COMP5900 OS Security

Readings and General Class Notes

by

William Findlay
Tri Do

February 25, 2020

Contents

1 Introduction

- trusted computing base
  - applications that are essential to functioning of the OS
  - e.g., passwd
  - these would probably be okay to talk about for OS vuln. but kernelspace code is preferred

1.1 Bloom's Taxonomy

- course targets top 3 sections (for evaluation)
  - create
  - evaluate
  - analyze
  - (some understanding)

1.2 What is an OS?

- kernel
- essential applications (systemd, passwd, etc.)
- what does it do?
  - scheduling
  - network stack
  - file systems
  - block I/O on disk
  - hardware interrupts (e.g. I/O)
  - at least basic access control (memory protection, etc.)
  - often runs in supervisor mode (apparently not necessarily? But I don't agree with this...)

1.3 What is the Most Secure OS?

- probably something task-specific

1.4 Group Activity: Come up with an OS-less implementation for a word processor

- interrupt handling for keyboard
  - need at least some basic scheduler that can pause and resume main execution
- interface with monitor for graphical display
- block I/O driver for disk
  - filesystem to organize data
- hmm... this is starting to feel like we just implemented our own task-specific OS
  - that's the main takeaway here!

2 Defining a Secure OS

2.1 Paul Chapter 5.0 - 5.2

2.1.1 Intro

- early security had same challenges we face today
  - protecting programs from others
  - restricting access to resources
  - "protection" means mostly memory access control
- memory is important
  - holds data
  - holds programs
  - I/O devices through memory address and files
  - files -> both main memory and secondary storage
- early protection
  - virtual addresses
  - access control lists
  - limited process address space
  - these fundamentals are still used today
- Multics
  - security very influential in its early design
  - original UNIX was heavily based on Multics

2.1.2 Memory protection, supervisor mode, and accountability

- batch processing
  - prepare jobs ahead of time and submit them together as a batch job
- time-sharing systems
  - allowed shared use of a single computer
  - (preferable to batch jobs from a usability standpoint)
  - same way single-user computers work today with one user running many programs
- resource conflicts
  - processes running simultaneously can try to access the same resources
  - intentionally or otherwise
  - if a program could access full memory of the machine, errors could corrupt OS data or code
- supervisor
  - runs with higher permissions in the protection CPU (ring 0, 1, 2)
  - no other program can alter the privileged bit
  - a special machine instruction immediately transfers control to the supervisor
- privileged bit
  - process is running in supervisor mode
- descriptor register
  - holds a memory descriptor that describes base and upper bound

    - lowest addressable memory by a process and a number of words from that point that are addressable
- limitations of memory-range based protection
    - all-or-nothing mode of control
    - either you have full access or no access
    - allows full isolation, but not fine-grained sharing
- segment addressing with access permissions
    - segment = collection of words representing a logical unit of information
    - descriptor segment per process maintained by OS
        - holds segment descriptors that define addressable memory and permissions
    - descriptor base register points to memory descriptor of active process
- permissions on virtual segments
    - R non-supervisor can read
    - W can be written to
    - X can be executed
    - M run in supervisor mode (if X)
    - F all access attempts trap to supervisor
    - now the same physical segment can be given different access for different processes
- accountability, UIDs, and principals
    - UID (maps users to a unique identifier)
    - "principal" -> abstracts the entity responsible for code execution from the actual user or program actions
    - UID is the primary basis for granting permissions
- roles
    - assign distinct UIDs to distinct privileges
    - should follow principle of least privilege

2.1.3 Reference monitor, access matrix, security kernel

- reference monitor
    - concept that "all references by any program to any other program, data, or device are validated"
    - conceptualized as one reference monitor, but in practice, would be a lot of reference monitors working together
- access matrix
    - 2D matrix of subjects, objects
    - taking a row (subject) gives a capabilities list
    - taking a column (object) gives an access control list
    - each intersection in this matrix defines a set of permissions
- security kernel
    - reference validation
    - audit trails via audit logs (user X did Y at time Z)
        - these might not necessarily need to be tamper-proof, depends on needs
    - needs to be:
        - tamper-proof

        - always invoked (not circumventable)
        - verifiable (needs to be minimal / small enough to make this possible)
- protection mechanisms
    - ticket-oriented (capabilities)
        - access token allows entry to an event, as long as ticket is authentic
    - id-based
        - authorization lists based on ID

2.2 Jaeger Chapter 1

- general-purpose -> complex
- task-specific -> not so complex
- general purpose OS are hard to secure because of their complexity
- ensuring security depends on securing
    - resource mechanisms
    - scheduling mechanisms

2.2.1 Secure OS

- enforce security goals despite the threats faced by the system
    - implement security mechanisms to do this
- secure OS possible?
    - probably not
    - a modern OS by definition can probably never be 100% secure
    - security as a negative goal
- understanding secure OS requires understanding
    - security goals
    - trust model
    - threat model

2.2.2 Security Goals

- define operations that can be executed by a system while remaining in a secure state
    - i.e. prevent unauthorized access
- high level of abstraction
- define a requirement that the system's design can then satisfy
- we want to maintain: secrecy, integrity, availability
    - secrecy = limit read access for objects by subjects
    - integrity = limit the write access for objects by subjects
    - availability = limit the resources that a subject may consume (i.e. no DoS)
- subjects
    - users, processes, etc.
- objects
    - resources of the system that subjects may or may not access in various ways
    - e.g. files, sockets, memory

- security goals can be
  - defined by function (e.g. principle of least privilege)
  - defined by requirements (e.g. simple-security property)

2.2.3 Trust Model

- trust model
  - defines the set of software and data we trust to help us enforce our security goals
  - we depend on this model to correctly enforce our security goals
- trusted computing base
  - trust model for an operating system
- TCB should **ideally** be minimal to the extent that we require
  - in practice, this is a wide variety of software
- TCB includes
  - all OS code (assuming no boundaries as in a monolithic kernel)
  - other software that defines our security goals
  - other software that enforces our security goals
  - software that bootstraps the above
  - software like Xorg that performs actions on behalf of all other processes
- a secure OS developer needs to prove their system has a viable trust model
  (1) TCB must mediate all sensitive operations
  (2) verification of the TCB software and data
  (3) verification of TCB tamper-resistance
- identifying and verifying TCB is a complex and non-trivial task

2.2.4 Threat Model

- defines a set of operations that an attacker may use to compromise the system
- assume a powerful attacker who
  - can inject operations from the network
  - may be in control of non-TCB applications
- if the attacker finds a vulnerability that violates secrecy or integrity goals, the system is compromised
- highlights a critical weakness in commercial OSes
  - assume that all software running on behalf of a subject is trusted by the subject
- our task? protect the TCB from threats
  - easier said than done
  - user interacts with a variety of processes
  - users are untrusted
  - TCB interacts with a variety of untrusted processes

2.3 Jaeger Chapter 2

2.3.1 Protection System

- protection system consists of

    - protection state
    - protection state operations
- protection state
    - what operations can subjects perform on objects
- protection state operations
    - what operations can modify the protections state
    - (this is distinct from the operations that the protection state describes)

Lampson's Access Matrix.

