

Task-Specific.

- minimal OSes designed for one specific task
- e.g., electric thermometer computer
- much easier to secure

1.2 Activity: Design a Task-Specific OS

- e.g. write a task-specific OS for a Word Processor
  - custom scheduler for Word Processor threads
  - driver for monitor
  - driver for disk (long-term storage)
  - driver for memory (how to store work temporarily)
  - need to process hardware interrupts (from keyboard input)
  - maybe driver for a printer?
- the main takeaway: these things are *tedious*
  - we don't want to write a task-specific OS for every application
  - trying to write a program without an OS leads us to essentially design a task-specific OS

2 Secure OS

2.1 Definition of Secure OS

- a secure OS satisfies the reference monitor model
  - complete mediation
  - tamperproofing
  - verifiability
- trusted computing base
  - set of programs (kernel and some userspace) that we implicitly trust
  - if trust of TCB is broken, system is compromised
  - e.g. Linux kernel, passwd program, etc.

2.2 Reference Monitor

- defines sufficient and necessary properties for a system to securely enforce access control on security sensitive operations
- main components
  - interface
  - authorization module
  - policy store

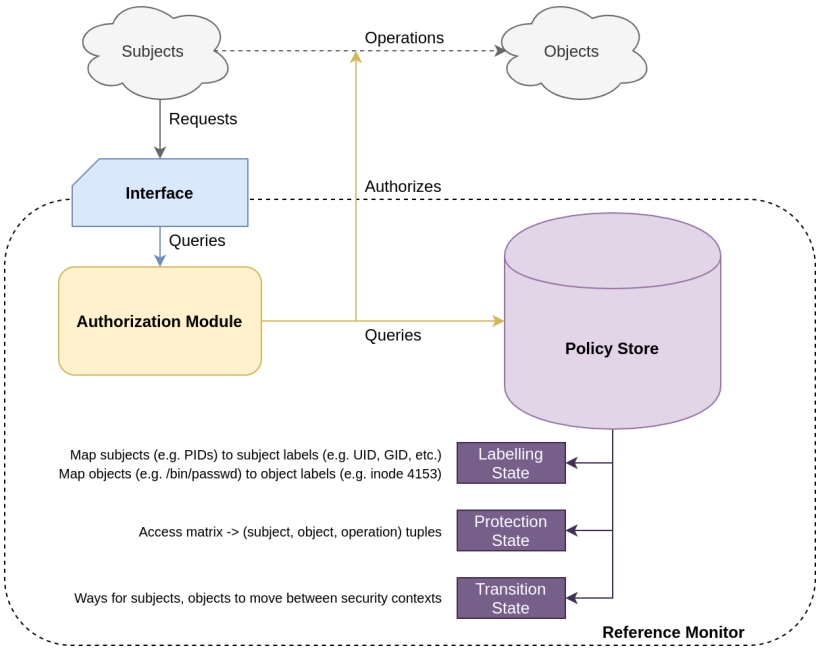

Figure 2.1: The reference monitor model.

Interface.

- defines queries to the reference monitor
- e.g. system calls

Authorization Module.

- take interface inputs, convert to query for the policy store
- query used to check authorization
- map PID to subject label, object references to object label

Policy Store.

- holds protection state, labeling state, transition state
  - define queries for each
- answer queries from authorization module

2.3 Reference Monitor Desirable Properties

Tamperproofing.

- reference monitor resistance to tampering
- i.e. unprivileged users cannot mess with the functionality of the reference monitor
- criteria:
  - how does the system protect the reference monitor from modification?
  - does the system protect the TCB?

Complete Mediation.

- reference monitor mediates all security sensitive operations
  - i.e. check subject capabilities before allowing them to operate on objects
- check permissions for system calls
- criteria:
  - how does the reference monitor mediate security sensitive operations?
  - does the reference monitor mediate security sensitive operations on all objects?

Verifiability.

- reference monitor is verifiability correct
- we can verify
  - complete mediation

    - tamperproofing
- in practice, verifiability is difficult to achieve
- criteria:
    - what is the basis for TCB correctness?
    - does protection system enforce security goals?

2.4 Access Control Fundamentals

DAC vs MAC.

- DAC
    - discretionary
    - users set permissions on objects they own
    - feel free to shoot yourself in the foot
- MAC
    - mandatory
    - usually augments DAC
    - administrator sets policy, cannot be changed by users

Access Matrix, ACLs, Cap-Lists.

