Today's plan

- distributed computing, continued
- security overview
- multics security
- file system review

Architectures for Distributed Computing

- Commercial cloud provider:
  - data center with cheap electricity and/or cooling
  - many compute nodes connected by high-bandwidth, low-latency network
  - 24/7 technical support – nodes will fail, have a strategy to deal with that
- homebrew architectures:
  - closely coupled processors with a single shared memory (but separate caches) and devices shared among all CPUs: **MP** (multiprocessor) often now handled by traditional operating systems (Linux, Windows)
  - Network of Workstations (NoW), general-purpose workstations that cooperate to get work done
  - Beowulf cluster, similar to an NoW but processors not intended for general-purpose use (maybe no graphics card, maybe no local disk), interconnected for high performance, and often in larger numbers than a typical NoW
  - grid computing: workstations, scattered across an internet, that agree to perform requested work while idle

Security Overview

- a *policy* determines who has access to what **data** and **resources**
- a *mechanism* allows the specification of a policy and implements the policy
  - e.g. file ownership and permissions are a mechanism, assignment of specific permissions (and/or ownership) to a file are policy
- an *audit trail* keeps track of who reads or modifies what data or code. It is best if audit trails are **write-once** or **append-only**
- users execute programs, but it is sometimes hard for a user to know what a program will do (even if the user has authored the program!), so programs should not trusted more than necessary

Multics Overview

- Multics was an early time-sharing system
  - and Multics inspired Unix
- designed for interactive use
- much of Multics was programmed in a high-level language (PL/1)
- potentially multiprocessor

Multics Security

- Multics designers didn't want to repeat the insecurities of previous time-sharing systems
- hardware segmentation (with paging), with each segment having access protection bits
- segments are persistent and are stored in directories which have Access Control Lists (ACLs)
- hardware rings -- 8 concentric rings of protection, implemented as a separate descriptor table for each ring
- rings were designed to be implemented in hardware, but were actually implemented by having "master mode" be ring 0 and providing facilities to cross from one ring to another. When a program tries to add a segment to its space, ring 0 checks the ACL
  - similar to the supervisory bit in modern processors
- process table entries contain a user ID and are only accessible from ring 0
- passwords are stored encrypted, but the passwords file is also inaccessible to users
- users are told when they last tried to log in
- all system code is reloaded from tape when rebooting

Multics attack

- classic paper, e.g. at http://csrc.nist.gov/publications/history/karg74.pdf by Karger and Schell
- conclusion: not secure enough to have untrusted users share the computer with classified documents
- attacks designed to break hardware, software, and procedures, to the point of being able to access or modify "protected" data
- attackers had access to software and to hardware design (no "security by obscurity")
- attacks designed on a computer on an Air Force Base, then reproduced on a similar computer at MIT
- attacks should be undetectable, e.g. not cause anomalous crashes
- procedural vulnerabilities included physical access, the ability to use the software attacks to masquerade as a different user or change arbitrary memory or files, modifying the software at the development facility or enroute
- overall result: securing a system takes much more work than breaking in to a system!
- suggestion: a small (understandable) security kernel might do the trick

Multics Attack Specifics

- random tests of illegal operations or parameters to try and find unexpected behavior
- random tests uncovered a hardware bug that allowed some accesses to go unchecked (the bug had been introduced after the original design by an "upgrade")
- gate 0 code uses a routine to determine if argument lists are valid, but an autoincrement feature could lead the check to succeed but the actually accessed word to not be permitted
- a procedure (the "signaller", to deliver signals to user processes) was for efficiency placed in user space ("user ring") but ran in master mode, allowing a careful user to transfer to arbitrary code while still in master mode
- a change in usage of the stack segment register allowed user programs to store the registers in an arbitrary segment on a context switch, e.g. allowing the installation of a trap door into user code
- trap doors include modifying the TTY driver to recognize a special string, and modifying the compiler to recognize itself (and produce a trapdoor-generating compiler) and a given piece of code

Multics 30 years later

- new paper, e.g. at http://www.acsac.org/2002/papers/classic-multics.pdf, also by Karger and Schell
- Multics had security as its goal from the start, other systems don't (and therefore are weaker)
- modifying other systems to be more secure may interfere with "normal" usage, so the code base forks
- Multics did not suffer from buffer overflows, since PL/1 is safer than C and the segments prevented memory misuse
- Multics stacks grow upwards, so an overflow does not overwrite the return address
- Multics is simpler than most current systems, e.g. Multics ring 0 is much smaller than the SELinux security module
- intrusion detection often does not deal with the possibility of professional attackers who refine the attack on their system before attacking the target
- the GEMSOS OS certified to "Class A1" security standards, so the technology exists
- why aren't we doing better?
- chicken and egg problem: makers don't think customers want it, customers don't know makers can do it