- protection state
    - rows = subjects
    - cols = objects
    - select row -> capability list
    - select col -> access control list
    - each entry specified privileges subject -> object
- protection state operations
    - determine which processes can modify cells

Mandatory Protection Systems.

- we don't want untrusted processes tampering with the protection system's state by adding subjects, objects, operations
- discretionary access control system (DAC)
    - an access control system that permits untrusted modification
    - *safety problem*
        - how do we ensure that all possible states deriving from initial state will not provide unauthorized access
- mandatory protection systems / mandatory access control (MAC)
    - protection system can only be modified by trusted administrators via trusted software
    - mandatory protection state -> subjects and objects are represented by labels
        - state describes operations subject labels -> object labels
    - labeling state
        - state for mapping subjects and objects to labels
    - transition state
        - describes legal ways subjects and objects may be relabeled
- set of labels being fixed in MAC doesn't mean that set of subjects/objects are fixed
    - we can dynamically assign labels to created subjects and objects (labeling state)
    - we can dynamically relabel subjects and objects/resources (transition state)

2.3.2 Reference Monitor

- classical access enforcement mechanism
- takes request as input
- outputs binary response -> is the request authorized or not?

- main components?
  - interface
  - authorization module
  - policy store

Reference Monitor Interface.

- defines queries to the reference monitor
- provides an interface for checking security-sensitive operations
  - (security-sensitive means it may violate security policy)
- e.g., consider the `open` system call in UNIX (reference monitor decides what is allowed / disallowed)

Authorization Module.

- takes interface inputs, converts to a query for the policy store
- this query is used to check authorization
- authorization module needs to map PID to subject label and object references to an object label
- needs to determine the actual operation (s) to authorize

Policy Store.

- database that holds protection state, labeling state, transition state
- answers queries from the authorization module
- has specialized queries for each of the three states

2.3.3 Secure Operating System Definition

- a secure operating system's access enforcement satisfies the reference monitor model
- the reference monitor model defines the necessary and sufficient properties of a system that securely enforces MAC
- three guarantees:
  (1) complete mediation -> ensure access enforcement for all security-sensitive operations
  (2) tamper proof -> cannot be tampered with from outside the TCB (untrusted processes)
  (3) verifiable -> small enough to be subject to testing, analysis

2.3.4 Assessment Criteria

Complete Mediation.

- how does the reference monitor interface ensure that all security-sensitive operations are mediated correctly
- does the reference monitor mediate security-sensitive operations on all system resources
- how do we verify complete mediation?

Tamper Proof.

- how does the system protect the reference monitor and its protection system from modification?
- does the protection system protect TCB programs?

Verifiable.

- what is the basis for TCB correctness?
- does the protection system enforce security goals?

2.4 Class Review

Other Verifiability Techniques.

- code review
- regression (TDD)
- formal verification
- fuzzing

3 Multics

3.1 Jaeger Chapter 3

- Multics = the first modern OS
- invented many fundamental OS concepts
  - segmented virtual memory
  - shared memory for multiprocessors
  - hierarchical filesystems
  - online reconfiguration
  - (many others)
- invented many fundamental *secure OS* concepts
  - reference monitor
  - ring protection model
  - protection systems
  - protection domain transactions
  - multilevel security policies
  - (many others)

3.1.1 Jaeger 3.1

- Multics emerged from the CTSS (Compatible Timesharing System) system
- early timesharing systems could support a few jobs at a time
  - vs batch processing could only do one at a time
- Project MAC (Multi-Access Computer) aims to create a general-purpose timesharing service to support large number of users simultaneously

    - this would require several functions such as multiplexing of devices for different processes, scheduling processes, communications between processes, and protection from other processes
- IBM wasn't interested in the idea, so GE took over
- although it was considered a failure (bigger, slower, not reliable), it brought important OS and security features that would lead to UNIX
- Multics cost around $7 million initially (only 80 licenses sold)
- last department to use Multics was the Canadian Department of Defense (LOL we got issues boi)

3.1.2 Jaeger 3.2

- layered architecture
- resources organized hierarchically

Multics Fundamentals.

- processes are executables (they run program code)
- all code, data, I/O devices,etc that processes have access to are called segments
- hierarchy of directories that may contain
    - sub-directories
    - segments
- a process's protection domain defines the segments it can access and what operations it can perform on them
- segments can be stored in a process's context or secondary storage
- each process has its own descriptor segment which contains segment descriptor words that point to the segments the process has access to.
- if the segment is not in the descriptor segment, it must name the segment (like a file path)
    - if the process has permission, the segment is added to its descriptor segment
- descriptor segment has a set of *segment descriptor words* that refer to all directly accessible segments

Multics Security Fundamentals.

- *answering service* process implements the login system
    - to authenticate the user, answering service loads password SDW into its own descriptor segment
    - Multics supervisor must authorize this
- *supervisor* implements all security-critical functionality such as authorization, segmentation, I/O, scheduling, etc.
- *protection rings* protect the supervisor from other processes
    - the rings form a hierarchy with ring 0 as the most-privileged
    - the supervisor's segments are assigned to ring 0 and 1
    - higher rings cannot access lower rings

- if the user and password match, the answering service creates a user process with the appropriate code and data segments for that user
- each live process segment is accessed by the SDW
- the SDW contains
  - the address of the segment in memory
  - its length
  - ring brackets
  - process's permissions
  - number of gates for the segment (code segments only)
- ring brackets
  - define access according to process rings
  - code segment also has a call bracket that defines whether it can be run
  - also define access to protection domain transition rules

Multics Protection System Models.

Access Control Lists (ACLs).

- each ACL entry specifies the process with a user identity with operations that it can perform on this object.
- segments can have r,w,e permissions
- directories can have r,w,e,s,m,a
- segments ACL are stored in its parent's directory
  - this means that checking for permission or any modification to segment ACL is done through the parent directory
- if the user with the process has permission for operation on the object, then the reference monitor authorizes the creation of SDW with those permissions.

Rings and Brackets.

- aside from ACL, Multics limits access based on protection rings as well
- each segment contains a ring bracket that has r,w,e permission of processes on that segment
  - a segment's bracket defines the ranges of ring that can have certain permission (rwe) to the segment
- suppose a process from ring r wants to access a segment with an access brackets of $(r1, r2)$ and we have these rules:
  - if $r < r1$, then the process can read and write to the segment
  - if $r1 \leq r \leq r2$, then the process can read the segment only
  - if $r2 < r$, then the process has no access to the segment
- the above rules ensure that lower rings are more privileged and has more access to segments than higher rings
- call brackets define execute permissions on code segments as well as transition states
  - can freely transition to lower rings
  - segment gates control transitions into higher rings

Multilevel Security.

- each directory stores a mapping from each segment to a secrecy level
- Multics also stores an association between each process and its secrecy level
- operations are authorized as follows:
  - writes require a segment to have segment secrecy $\geq$ process secrecy
  - reads require a segment to have segment secrecy $\leq$ process secrecy
  - read/writes require segment secrecy = process secrecy
- MLS prevents information leakage

Multics Protection System.