- access matrix defines
    - x is objects
    - y is subjects
    - cells of allowed operations
- access matrix is abstract
    - in reality, implemented as ACLs or cap-lists, not both
    - sparse matrix, inefficient memory usage
    - ACLs can be derived from cap-lists, cap-lists from ACLs -> no need for both
- ACLs
    - rows of access matrix
    - define what objects the subject can interact with in what way
- cap-lists
    - cols of access matrix
    - define what subjects can perform what operations on the object

2.5 Activity: Design a Reference Monitor for a Real Life Situation

- safety deposit box
  - boxes are kept in a secure area
  - the secure area is under surveillance
  - need two keys to open (teller + client)
  - clients must provide documentation to prove their identity
  - once box is open, client moves to *another* secure area (without surveillance) to view contents
  - teller cannot see contents of box (only client may know its contents)
  - once client is finished, box is returned to vault, where teller and client lock it up again
  - only trusted vendors (locksmiths) may perform maintenance on the lock
  - locksmiths must be accompanies by bank personnel
- complete mediation
  - all access attempts to the box are mediated by the policy
  - teller + client
  - teller + locksmith
  - MAC: client cannot permit others to look at their box
- tamperproofing
  - the system cannot be tampered with unless a corrupt bank officer works with a locksmith
- verifiability
  - banks must conform to certain standards specified by the industry
  - entire process is simple enough such that it may be verified
  - who cares about verifiability? the banks, their customers (and potential customers), and insurance companies

3 Multics

- first modern OS
- invented lots of new concepts in OS design
  - segmented virtual memory
  - shared memory
  - hierarchical filesystems
  - online reconfiguration

- invented lots of new concepts in *secure* OS design
  - reference monitor
  - ACLs
  - ring protection
  - protection domain transactions
  - MLS security policies
- everything is:
  - a directory;
  - a subdirectory; or
  - a segment

3.1 Segments

- units of memory
- stored in:
  - process' context (things like DRAM)
  - secondary storage (things like tapes, disks)
- each process has a descriptor segment
  - what segments it has access to (segment descriptor words)
  - to add a segment to descriptor, specify its path (need permission)

3.2 Answering Service

- authenticates users on behalf of the supervisor
- load password SDW into its own segment
- check for a match, create user process with code and data segments for that user

3.3 Supervisor

- runs in ring 0 and ring 1
- implements all security sensitive functionality

3.4 Segment Descriptor Word

- lives in descriptor segment for each process
- each SDW describes a segment the process has access to
  - address
  - length

