ECE455: Week 4 readings notes

Jonathan Lam

09/30/21

1 Security kernel design and implementation: an introduction

Ames et. al (1983)

- "Security kernel technology provides a conceptual base on which to build secure computer systems, thereby replacing this game of wits with a methodoical design process"
- Security kernel is good because:
  - Addresses size and complexity problem by limiting the protection mechanism to a small portion of the system
  - Clearly defines a security policy
  - Following a rigorous metodology that includes developing a mathematical model, constructing a precise specification of behavior, and coding in a high-level language
- **Reference monitor**: each access to information or change of authorization must go through the reference monitor
- Three major engineering requirements:
  - **Completeness**: cannot bypass security monitor
  - **Isolation**: kernel is tamper-resistant
  - **Verifiability**: mathematical evidence that it does what it is intended to
  - (Achieve first two through hardware)
- Important to clearly define security – system is only secure w.r.t. a policy; other attempts have been much more lax and pathwork

  - Model should not be created w.r.t. individual assertions, but rather the entire system as a whole
  - **Bell and Lapadula** model – FSM, each subject/object is given an *access class*, lattice to give more possibilities.
    * Two important non-discretionary rules:
      · **Simple security condition**: subject cannot observe contents of an object unless the access class of the subject is greater than or equal to object
      · **Star property**: subject may not modify an object less unless object's access class is greater than or equal to access class of the subject – meant to avoid Trojan horse software
      · These don't protect against/distinguish between different users within the same access class – this requires discretionary rules (which are of lower precedence)

- Three classes of formal verification techniques:
  - **Security flow analysis**: prove that formal high-level interface specification is secure w.r.t. policy model; does not check for correctness of functionality, only security of API
  - Second class: "verify the correspondence of mappings between any intermediate specifications in the hierarchy and the interface specification"
  - Third class: "traditional" correctness proving

1.1 Implementation considerations

- Kernel/supervisor trade-offs: kernel must manage resources; may find it difficult to unentangle code
- **Trusted subjects**: a set of interfaces with extended privileges, e.g., for maintenance purposes
  - Usually implemented as trusted processes or trusted functions
- Hardware/software features: four general hardware areas required:
  - Explicit processes: processes are the subject, thus we need these identifiers to be correct; hardware support for context switching is necessary for this to be efficient; also desire a race-free IPC mechanism

- Memory protection: all memory access needs to have a descriptor
  - "With a reference monitor, all information within the system must be represeted in distinct, identifiable objects"
  - **Segment** (as in segmentation fault): logically-distinct memory, can have multiple under the same process, has a defined access level
  - Each process should be able to have a relatively large number of independent segments, and any segment should have a wide range of possible sizes
  - System performance can be improved by including a referenced and motified flag for each block of physical memory (caching?)
  - Security kernel maintains segment descriptors, but the hardware (MMU?) manages the actual permissions
  - **Control information**: metadata may be leaked, should be a security concern from the beginning
- Execution domains: usually two or three execution domains: user-space, (supervisor), and kernel; mostly hierarchical; want efficient calls between levels
- I/O mediation: two major ways to handle I/O:
  - Programmed I/O: manually perform each small I/O operation
  - Independent I/O controllers and processes: I/O controller accesses data on its own; manage a descriptor to the I/O directly rather than to the memory accessed by it (**this part was a little confusing**)
  - For removable/external I/O media, need a "trusted labeling technique to ensure that the access classs of the medium is correctly marked"

2 Engineering a security kernel for MULTICS

MIT 1975 (?)

- MULTICS: general-purpose, remotely-accessed, multiuser computer system
- Want simple and efficient system that can be "verified by auditing"

- Want to implement "information release constraints of the military security system"
- "System is secure if it is known to prevent all actions defined as unauthorized by the specification of its security properties"
- Security cannot be proven by testing, only by a logical verification
- One method for secure systems is a top-down approach via formal specification
- **Security kernel** is a minimal, protected core of software whose correct operation is sufficient to guarantee enforcement of the claimed constraints of access
  - "A security kernel should be the least amount of common mechanism necessary to implement the patterns of information sharing, interprocess communication, and physical resource multiplexing that are desired in the system"
- **Common mechanisms** are any mechanism that allows one computation to influence (communicate with) another
- Four categories of non-kernel software; these depend on the security kernel and can will be protected against actions that break the specifications defined by the security kernel (but can still be a security concern if the user-space code on top of these is incorrect and/or malicious):
  - System-provided software
  - Programs constructed by the user
  - Programs borrowed from other users
  - Common mechanisms that a group of users sets up to implement some function involving interuser communication or coordination.
- A high-level language and compiler may be useful for verifying the security kernel; while this adds additional complexity and requires the compiler to be correct, it is easier for the compiler to perform verification than to do this manually, and we can perform **translation verification**

2.1 Goal 1: Reduce size of kernel by moving things out of kernel ring

- Original MULTICS system had protection rings implemented in software
  - Cross-ring calls were expensive and thus occasionally extra functionality was implemented in lower layers; we undo this to keep the size of the kernel small
  - "Current" MULTICS system is Honeywell Level 68, which has hardware protection for rings, where there is no performance hit
  - Part of this moving between performance layers is difficult because code is entwined
- Activities to reduce size of kernel:
  - "Dynamic intersegment linking and direction of file system searches to satisfy symbolic references"
  - "Facilities for managing the association between reference names and the segments in the address space of a process"
  - System initialization from userspace
  - Combining user login authorization and process authorization

2.2 Goal 2: Restructure mechanisms that must remain in the kernel

- Examples:
  - Generalizing I/O to standardized model
  - Using the general vm architecture rather than a special-purpose circular buffer for network input
  - Separating the implementation of processes into a two-level scheme: one to manage vm, the other for the use of it
    - This simplifies things by making actions sequential (in the context of a virtual processor) – the parallelization/asynchronous behavior is handled by the process implementation
    - This could also be used to implement interrupts, giving them a dedicated virtual processor

2.3 Goal 3: Partition/modularize the kernel

- Divide the kernel into domains arranged so that each property is implied by a subset of the domains
- Two specific methods:
  - "Dividing part of kernel that is in address space of each process into multiple layers in diffirent rings of protection"
  - "Placing some of the kernel processes in separate addresses spaces and also using the protection rings to layers them"
- General idea of **separating policy from mechanism**

3 Bypassing non-executable-stack during exploitation using return-to-libc

c0ntex (year?)

- Assume non-executable stack, standard buffer overflow

4 Smashing the stack for fun and profit

Aleph One (year?)

- **Shellcode**: bytecode of executable code to be injected
- TODO: take more notes on this

5 Questions

5.1 MULTICS

- This paper never gets into the details of formal verification – all about how to reduce the size of a system

5.2 Reference monitor

- Lattice is confusing – why inequalities go opposite ways for two non-discretionary rules

5.3 return-to-libc

- When is stack executable?
- Why overwrite ebp with return address? Shouldn't the eip be overwritten?