- ACL, Ring Brackets, and MLS are all taken together
  - ACL and Ring Brackets are discretionary, while MLS is mandatory
- examples
  - requested operation is READ:
    - ACL checks if user has read access
    - MLS policy checks to verify that the object's secrecy level is dominated by or equal to the process
    - the access bracket is checked to ensure the process has access to the object's segment
  - requested operation is WRITE:
    - ACL checks if user has write access
    - MLS policy checks to verify that the object's secrecy level dominates or equal to the process
    - the access bracket must allow the current ring write access
  - requested operation is EXECUTE:
    - similar to write operation
    - the process must have execute permission in the segment's ACL
    - MLS policy must allow for reading the segment
    - call bracket is used instead of access bracket, and call bracket must allow execution
    - the request may result in protection domain transition
      - transition from process's current ring r to the ring specified by the segment in the call bracket r'
      - when a process invokes a code segment with a call bracket with r < r1:
        - the process must transition to r'(a lower privileged ring)
      - when a process invokes a code segment with a call bracket where r2 <= r <= r3:
        - the process transitions to r' by using one of the gates in the code segment as entry point

Multics Reference Monitor.

- Multics' reference monitor is implemented by the supervisor
- each Multics instruction either accesses a segment via a directory or via a SDW, so

authorization is performed on each instruction
- the supervisor also performs protection domain transitions as mentioned above
- a transition that requires to go through a gate segment needs to verify the following:
  - the number of arguments expected
  - the data type on each argument
  - access requirement for each argument (read, read/write)
- the gate segment is also known as the gatekeeper
- when we return from the more privileged ring to a lower privileged ring, we need to ensure that no information is leaked.
  - the supervisor copies arguments from its segment to another segment (accessible to the called procedure).
    - copying is necessary because we don't want unauthorized access from the calling procedure
    - the caller must also be careful not to copy unauthorized information (private keys), since the less-privileged code may be able to use to impersonate the higher-privileged code
  - Multics allows the caller to provide a gate for return (return gate).
    - multiple calls can result in a stack of return gates, so SDW is not suitable for return gates
    - the supervisor must maintain this stack of return gates for each process.
- the fundamentals of the reference monitor is implemented in ring 0
  - other utility such as file system search utility are in ring 1. determination of a directory or segment from a name is performed there, but authorization of whether this access is permitted is done in ring 0

3.1.3 Jaeger 3.3

Complete Mediation.

- how does the reference monitor interface ensure that all security-sensitive operations are mediated correctly?
  - Multics provides complete mediation at the segment level.
- does the reference monitor interface mediate security-sensitive operations on all system resources?
  - Multics mediates each segment access at the instruction level,Multics mediates memory access completely. Multics also mediates ring transitions, in both directions. Thus, the reference monitor provides mediation at memory and ring levels.
- how do we verify that the reference monitor interface provides complete mediation?
  - to verify complete mediation, we need to verify that ring transitions and segment accesses are mediated correctly. These operations are well-defined, so it is straightforward to determine that mediation occurs at these operations.

Tamper-proof.

- how does the system protect the reference monitor, including its protection system

from modification?
  - Multics' reference monitor is implemented by ring 0 procedures. Ring 0 procedures are protected by a combination of protection ring isolations and system-defined ring bracket policy. The ring bracket policy prevent processes outside of ring 0 from reading or writing reference monitor code or state directly.
- does the protection system protect all the trusted computing base programs?
  - Multics TCB consists of the supervisor (ring 0) and from ring 1 to ring 3. Ring 4 and above are standard user processing. Rings 0 to 3 can be considered part of the TCB.
  - discretionary nature of ring protection makes this difficult
  - if we compromise one process in TCB, it can undo all ring protection at its ring level
  - we can say that the TCB is securable, but that its tamper resistance is brittle

Verifiable.

- what is basis for the correctness of the system's trusted computing base?
  - the implementation of the Multics TCB is too large to be formally verified. This goal was not achieved even though they did try to aim to minimize the implementation as much as possible
- does the protection system enforce the system's security goals?
  - protection state: MLS secrecy protection is enforced as long as the TCB is not compromised
  - labeling state: Verifying the correct labeling of segments and processes is challenging, since most of the labeling is done manually. The Multics policy does not explicitly state how new processes and segments are labeled
  - transition state: Permits code from a less-privileged ring to transfer control to code in a more-privileged ring through either gates or return gates based on the call bracket rules. The security of these transitions depends on the correctness of the gates

3.2 Virtual Memory in Multics (Video)

Generating Address.

- 36 bits in total
  - 18 bit segment number
  - 18 bit word number
- i.e. 2^{18} segments and within each segment 2^{18} words

Instruction Format.

- segment tag
  - has segment number and word number
- address
- operation code

- external flag
- addressing mode (2 bits)
  - corresponds to one of four pointers
    (1) argument pointer
    (2) base pointer (bottom of stack frame)
    (3) linkage pointer
    (4) stack pointer (top of stack [frame])

Segment Number.

- can only be changed by ring 0 (supervisor mode) program

Word Number.

- can be changed by user mode programs

Switching Processes.

- all the kernel does is substitute a new descriptor base register (with a different segment number)

Apple II and Stuff (1982).

- introduced copy protection on memory / disks
- no virtual memory (they didn't need it, hobbyists didn't need to care about security)

3.3 Protection in an information processing utility (Graham 1968)

Challenges of protecting an IPU.

- lots of users
- many users using the system simultaneously
- more than one CPU, running multiple programs at a time
- sharing and not sharing
  - system could be used for applications that have sensitive data
  - system could also be used by applications that do want to share their memory with others (maybe on parts of it)

Why do we need protection?.

- protection is not strictly necessary for single-user applications, but still desirable
  - aid debugging
  - prevent application errors from propagating (good even in the benign case)
  - (localizes the source of the original error)
- more than one user makes things much more interesting
  - even simplest multi-user systems share the supervisor program (i.e. reference monitor state) between all users

- we need to protect this from being tampered with
- user information stored on multi-user system must be protected for privacy reasons

3.3.1 Defining Satisfactory and Unsatisfactory Protection Mechanisms

Protection Mechanisms Before Multics (Unsatisfactory).

- mode switch for supervisor instructions (paper calls them master and slave)
- memory bounds register (descriptor register)
- what is included in privileged instructions?
  - I/O instructions
  - changing mode switch
  - changing contents of memory bounds register
- mode switch can protect information on storage media that is not actively being looked at
  - but it cannot protect data in active memory... we need the memory bounds register for that
- what was wrong with the original solution?
  - all or nothing
  - either have all privileges, or have none
  - **unsatisfactory** for an IPU

Satisfactory Protection Mechanism Properties.

- isolation
  - should be possible to completely isolate one process from another
- sharing
  - should be easy and convenient to select which parts of a process should be shared with another process
- layering
  - single process layers of protection should be available to system and user
  - "need to know" principle should be easily applicable to a reasonable degree
- calling procedures across layers
  - should be do-able without any special programming

3.3.2 The Abstract Model

Segment Descriptor (Isolation, Sharing).

- pages are invisible to the user (managed by hardware)
- segments contain words of arbitrary length stored in pages
- addressing is done by (S, W) where S is segment number and W is word number
- descriptor needs to have
  - beginning of segment
  - length
  - access indicator
  - (ring number)

- every processor has exactly one segment descriptor

Location Counter.

- keeps track of next available location for memory assignment in current segment
- needs to have
  - (ring number)
  - procedure segment number
  - word number

Protection Rings (Layering).