- ring brackets (rwe)
- ACL (rwe)
- gate transitions

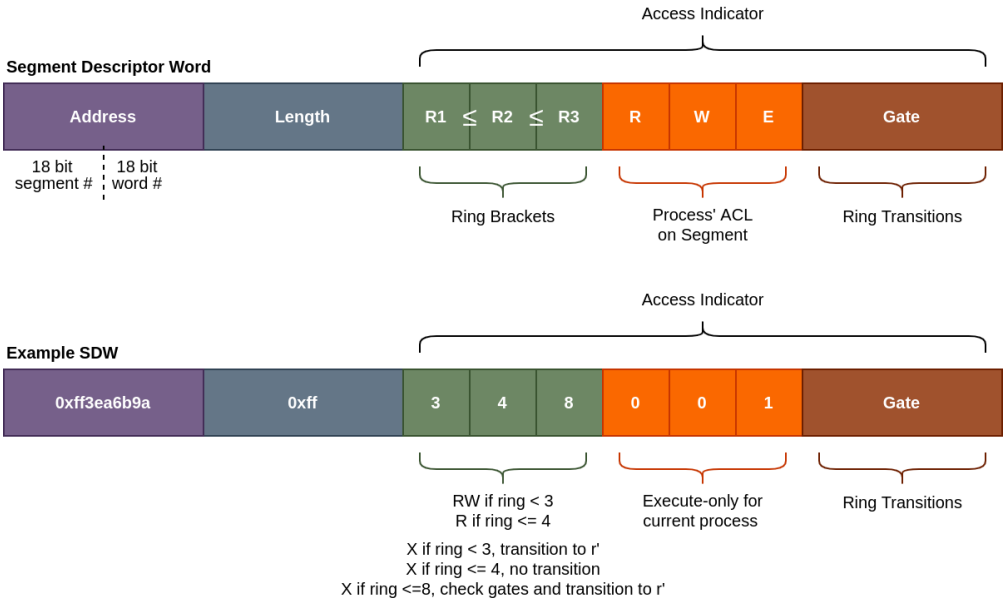

Figure 3.1: Segment descriptor word layout, accompanied by an example. In this example, the process can only execute the segment if it is running in ring 8 or lower. If it is running in a ring lower than 3, it must transition to execute. If it is running in a ring higher than 8, it must transition, but only if a gate allows it to do so. It cannot read or write to the segment due to its ACL.

3.5 Multics Access Control

ACLs.

- each object (segment) has an ACL that specified allowed operations for subjects (processes)
- ACL is stored in parent directory
- entries are
  - `r` read
  - `w` write
  - `e` execute (`x` in UNIX)
- for directories, 3 extra entries
  - `s`
  - `m`
  - `a`
- a process' ACL entry for a segment is also included in its SDW

Protection Rings.

- ring 0 (most privileged) to ring 63 (least privileged)
- Multics only implements first 8 (hardware restrictions)
- RW access
  - specified by R1 R2 ring brackets
  - below R1 implies RW
  - between R1 and R2 inclusive, R-only
  - above R2, no access
- processes can transition rings to execute
  - specified by R1 R2 R3 ring brackets
  - below R1 exclusive, we can execute but must transition to r'
  - between R1 and R2 inclusive, we can execute without transition
  - below R3 inclusive, we can execute if and only if a gate allows us to transition to r'
- transitioning back to original ring is also mediated
  - ring return gates

MLS (Multi-Level Security).

- mandatory access control
- subjects, objects have secrecy level
  - subjects can write to objects that have secrecy level $\geq$ their secrecy level
  - subjects can read from objects that have secrecy level $\leq$ their secrecy level
  - subjects can read from and write to objects that have secrecy level $\equiv$ their secrecy level

3.6 Activity: Using Multics

- Multics is hard to use
- difficult to do anything cool with it
- usability is a big reason why UNIX took over

3.7 Activity: Multics Security Evaluation

- Multics is touted as being very secure
  - ... but is it?
- in theory:

    - tamperproofing from rings
    - complete mediation from ACLs, rings, MLS
    - verifiability... somehow?
- in reality: better than UNIX, but still not great
    - vulnerabilities with exceptions trapping to lower rings
    - no initial hardware backing for ring transitions (this was later rectified)
    - only MLS is really MAC, the rest is still DAC (even rings can be changed if we can modify ACL in parent directory)
    - large codebase, verifiability is difficult

4 UNIX (and UNIX-Like) Security

4.1 Permissions

- UNIX uses DAC by default
- UGO model, augmented by ACL
- access matrix
    - subjects, objects matrix with operations in each cell
    - rows are cap lists, columns are ACLs (might be other way around)
    - in practice, we don't use the matrix
        - it is sparse, inefficient
        - ACL can be derived from cap list, cap list from ACL, no need to store both
- root user
    - UID = 0
    - access to entire system, incl. TCB
- setuid, setgid
    - run programs with euid = owner / egid = group respectively

4.2 Problems with UNIX Security

- UGO model is coarse (ACLs make things a bit better, but still problematic)
- TCB is mutable (anyone with root access can modify things)
- setuid binaries can be compromised -> this can be used to get a root shell, then entire TCB is compromised (see above)
- network facing daemons can be compromised (these are often listening on privileged ports)

4.3 Activity: My Answers to UNIX Quiz

Question 1. Access matrices should be thought of as an abstraction for management of permissions rather than a precise specification for implementation. Since an access matrix defines access control lists as columns and capabilities lists as rows, we are using at least O(nm) memory to store this matrix where n is number of subjects and m is number of objects. There is no practical need to do this, since we can derive all capabilities lists by taking all access control lists together, and conversely can define all access control lists by taking all capabilities lists together. Therefore, it makes the most sense to simply choose one of the two models and stick to it. With regards to why ACLs are the popular choice rather than capabilities lists, this perhaps boils down to the popularity of file abstractions for objects, especially in UNIX and UNIX-like systems; it simply makes more sense from a usability perspective to consider permission granularity in terms of files rather than in terms of subjects. One could also make the argument that it is more intuitive that a user be able to modify the access control list of a file they own to grant additional permissions, rather than inserting entries into another subject's capabilities list.

