# COMP5900T Midterm Study Guide

## Topics, Readings, and Activities

by

**William Findlay**

March 2, 2020

# Contents

# List of Figures

# List of Tables

# List of Listings

# 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 was a problem
- why do we use these? absract 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

## 2.2 Reference Monitor

**Tamperproofing.**

**Complete Mediation.**

**Verifiability.**

## 2.3 Access Control Fundamentals

## 2.4 Activity: Design a Reference Monitor for a Real Life Situation

# 3 Multics

## 3.1 Activity: Using Multics

## 3.2 Activity: Multics Security Evaluation

# 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

## 5.2 SELinux

## 5.3 Activity: Compare Custom SELinux Policy with Real SELinux Policy

# 6 UNIX Sandboxing

## 6.1 Linux `seccomp(2)` and `seccomp-bpf(2)`

## 6.2 OpenBSD `pledge(2)`

## 6.3 FreeBSD `capsicum(4)`

## 6.4 Acitivity: Compare the Three Alternatives

# 7 Mobile OS Security

## 7.1 Android

## 7.2 iOS

## 7.3 Activity: Comparing Android, iOS, and Desktop