- up to m rings
- ring 0 is most privileged, ring m is least privileged
- every segment is assigned to one and only one ring
- a process running in ring i has no access to any segment in ring $j < i$

4 Unix

4.1 Paul Chapter 5.3 Object permissions, file-based access control

Owners and groups.

- files have an owner and a protection group
  - UID and GID
- used for the UG component of UGO

Superuser (usually `root`).

- UID 0
- often called root, not necessarily
- has access to all files, resources in userspace, independent of protection settings

UGO model.

- three-tuple of permissions
  - `rwx`
  - for user, group and other
  - where:
    - user = owner
    - group = members of protection group
    - other = anyone else on the system
- `r` = read (can list contents for directories)
- `w` = write (can append for directories)
- `x` = execute (can cd for directories)
- `-` = no permission in that slot

- first flag is special and specifies what type of file it is
  - `-` = normal file
  - `d` = directory
  - `c` = character device
  - `b` = block device
  - `p` = named pipe
- can also be represented as a 12-bit number or a 4 digit octal
  - 3 most significant bits represent setuid, setgid, sticky bit
  - then the other 3 sets of 3 represent UGO

Default permissions and umask.

- default permissions are `666` for files and `777` for directories
  - `rw-rw-rw` and `rwxrwxrwx` respectively
- however, when creating a file or directory, we also use the `umask`
  - `umask` is negated and then anded with permissions to calculate actual permissions
  - `umask` is often `0022`, which removes write permissions for group and other
- the result?
  - `rw-r--r--` for files and `rwx-r-xr-x` for directories

Augmenting UGO with ACL.

- UGO sets base permissions, optional ACL for more fine-grained control
- check ACL first, fall back to UGO if no entry

4.2 Paul Chapter 5.4 setuid, setgid, RUID, EUID, SUID

- Setuid bit:
  - when a process runs a program with this bit set, the process's userid will be set to the owner's userid so that the calling process can have more resources
  - `s` means executable with setuid
  - `S` means setuid bit but not executable
- Setgid:
  - analogous to setuid, but applies to groups
- Real UID (RUID):
  - process' owner
- Effective UID (EUID):
  - determines privileges on resource access requests, changes to allow non-privileged user to access files (owned by root)
- Saved UID (SUID):
  - saves UID when a process needs to lower its privileges temporarily
  - EUID is saved as SUID and then changed
  - it can return its EUID to SUID later
- PID:
  - when a process forks, it creates another process with the same parent UID triple (RUID, EUID, SUID)

  - PID is new for child process
  - kernel version = TGID (thread group ID)
- TID
  - same as PID but for threads
  - kernel version = PID (not to be confused with userspace PID, which is TGID)

4.3 Paul Chapter 5.5 Directory permissions and inode-based example

- each directory has a list of entries structured as follows:
  - dir-entry = (d-name, d-inode), where d-name is directory name and d-inode is the file's inode
- directory permissions:
  - R: list contents
  - W: edit contents, however X is also needed to rename and delete files
  - X: traverse and search, allows access to inode meta-data
  - setuid: no meaning (used to mean something historically)
  - setgid: group value assigned to group created the dir-entry
  - t-bit(text/sticky bit): prevents deleting or renaming of other people's files within dir
    - Root and owner have all controls
- world-writable files:
  - If second last bit is w then it is world-writable
- a common system default for directories is 777 (default mask 022)
  - results in 755 (rwx for user, rx for others and groups)

4.4 Paul Chapter 5.6 Symbolic links, hard links, deleting (unlinking) files

- symbolic link (indirect alias)
  - different inode, different pathname, points to pathname
  - "a file whose datablock points to another file"
- hard link (direct alias)
  - same inode, different pathname
  - usually disallowed for directories, to prevent looping
- deleting a file
  - actually unlinking it
  - the file is only removed from the filesystem after the **last** link is removed
    - that is, the last reference to its inode
  - problem arises when considering **permanent removal**
    - users might expect data to be deleted, but in fact datablock is not necessarily overwritten
    - datablock may persist until it is overwritten by future data

4.5 Paul Chapter 5.7 Role-based (RBAC) and mandatory (MAC) access control

- DAC: Owner decides the permissions
  - subject who owns the object may grant permissions to other subjects
- MAC: System decides the permissions
  - permissions are mandatory
  - security policy administrator decides on subject permissions for every object
- RBAC: Permission is based on roles of users
  - subject is assigned one or more roles
  - roles have pre-assigned permissions

SE-Linux.

- provides MAC and other security features for the Linux kernel
- built on the Flask architecture

4.6 Paul Chapter 5.8 Protection rings: isolation and finer-grained sharing

- provide layered protection within processes, separation and sharing of subsystems
  - low ring = privileged
  - high ring = less privileged
  - ring 0 = supervisor/hypervisor
- introduced by Multics in 1960s
  - eventual hardware support for 8 rings
    - I think David said modern x86 can only do 4, though
  - modern systems like Linux only really use 2 rings (0 and 3)
    - supervisor and user mode respectively
- transfer control to stronger rings
  - e.g. for sensitive operations like I/O, changing permissions, etc.
  - require passage through segment gates
- transfer control to weaker rings
  - call simple shared services
  - always allowed
- transfer control within access brackets
  - always allowed

Segment Access Brackets.

- specify allowed ranges for ring access transfer
- access attempts outside access brackets are either denied or mediated by gatekeeper

Segment Gate Extension.

- gate list

    - specified entry points into a ring bracket
- gates are needed to cross into higher rings outside access bracket
- mediated by the gatekeeper software (when we are not in access bracket)
    - calling weaker ring -> allowed
    - calling ring within gate extension -> allow if transfer on gatelist
    - calling ring stronger than gate extension -> error

Cross-Ring Returns.

- crossing back to original ring also triggers mediation
    - may involve return gates

4.7 Paul Chapter 5.9 Relating subjects, processes, protection domains

- Protection domain
    - permission set associated with the objects a process can access

4.8 Jaeger 4.2.3 UNIX security evaluation

- UNIX fails to meet any of the three desirable properties of a secure OS reference monitor
    - mediation is not complete
    - reference monitor is not tamperpoof
    - reference monitor is not verifiable

4.8.1 Complete mediation

- check permissions for accessing a file or inode on *some* system calls
- reference monitor authorizes access to system objects that the kernel uses in its operations
- the problem?
    - (r, w, x) permissions are not expressive enough to mediate all access
    - an open file descriptor can be modified freely via `ioctl` or `fcntl`
- no authorization provided for some objects
    - e.g., network communication
- difficult to verify complete mediation
    - reference monitor is placed where security-sensitive operations are performed
    - *why is this a bad thing? maybe ask in class...*

4.8.2 Tamperproof

- reference monitor, protection system are stored in kernel **but...**
    - this doesn't guarantee tamperproof
- discretionary protection system (may be tampered with by any running process)
    - untrusted user process' can change permissions on that user's data freely
- kernel is not as protected from untrusted processes

    - no specific gates specified for ring escalation (procedures to verify system calls may be misplaced)
- user level processes have interfaces to manipulate the kernel
    - sysfs
    - procfs (note from William: I'm fairly certain this didn't exist until Linux)
    - kernel modules
    - netlink sockets
    - direct access to kernel memory through character devices (e.g., `/dev/kmem`)

Root and TCB issues....