Question 2. This would, in my opinion, be a foolish and an incredibly dangerous assumption to make. Just because our reference monitor plays nicely and obeys the permissions specified in metadata, does not mean that the reference monitor of another system would do the same. Once the drive is mounted on another system, it is up to the OS running on that system to decide how to enforce the permissions we have set on the files in that drive. If we assume the attacker has full control over the system (e.g. they are plugging the USB device into their own GNU/Linux machine), we have absolutely zero guarantees as to the confidentiality of information on that drive. A much better choice would be encrypt each file on the drive or encrypt the filesystem itself.

Question 3. In UNIX and UNIX-like systems, we use a special executable (in Linux, this is called passwd) to change user passwords. This executable has a special permission bit set called the setuid bit. Applications run with the setuid bit set the effective user ID (eUID) of the process to be equal to the owner's UID. This means that the process effectively runs with the privileges of the owner. In the case of the passwd executable, its owner is root (i.e. the superuser with UID=0), and therefore it is able to access any file on the system, including the password file (which is owned by root anyway). In order to prevent the user from modifying other users' passwords, the passwd program performs extra checks to authenticate the user first and only allow them to modify their own password (for example, asking the user for their current password before asking them for the new password).

Question 4. From a security standpoint, the notion of running each application as a different user makes some degree of sense. This would follow the principle of least privilege, wherein processes would only have access to the information they absolutely need, and any additional privileges would need to be specified by object owners through modification of ACLs on objects. In fact, many background processes spawned by systemd on Linux (a modern UNIX-like OS) actually do run under their own users/groups. This is generally done to give them specific access to certain resources while limiting access to other resources on the system. While this is not exactly the same as what is being described here (where users run binaries and they execute with different permissions), it does illustrate the value that such a solution might provide.

Now, let us consider the usability perspective of such functionality. In short, it would be a usability nightmare. The modification of separate ACLs every time a user wants to run a new binary that needs access to one or more objects owned by that user would be extremely tedious. Something like the Multics ring protection model might work better in this context, wherein applications run under a certain ring and users can specify which rings should and shouldn't be allowed to access each object. This would provide a nice mix of fine-grained control and acceptable default permissions on objects such that users wouldn't constantly need to modify ACLs whenever running new programs.

Question 5. As I explained in the previous question, there are other users on a UNIX or UNIX-like system beyond whatever human is actually using the machine, and the other category can apply to processes running under these "users". For example, on Linux, the dbus daemon is spawned under the dbus user, in order to grant the daemon specific, and more fine-grained privileges over objects on the system.

Question 6. When we consider the security of a system like UNIX, I think we have to consider two factors: what was the system designed to do? and what was/is the system actually used for? UNIX was written by a bunch of bored hackers at Bell Labs who wanted to play around with a PDP-11. In order to justify the time they wasted on a side project, they gave it a nice spin when presenting it to managers. Bored hackers who just want to mess around don't care about security; they care about being able to do as much with their system as possible. Fortunately (or perhaps unfortunately, from a security perspective), UNIX took off in a big way. This means that now a system that was designed without many security considerations at all was now seeing widespread use across both academia and eventually industry. Of course, this security would later be improved by extensions such as SELinux that follow the FLASK model (it HAD to be improved, since UNIX's security model simply

couldn't keep up with real-world applications of the system, especially in industry after the advent of the internet).

From a security perspective, the all-or-nothing approach taken by UNIX doesn't make much sense; systems like Multics that supported finer-grained access control through multiple levels will naturally be better suited to extremely security-critical use cases. That being said, the usability benefits of UNIX's security model should not be ignored, particularly from the perspective of technical users who just want to hack on their systems and don't necessarily care about the consequences of granting too much access to the executables they run. I maintain the opinion that UNIX and UNIX-like systems offer absolutely unparalleled convenience and usability for application programmers and systems programmers who want to be as productive as possible. At the risk of quoting Todd Howard, it just works.

Question 7. In general, I agree with basically everything Jaeger says about the security of UNIX. I think that the crux of his argument is essentially that the scope and extensibility/modifiability of the UNIX TCB makes it essentially unverifiable. This lack of verifiability is coupled with the fact that the default protection system on UNIX is far too permissive with respect to user processes being able to access all objects owned by that user and superuser processes being able to ignore permissions entirely. Without verifiability, complete mediation and tamperproofing completely fall apart, as there is no way to verify whether these properties even hold in the first place.