- **ALL** root processes are part of the TCB
    - because these processes have access to all of userland, including other TCB processes
    - these processes can also interact with the kernel in dangerous ways
        - (including any aspect of protection system and reference monitor)
- anyone logged in as root can run any program as root
- unsecure root processes can be tampered with
    - e.g., root-owned daemons listening on open ports (may be vulnerable to binary exploitation)
    - e.g., TOCTOU in setuid binaries

4.8.3 Verifiability

- unbounded TCB size due to root processes implicitly being part of the TCB
    - this makes formal verification *impossible*
    - userland part of the TCB is effectively unverifiable
- kernel can be freely modified and extended (e.g., kernel modules)
    - this means we have the same problem in kernelspace as we do in userspace
- no complete mediation and no tamperproofing effectively means the system is unverifiable

4.9 Jaeger 4.2.4 UNIX vulnerabilities

Network-facing daemons.

- daemonized processes running in higher privileges (usually root)
- maintain network ports that are open to all parties
- these processes are part of the TCB since they run as root
    - therefore, they must protect themselves from bad input (malicious or accidentally problematic)
- several vulnerabilities reported for network-facing daemons
    - usually as a result of buffer overflows (binary exploitation)
    - e.g., the Morris Worm exploited `sendmail`, `finger`, and `rsh`
- vulnerable software has been patched, but protection is ad hoc and unverifiable

- remote login daemons, file transfer protocol, and network filesystems like NFS put a lot of trust in the network
  - network is implicitly untrustworthy

Rootkits.

- kernel modules may be used to load malicious code into the kernel
- can enable attacker privilege escalation, full control over system calls
  - e.g., hooking into `execve` system call to spawn a root shell when given a special keyword
- can completely hide themselves via modification of kernel data structures or obfuscation of output from specific kernel functions

Environment variables.

- `LIBPATH` specify path to library code
  - modifying this can force loading of attacker-provided library code which can supplant legitimate code used by many or all dynamic executables
- many other environment variables are used by processes
  - standard ones like `PATH`, etc.
  - non-standard ones like `MY_API_KEY`, `PATH_TO_IMPORTANT_DATA`, etc.
- environment variables are loaded from parent process
  - untrusted process can invoke a TCB program with environment variables that can compromise it
  - this is fine if we can verify all TCB programs, but we **can't do this**, as explained previously

Shared resources.

- problems arise when TCB processes share resources with untrusted processes
- common problem: sharing `/tmp`
  - untrusted processes can create files in `/tmp` and share access with others
    - (incl. TCB processes)
    - untrusted process can
      - guess the name of TCB file in advance
      - make it ahead of time
      - and grant access to TCB process
    - TCB process may have no idea anything is wrong
    - we can prevent this problem by checking for file existence on creation

TOCTOU attacks on vulnerable processes.

- class of attacks where untrusted processes change the state of the system between a check and an access
- classic example:
  - setuid process uses `access` system call to check permissions for RUID

    - then it opens the file (e.g., for writing)
    - meanwhile, a racing process has switched what that pathname points to between when the file was checked and when it was opened
    - now the privileges process is writing to a file that the user shouldn't have access to
- UNIX added a flag to prevent `open` from following symlinks
    - but untrusted processes may still manipulate the mapping between file names and objects

4.9.1 A summary

- main problems with UNIX
    - discretionary protection system
    - unbounded size of system TCB, both in userspace and kernelspace
    - many vulnerabilities described above
- converting UNIX to a secure OS is a *very* hard problem

5 MAC, LSM, SELinux

5.1 MAC

- mandatory access control
    - not discretionary
- MAC policy is set by administrator, and fixed
- MAC policy supersedes discretionary policy

5.2 SELinux

- provides Linux kernel with MAC policies
- implements the Flask flexible access control architecture in Linux

Type Enforcement and Labels.

- labels
    - objects and subjects get labels
    - for files, directories, sockets, special files, etc. -> extended fs attributes
    - for processes, ports, etc. -> managed by kernel tables
- label format
    - `user:role:type:level(optional)`
    - `user` = SELinux user, not same as Linux concept of users
    - `role` = what role the subject/object has
    - `type` = what type the subject/object is
    - `level` = used with MLS/MCS, optional
- type enforcement
    - for example:

        - it probably makes sense for a subject of type `x_t` to access an object of type `x_config_t`
        - it probably doesn't make sense for it to access an object of type `shadow_t`
    - we set policies to allow interactions
    - all other interactions are denied by default

Booleans.

- settings and policy defaults that you can enable or disable
- SELinux picks sensible defaults based on user feedback
- hundreds of these settings

Label Transitions.

- policy which specifies transition of labels when creating
    - subdirectories or
    - files within directories
- for example:
    - `/home/will` has type `:will_home_t`
    - add a transition for application with `vim_t` so that new files in `/home/will` become `:will_txt_t`
    - new file saved with vim under `/home/will/new.txt` will have label `:will_txt_t`

Multi-Category Security (MCS).

- MCS label on subject **must match** MCS label on object
- this is combined with type enforcement
    - allowed iff type enforcement rules are OK and MCS labels match

Multi-Level Security (MLS).

- levels of secrecy on subjects and objects
- SELinux uses BLP model:
    - subjects can
        - read-only less secret objects
        - write-only more secret objects
        - read/write equally secret objects

5.3 LSM

- Linux Security Modules
    - framework for the development of access control enforcement modules
- some cool things that use LSM (now):
    - SELinux
    - Domain Type Enforcement (DTE)
    - POSIX.1e capabilities

LSM Features.

- lightweight
- general-purpose
  - enables creation of many access control models as loadable kernel modules

LSM Design Goals.

- truly generic
  - using a different model = loading a different module
- simple, minimally invasive, efficient
- support existing POSIX.1e capabilities

What LSM Allows Modules to Do.

- defines rules describing subject access to kernel objects
  - does subject `S` have permission to perform operation `O` on object `OBJ`
- purposely only implements the necessary access control functionality required by existing security projects

Implementation of LSM Patch.

- adds security fields to kernel data structures
- insert calls to security hooks at various points in kernel code
- adds generic security system call
- provides functions for modules to register and unregister themselves as security modules

Interaction with LSM Modules.

- modules can use virtual filesystems
  - procfs
  - custom filesystem
- or they can use the `security` system call
  - this system call is implemented by the module
  - not implemented by the kernel by default
  - uses the same signature as the `socketcall` system call

POSIX.1e Capabilities.

- permissive access control
  - grants permissions to processes, rather than taking permissions away
- allow select processes to execute a subset of root capabilities
- this was moved into an LSM module

6 Sandboxing

6.1 Why Sandbox?

- traditional UNIX systems not great at offering protection
- computers are more connected than ever before
  - internet
- distributed/cloud computing services -> increased need to isolate user processes from each other

Early Efforts.

- virtual memory
- capabilities
- DAC
- UNIX implemented these as UGO model augmented by ACLs
  - this was okay but not really adequate anymore
  - certainly not good enough for sandboxing

Primary Goal of Sandboxing.

- protect users from their own applications
  - when these applications are exposed to untrusted content
- complexity of applications increases risk
  - especially true on internet
- sandboxing policy limits impact of compromised processes

6.2 DAC/MAC Sandboxing

OpenSSH (DAC Sandboxing in Practice).

- pre-auth sandbox
  - needed root privileges to bind to TCP port 22
  - compromising before authentication should not grant access to entire system
- post-auth sandbox
  - server should only be able to access resources of the authenticated user
- solution? trusted monitor process and untrusted child process
  - monitor keeps superuser privileges
  - `chroot` jailing child process in pre-auth + nobody user
  - UID/GID changed to actual user in post-auth

Other Applications (DAC).

- technique was effective for OpenBSD due to
  - user-oriented policy (aligns well with UNIX DAC model)
  - extant privilege (the initial application required root privileges anyway)

- problems for general sandboxing
  - doesn't necessarily align with UNIX DAC
  - application does not necessarily have root to begin with
    - e.g., need to ship Chromium with setuid root binaries in order to invoke `chroot`

MAC Sandboxing.

- solutions often way too complex
- complexity hints that we are taking the wrong approach

6.3 System Call Interposition

- idea was to simply hook into system calls and insert special policies within the reference monitor, based on calling application
- example for `chroot`
  - instead of calling `chroot`, hook `open` calls, providing a whitelist of allowed files/directories
- this approach was shown to be ineffective due to concurrent access
  - threads all have concurrent access to objects
  - includes arguments passed to system calls
  - TOCTOU race results
- other concurrency issues?
  - at a higher level, system call wrapper meanings can change
  - file paths can point to different inodes
    - no guarantee that file path at time of policy decision is the same as file path during system call

6.4 Linux `seccomp(2)`

- process makes `seccomp` system call, goes into secure computing mode
- originally, four system calls allowed
  - `read`, `write` on files already opened before seccomp call
  - `sigreturn` for signal handler support
  - `exit` to quit
- to allow more complicated interactions, need support for a whitelist
  - `seccomp-bpf`

`seccomp-bpf`.

- use BPF syntax to define `seccomp` filters
- big problem?
  - whitelisting policies are error-prone
  - example:
    - deny `open(2)` but allow `openat(2)`
    - but `openat(2)` can be made to behave like `open(2)`

- also allows definition of `seccomp(2)` filters on syscall arguments
  - but we have already shown system call arguments are effectively meaningless for policy definitions

Summary of Findings.

- `seccomp-bpf` is insufficient for sandboxing
- for complete solution, need to cut off from system's
  - IPC namespace (`clone(2)`)
  - network namespace
  - mount namespace (similar to `chroot(2)`)
  - PID namespace
- creating new namespaces requires `CAP_SYS_ADMIN` -> essentially superuser privilege
- same problem as we discussed with DAC sandboxing techniques

6.5 OpenBSD `pledge(2)`

- similar to `seccomp(2)` but easier to use
- instead of using BPF program to filter system calls...
  - pledge groups system calls into groups (`stdio`, `rcpath`, etc.)
    - you choose what categories your process can do with a string
  - pledge with empty string only allows `_exit(2)`
    - this causes problems with calls like `mprotect(2)`
- however, a downside is that it creates inconsistent and meaningless policies
  - pledging wpath, but wpath can write to any file on the system
- originally had functionality for whitelisting specific paths in the filesystem
  - this was removed
  - if it was still there, we would have same TOCTOU problem as `seccomp-bpf`
- **pledge's main weakness**
  - if `exec` category is allowed, we can escape/disable the security mechanism
- pledge is also insufficient for sandboxing

6.6 FreeBSD `capsicum(4)`

- capsicum focuses on **access to global namespace**
  - as opposed to previous 2, capsicum doesn't focus on restricting specific system calls
- capsicum puts processes into capability mode
  - this means no access to new resources by global namespaces
- system call to start `capsicum` on a process is `cap_enter(2)`
- processes in capability mode can only open files relative to directories they have permission to access
  - `CAP_READ`
  - `CAP_WRITE`
  - `CAP_LOOKUP`

- inherit open descriptors from parents
- new descriptors can be derived from existing ones
- very minimal effort to use capsicum

7 Mobile OS Security

7.1 App Isntallation Security, Barrera et al.

7.1.1 Android Preliminaries

- App Packages (.apk files)
  - an archive of a Dalvik file that runs on Dalvik VM
  - also has AndroidManifest.xml that contains meta info. for the app
  - above components are signed with the developer's key, included in apk files
- Android Security Model
  - one or more apps are contained in a sandbox (preventing them from writing to other memory)
  - developer can ask for permission outside of sandbox such as camera, GPS sensors by declaring in manifest file
  - manifest is read at installation time
  - codesigning is used with app sandboxing to provide fundamental security aspects to Android

7.1.2 Deconstructing App Installation

- lack of control in app distribution is dangerous(users can install apps from other sources not just play store)
- figure 1 from the reading explains the process of any app being installed pretty well, I will not explain each step in this section

7.1.3 Empirical Dataset

- The research contains datasets from:
  - official and alternative app markets(Play Store, Amazon appstore, Aproov)
  - file sharing networks: Bittorrent
  - malware: Infected apps from researchers

7.1.4 App Update Integrity

- App signing only allows the developer to push update to the app

7.1.5 Signing Details

- Signing is handled by jarsigner
  - Creates RSA certificate: META-INF/NAME.RSA

    - Creates META-INF/MANIFEST.MF(manifest of every files in the app package, not the same as AndroidManifest.xml)
    - Creates a signature file: META-INF/NAME.SF
        - The file may be hashed and within the file, each entry is also hashed
    - The NAME.SF is hashed and signed
    - Appends NAME.SF to NAME.RSA
- When an app is installed:
    - The OS checks the signature from NAME.SF with the public keys in the RSA file
    - It then check the correctness of the hashes between NAME.SF and MANIFEST.MF against the files in the package
- Authentication model
    - trust-on-first-use: once an app is installed from a legit developer then developer can push updates any time. Identity of app developers are not authenticated
- Signature Stripping
    - Allows attackers to hijack apps by changing the signature

7.1.6 Alternative Signing Key Management

- Full Authentication
    - Uses PKI(public key infrastructure) so devs needs to prove their identity to a CA
    - This is a model with bad tradeoff. It would require Android to decide on a list of trustworthy CAs and requires devs to get a CA
- Certificate Tree
    - A long term self-signing key is located at the root
    - Has many benefits such as transfer of ownership
    - Update process will still use the leaf certificates to decide if an update is allowed
    - Reduce replicate copies of keys
- Certificate Expiration
    - Although Google asks for validity period, it actually doesn't enforce certificate expiration during app installation
- Signature Key Updates
    - Developer can use existing key to sign their new certificate
- Revocation of Signing Keys
    - The only way for this to work is to remove the app from the marketplace store
- Distributed & Threshold Signing
    - n people sign the app, and the same n people are required to sign an update
    - this is supported in Android

7.1.7 Publicly Available Key Pairs

- Apps are using a publicly available test keys. Usually sign for third-party Android builds
    - Which means anyone else can issue an update for the app with the same key
    - Android needs to disable installing apps with publicly distributed keys