With respect to complete mediation, I would give UNIX a grade of about 50% (of course this is somewhat oxymoronic, since complete mediation should be, by definition, complete). When we assume every binary run by users in userland is playing nicely, sure the UNIX model works fine. But as soon as we introduce factors like background processes (of which the user does not necessarily even have knowledge), binary exploitation through things like buffer overflows, setuid binaries that may be lacking sufficient checks (and thus vulnerable to TOCTOU exploits), and the implicit membership of all root processes within the trusted computing base, complete mediation is thrown out the window.

Tamperproofing also gets a pretty mediocre grade or about 50%. Since all root processes are implicit members of the TCB and a user with root privileges is able to freely modify and extend the kernel (and thus the reference monitor), we have absolutely no guarantees with respect to the integrity of the TCB at any given time. This is exacerbated by how potentially easy it is for a malicious user to escalate their privileges in the first place (see the previous paragraph).

Finally, I give verifiability a grade of 20%. I think Jaeger absolutely nails it when he says

that the extensibility of modifiability of the UNIX TCB and reference monitor renders the system essentially unverifiable. This lack of verifiability is a major contributing factor to the low scores that I gave the previous two properties.

5 MAC

5.1 Linux Security Modules

- a framework for implementing security policy with kernel modules
- added LSM hooks to security-sensitive operations
- new LSM module type can use these hooks to enforce policy
- SELinux is an example of an LSM module

What is Good About LSM?.

- low overhead (debatable)
- flexible (you can do a lot of different things with it)
  - can implement MAC
  - can implement POSIX capabilities (whitelist above DAC)
- users can plug and play security policies they want
  - can use SELinux, POSIX capabilities, or anything else
- provides a general solution to Linux security, rather than forcing one or two options

5.2 SELinux

- implements FLASK model (a MAC model) in Linux with LSM
- how policy works?
  - default policy
  - enforcing mode or permissive mode (permissive logs violations but does not enforce)
  - can configure policy with booleans
- every subject has a label, every object has a label
  - policy is defined as a whitelist on labels
  - everything else is denied by default
- SELinux implements
  - label/role type enforcement
  - MCS (multi-category security)

- MLS (multi-level security)

5.3 Activity: Compare Custom SELinux Policy with Real SELinux Policy

- SELinux policy is **complicated**
  - a lot of factors to consider
- they define their own language for defining policy
  - difficult to use, need to read documentation
- a lot of experience required to write policy, better to just use booleans

6 UNIX Sandboxing

- general pattern
  - main loop, not isolated
  - work loop, become isolated
- system call interposition is common
- pitfalls often include
  - coarse granularity of policy
  - incomplete policy (e.g. `open` but not `openat`)
  - mistakes by developers
  - developers needing to modify their applications (no transparency)
  - accidental ability to escape sandbox

6.1 OpenBSD `pledge(2)`

- place system calls intro groups
  - e.g. `stdio` for `read`, `write`, etc.
- process "pledges" itself to only make calls from specified groups
- allowing `execve` family calls allows us to circumvent pledge by simply executing new code
- very coarse-grained option, not really effective at sandboxing

6.2 Linux `seccomp(2)` and `seccomp-bpf(2)`

- process makes `seccomp` call
- only allow 4 system calls by default

    - `read`
    - `write`
    - `sigreturn`
    - `exit`
- other system calls can be whitelisted
    - use classic BPF programs to do this (`seccomp-bpf`)
    - filter based on system call, even system call arguments
- problems?
    - holes in policy (e.g. allowing `openat` but not `open`)
    - TOCTOU race conditions in BPF program (i.e. if you write a filter based on file path)

6.3 FreeBSD `capsicum(4)`

- prepare capabilities
    - assign capabilities to objects before we enter capability mode
    - each object gets a unique token, capabilities are predefined
- make `cap_enter` call
- once we are in capability mode, lose access to global namespaces
    - we can only access paths we already have capability to access
- probably the most sensible option

7 Mobile OS Security

7.1 Crypto Stuff

Symmetric Key.