7.1.8 UID Assignment

- each app have a UID
  - apps can share UID via sharedUserId in AndroidManifest.xml[Requires same signed key]
  - sharing UID is beneficial because APPS can access shared resources(fonts,images,sound)
  - also allows app to use permissiosn from shared apps, which is bad

7.1.9 Properties for UID Sharing

- Groups are Disjoint
  - processes run under a single UID, no way for an app to be running under 2 or more groups
- Groups are Consistent
  - if app A is in the same group as B, and app A is in the same group as C, then B and C must be in the same group
- Group Size is Arbitrary
  - it is possible to specify UID-sharing groups of size 1, 2 or greater than 2
- Membership is Authorized
  - an app cannot join a group without being authorized to join
- Adding New Members is Efficient
  - if a new app is added to the group, then 0,1 or all members need to be updated
- Different Groups with Same Key
  - apps with same keys can belong to different groups

7.1.10 Alternative Mechanisms for UID Sharing

- Mutual Approval
  - each app indicates any other app sharing same UID in manifest
  - difficult to add new memebers because all app needs to be updated
  - prone to group inconsistencies: A includes B and C, but B and C don't include each other
- Pairs
  - groups can only have 2 apps max, since most groups only have 2 apps anyways
  - satisfy group inconsistencies
- Parent-child
  - one parent app authorizes for eevry child app
  - parent app needs to be updated every time a child is added

7.1.11 Permission Assignment

7.1.12 Inheritance Through UID Sharing

- allows app with same UID to have same permissions (LUL bad)
- can mitigate this issue by having the OS installer asking the user to change permissions per app

7.1.13 Signature Permissions

- IPC is slow, only used to transfer small data, which is why UID sharing is used
- Parent-child method can also be used for permission-based IPC calls

7.2 Android Security Documentation

7.2.1 Application Sandbox

7.2.2 Authentication

7.2.3 Encryption

7.2.4 Hardware-Backed Keystore

- dedicated chip (SoC -> system on a chip) for cryptographic operations and keystore

Before Android 6.0.

- simple hardware-backed crypto API
- operations?
  - digital sign, verify
  - generate, import signing key pairs
- this functionality is okay, but misses several security goals

Since Android 6.0.

- symmetric crypto primitives
- AES, HMAC
- access control system for hardware-backed keys
- usage control scheme for limiting key usage (ephemeral keys)

Since Android 7.0.

- key attestation
  - public key certificates describing key and access controls
  - provides remote verifiability for keys
- version binding
  - bind keys to OS and patch version
  - ensures attackers cannot roll device back to vulnerable version and use existing keys
  - when upgrading to new version, keys are upgraded before they can be used

Since Android 8.0.

- no longer express hardware abstraction layer (HAL) in C-style structs
  - now, use higher level HIDL (hardware interface definition language) to generate C++
- ID attestation

    - optionally strongly identify hardware identifiers
    - e.g. device name, device serial number, phone ID, etc.

Since Android 9.0.

- embedded secure elements
- secure key import
- 3DES encryption
- separately set version binding for `boot.img` and `system.img` (independent updates)

Glossary of Android Keystore Terms.

- `AndroidKeystore`
    - API used by apps to access keystore functionality
    - forwards requests to keystore daemon
- `keystore daemon`
    - system daemon that provides access to all keystore functionality
    - stores encrypted key blobs
- `keymasterd`
    - provides access to Keymaster TA (trusted application)
- `Keymaster TA`
    - runs in a secure TCE (like an SoC)
    - provides all keystore operations
    - has access to raw keying material
- `LockSettingsService`
    - responsible for user authentication
    - not part of the keystore, but many keystore services require user authentication
    - interacts with Gatekeeper TA and Fingerprint TA
- `Gatekeeper TA`
    - authenticates user passwords
    - generates authentication tokens
        - (proves to Keymaster TA that authentication was done for a particular user at a particular point in time)
- `Fingerprint TA`
    - like Gatekeeper, but for fingerprints

Architecture.

- client makes a shared library call to Keymaster HAL (just a shared library provided by OEM)
    - this keymaster HAL is no accessed directly by applications, used by platform-internal components
- shared library accesses kernel interface
- kernel interface accesses keymaster in TCE hardware

7.2.5 SELinux

- Android can use SELinux for MAC
  - augments traditional DAC, just like we have seen
- for Android 4.3 and higher

Background.

- Android security model is based on sandboxing
- each application runs in its own sandbox
- Android 4.3 -> everything permissive, Android 4.4 -> partial enforcement
- after Android 5.0 -> full enforcement
  - over 60 domains
- Android 6.0 -> further reduced permissiveness
  - better isolation between users
  - ioctl filtering
  - tightening domains
  - limited access to `/proc`
- Android 7.0 -> even further lockdown
  - break up mediaserver stack into smaller processes, reduced permission scope
- Android 8.0 -> update SELinux to work with Treble
  - separate lower level vendor code from Android system framework
  - compatibility issues if platform version is higher than vendor version?
    - no, we have a method to retain compatibility

7.2.6 Trusty TEE

- separate OS from Android
- runs on same processor, but isolated from the rest of the system
  - protected from malicious apps
  - protected from Android vulnerabilities
- provides a trusted execution environment for Android TCB

Architecture.

- kernel is derived from Little Kernel (not Linux)
- driver in Linux kernel to transfer data back and forth
- userspace library for Android to communicate with trusted applications via kernel driver

7.2.7 Verified Boot

7.3 Behind the Scenes of iOS Security

- got rid of encryption on kernel caching in iOS 10
  - turns out they didn't need it

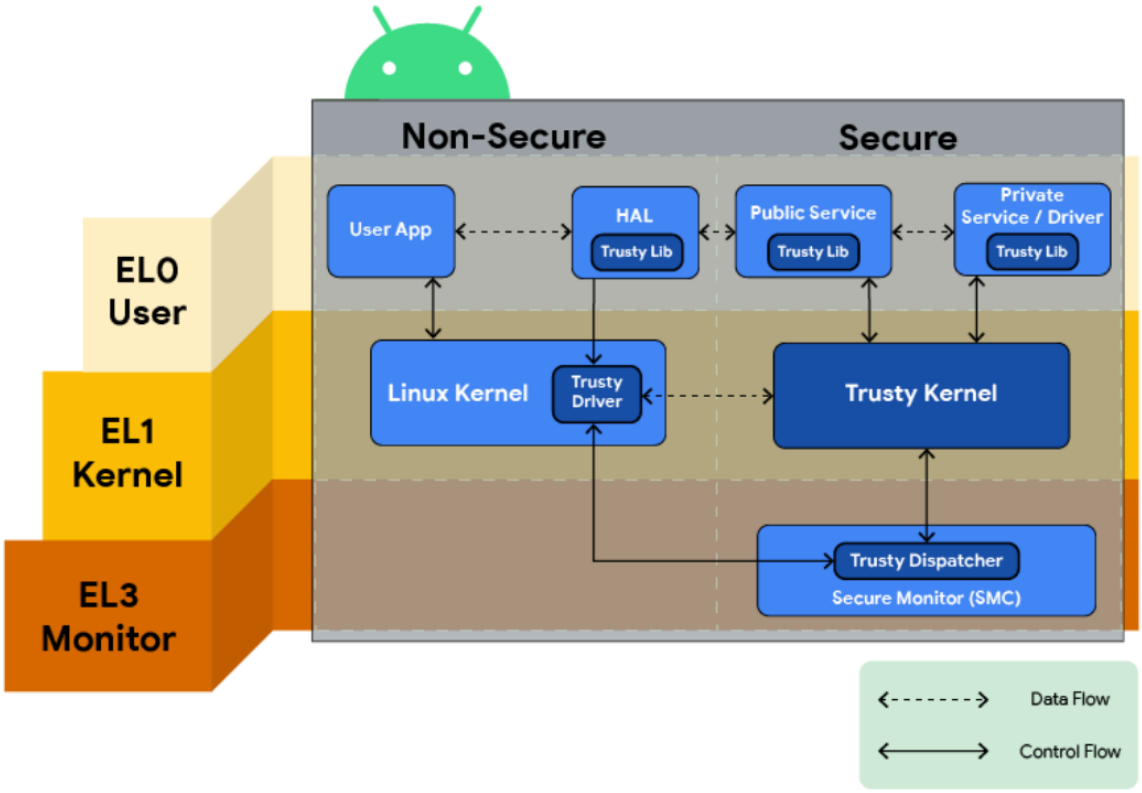

Figure 7.1: Design of trusty, from Android docs.

    - question for David: is this related to the recent unpatchable jailbreak? (note video is from 2016, jailbreak came out in 2019)

7.3.1 Hardened WebKit JIT Mapping

- JIT compilation for javascript in Safari
    - boost performance
    - but with JIT, we cannot enforce codesigning
    - (we don't necessarily know what the compilation will produce beforehand, so we cannot sign it)
- make a concession:
    - bring JIT to Safari, and allow JIT to emit unsigned code

How Things Worked in iOS 9.

- 32MB RWX JIT memory region
    - whoops this is actually terrible
    - attacker writes to JIT memory region -> code execution

Execute-Only Memory Protection.

- hardware support added for this in ARMv8 processor
- added kernel support in iOS10
- processes can now execute code *without reading it*

The Solution.

- what can we do with this?
    - create two virtual mappings to same physical JIT memory
    - one is writable and not executable
    - one is executable-only
    - location of writable mapping is kept secret
    - create special version of `memcpy` that keeps track of this secret address

Consequences for Attackers.

- harder to exploit memory corruption in WebKit
- now need to rely on return oriented programming or something similar

7.3.2 Secure Enclave Processor for Data Protection

- protect user data at rest
- goals for this?
    - strong crypto keys derived from user passcode
    - no offline attacks on passcode
    - no brute force on passcode (rate limiting, hard limit on number of attempts)
    - hardware keys for key derivation are not exposed to mutable software

- secure support for alternative authentication (touch ID, auto unlock, face ID, etc.)

Additional Goals.

- sidestep application processor attack surface
  - enforce authentication policy even under adversarial AP
  - master key never exposed to AP
  - non-master key material exposed to AP is ephemeral

Secure Enclave Processor.

- dedicated core provides trusted environment for handling crypto material
- arbitrates all user data access
- hardware accelerators for AES, EC, SHA
- manager own encrypted memory
- communicates with AP via mailboxes
- secure channels to touch ID sensor, secure element (for Apple Pay)

Device Unique Key.

- each SEP has access to a unique private key (called UID here)
- generated by SEP using its own TRNG (True RNG)
- available for cryptographic operations via commands exposed by read-only memory
- SEP has no access to key material for its private key after fuses blown
  - fuses are blown after first power on

User Keybags.

- sets of keys generated for each user
- wrapped around master key derived from user passcode and SEP UID
- after 10 incorrect passcode entries, SEP does not process further attempts
- different policy associated with each keybag

Class	Availability
A	pub, priv $\rightarrow$ only when device is unlocked
B	pub $\rightarrow$ always, priv $\rightarrow$ only when device is unlocked
C	pub, priv $\rightarrow$ available after the user unlocked their phone at least once
D	pub, priv $\rightarrow$ always available

Deriving Keys.

- userland
  - passcode and salt given to KDF
  - derive first key, pass over to hardware (SEP)
- SEP
  - derive further keys using $\text{MK}_i = \text{KDF}_2(\text{E}(\text{UID}, \text{MK}_{i-1}))$
  - timed iterations 100-150 ms

Protecting Data on Filesystem.

- encrypt blocks with AES-XTS
- encrypt each file on user partition with unique random key
- raw file keys are not exposed to AP
  - wrapped with user keybag key for long term storage
  - wrapped with ephemeral key bound to boot session while in use
- keys shared from SEP to sotrage controller via secure channel

First Unlock.

- SpringBoard sends passcode to SEP
- SEP generates the master key from passcode + UID hardware key
- use master key to decrypt all class keys and add them to the key ring
- generate random secret and encrypt master key with that secret
  - sent random secret only to SBIO (biometric authentication in the SEP)
  - destroy raw master key and hold onto encrypted version

Device Lock.

- notify SEP, SEP purges keys from keyring according to class

Touch ID Unlock.

- authenticate, send random secret back to SKS
- SKS then decrypts the master key and uses that master key to decrypt class keys
- add them once again to keyring
- securely destroy master key again

Update Later.

- if user chooses to update their phone later, ask for password now and enable creation of a token
- predict using AI the last time the user will unlock phone before the window
- create one time token at that time
- use this token to update software

SoC Security Mode.

- **demotion**
  - devices can be demoted to enable to debugging features
  - requires full OS erase + device is authorized
- this demotion forces a different UID on the SEP
- this means no access to existing user data (because wrong master key)

7.3.3 Synchronizing Secrets

- help the Apple ecosystem share keys with each other

- e.g. auto phone unlock from iMac or Apple Watch
- e.g. HomeKit storing cryptographic keys

Two Traditional Approaches.

1. strong random key that encrypts all others
   - user has to take care of a key and use it for all devices
   - losing printed key means loss of secrets
2. wrap user secrets with KDF-derived key from password
   - intercepting or brute forcing account password
   - resetting account password, must prompt for old password
   - guessing attacks

iCloud Keychain.

- each device generates local key pair
- explicitly approve new devices from a device already in the circle
- Apple cloud backend for storage and message passing
  - no data accessible to Apple
  - backend is not in a privileged position

Backend Attack Surface.

- limit attack surface
- no brute force
- solution? key unwrapping only takes place in hardware security modules

Cloud Key Vault.

- HSMs running custom secure code
- key vault fleet operates its own CA (private key never leaves hardware)
- each iOS device hardcodes key vault fleet CA cert

HSM Keys.

- see following table

Key	Description
CSCIK	custom secure code signing key, signs custom secure code to run on HSM
AK	authenticate messages between vault units
CA	certify SK
SK	unwrap escrow records

Cloud Key Vault Design.

- kind of like SEP data protection for escrow records
- private key material not available to mutable software

- group units into "clubs"
  - provides redundancy, but does not solve brute force
  - vulnerable to partitioning attacks to defeat rate limiting
- to solve brute force, we require majority votes from all club members to proceed

Protecting CSCIK.

- three physical cards
- place in evidence bags
- send to three departments
- all three are needed to decrypt CSCIK
- destroy cards after use