- share key over some secure channel (out-of-band) or with key sharing scheme
    - e.g. DH key agreement
- encrypt and decrypt with that key
- problems?
    - no non-repudiation
    - hard to share or agree on keys (DH makes it better, but it has weaknesses)

Asymmetric Key (Pub, Priv).

- encrypt message with their public key

- they decrypt with their private key
- problems?
  - PKI necessary
  - distribute keys
  - figure out if public key is authentic

Digital Signatures (with Asymmetric).

- sign the hash of the message with your private key
- others verify with your public key

7.2 Android

App Installation/Update.

- how it works
  - if app is already installed then check if it's same certificate, if so then it's an update
  - if it is a new app and it is sharing UID with same cert then add to existing UID
  - if it is a new app and it is not sharing UID then assign new UID (initial installation)
  - permissions are assigned to UID, so if UID sharing is turned on then the UID gets union of permissions from all apps that use that UID
- app signing
  - developers sign their own certificates
  - use certificates to sign manifest, add file signatures to manifest
- problems?
  - developers using publicly available keys
  - signature stripping attack (catch signature, strip, re-sign)
  - attacker can flag downgrade as an upgrade (but they still need the key to sign the app)
  - trust-on-first-use is not a great model (PKI would be better, but difficult to change this after the fact)
  - no way to enforce certificate expiration, revocation

Authentication.

- authentication services run in Trusty TEE
  - fingerprintd for biometric, gatekeeperd for PIN

- daemons pass data to specific component in Trusty TEE, replies to user with AuthToken

Trusty TEE.

- similar to iOS secure enclave
- SoC (system on a chip), separate OS that runs on a separate processor
  - isolated from the rest of the system
  - exposes an API, main OS uses a driver to interact with it
- Trusty TEE is **not Linux**
  - it is based on Little Kernel
- houses the entire TCB of the Android device
  - cryptographic primitives, keymaster, gatekeeper, keystore
- keys stay in TEE

SELinux in Android.

7.3 iOS

Hardened WebKit JIT Mapping.

- JIT = just in time compilation
  - great for performance
  - but you can't sign it (because code is generated dynamically)
  - so what do we do?
- make sure code can't be tampered with!
  - have a memory region marked execute-only
  - have another mapping to same region marked readable and writable
  - lose the mapping of the writable location, but keep a special `memcpy` function that keeps track for us
  - now attacker doesn't know where the writable location is, but JIT compiler still can do its job

Secure Enclave Processor.

- SoC solution, similar to Android Trusty TEE
- keeps track of user keybag
  - four key types, all ephemeral and session-bound to varying degrees
  - all keys wrapped with master key

- master key
  - derived from SEP UID and use passcode
  - used to encrypt/decrypt all user keys
  - once key is set, encrypt it, lose the original
- SEP UID
  - set via a fuse in factory, non-mutable
  - important takeaway: **Apple has no access to user keys**, total privacy

8 IoT Security

8.1 RIOT OS for Low-End IoT Systems

8.1.1 The Problem

- low-end IoT cannot run conventional OSes
- need to support a variety of architectures and run with low overhead
  - limited memory in MCU (RAM + ROM)
  - 8-32 bit architecture support
  - various external devices
- need to design a new OS from scratch to meet requirements

8.1.2 Goals

- comply with network standards
- comply with system standards (C99)
- unified APIs across hardware
- modularity
- static memory
- vendor independence
- open source (free software)

8.1.3 Architecture

- microkernel (`core` → kernel + data structures)
  - in a microkernel, trusted userspace software makes system (API) calls on behalf of other userspace software
- `drivers` (implementation of device drivers)
- `periph` (unified access to MCU peripherals)

- `pkg` (imports third party components)
- `sys/net` (system libraries → crypto, networking, filesystems, etc.)

8.1.4 RIOT Kernel

- provides
  - multi-threading
  - context switching
  - IPC
  - synchronization primitives (mutex, semaphore, etc.)
- components make API calls to the `core` API (kernel's API)

Multi-threading.

- overhead in exchange for
  - easier code import
  - logical task separation
  - task prioritization
- what is the overhead?
  - memory for thread control block
  - memory for stack space
  - memory for CPU registers
- light-weight synchronization
  - mutex, semaphore, messaging
  - optional kernel submodule, compiled as needed
- multi-threading is optional
  - eliminate memory overhead when it's simply not feasible

Scheduling.

8.2 IoT Security Talk by ARM CTO