example file systems: NTFS

- a Master File Table (MFT) contains a set of records, each corresponding to one file
- MFT is similar to a combination superblock and inode table
- one of the early files in the MFT is a mirror of the MFT
- another of the early files is the bitmap of blocks used
- MFT records have a record header, followed by (attribute-header, attribute-value) pairs
- attributes include various inode-type information such as size, ownership, etc
- one attributes is the data of the file
- attribute values can be stored immediately within the MFT, if small, or be nonresident
- nonresident attributes are stored in a sequence of records, each of which is a sequence of logically consecutive blocks (allowing files to have holes)
- each record is a sequence of (block-number, number-of-blocks), which is the same as an extent, or segment on disk
- if the information for a file needs more than one MFT record, additional MFT records are used, and are recorded in the first MFT record

example file systems: 1980s Unix

- older unix file systems used fixed-size inodes (128 bytes) to hold data for any kind of file, including directories
- the inodes had room for up to a dozen pointers, of which the first few referred to data blocks, the next one to an indirect block, and the final two to a double and triple indirect block
- file system consistency requires every inode's reference count to match the number of its incoming links, the bitmap to correctly record used and free blocks, and every directory entry to refer to a valid inode

Security in 2019

- Systems are routinely tested in real life
  - powerful actors have incentives to break systems
  - financial rewards for successful break-ins
    - both legitimate and not
- buffer overflow is still a problem (heartbleed)
- new side-channel attacks
  - e.g. meltdown and spectre (but no known exploits)
- users are often the weakest link (still!)

Storage Management

- storage is an addressable collection of bytes
- sometimes the bytes are grouped into blocks, e.g. 512-byte pages
- at the very least, storage management must keep track of the free space
- there is some overhead to keeping track of space, so free space is usually managed in units of blocks (fixed size) or segments (variable size)
- there are two basic techniques for keeping track of free space:
  - **free list**: each free block (segment) has a reference to the next free block (segment)
  - **bitmap**: each block in the system corresponds to a bit in the bitmap. The bit is zero if the block is allocated, and one if it is free
- the bitmap has an overhead of 1 bit per block in the storage system
- bitmaps only really work with fixed-size blocks, not segments

Free List Management

- freeing a block or segment adds it to the front of the free list -- O(1)
- allocating a block or segment will:
  - take the first block from the free list -- O(1)
  - take the first segment from the free list -- O(1), but may not give sufficient space
  - take the first segment from the free list that is large enough -- *first fit*, O(n), where n is the number of segments in the free list
  - take the smallest segment from the free list that is large enough -- *best fit*, O(n)
  - take the largest segment from the free list -- *worst fit*, O(n)
- O(1)/O(n) matters because each access to a segment requires a disk access
- first-fit is faster than best fit or worst fit, even though all are O(n)
- first-fit may also give a better distribution of segment sizes overall

- free list management makes it hard (O(n)) to give preference to storage near a given location, or to merge adjacent segments

Storage Allocation

- allocating storage means marking the storage as not free:
  - clearing the corresponding bits in the bitmap, or
  - removing the block(s) or segment(s) from the free list, or
  - readjusting the size of a segment in the free list
- freeing/recycling storage means marking the storage as free:
  - setting the corresponding bits in the bitmap, or
  - adding the block(s) or segment(s) to the free list, perhaps merging adjacent segments (which takes O(n) for a straightforward free list)

File Systems

- a file system must do two things:
  - manage free space
  - keep track of named, persistent data
- keeping track of data is similar to keeing track of free space, except:
  - one bitmap per file would be impractical
  - a block or segment list is fine for sequential access, but not enough for random access to the data
  - efficient file systems must keep some sort of index to all the blocks or segments
- keeping track of names can be simply a matter of keeping a table of (name, index-for-data, other-info-if-any) entries in a directory

file indices and inodes

- a file index needs to keep track of all the pointers to blocks or the pointers and sizes of segments
- this requires a number of entries proportional to the size of the file (for blocks) or to the number of segments
- a fixed-size index node is not suitable for this, but can be if it refers to the root of a tree of block references
- the (dis)advantage of an inode is that a file exists independently of whatever directories may link to it:
  - multiple hard links to a file are possible
  - file ownership and protection are independent of links

directory structure

- a superblock contains a reference to the bitmap or free list, and a reference to the root of the directory tree
- each directory has a table of (name, index) pairs, of which both the name and the index may be either fixed or variable sized
- when inserting a new name into a directory, can add into a free position or at the end -- this is similar to storage allocation
- when removing a name from a directory, can move the rest of the directory so there is no gap -- this is slow for big directories
- if the directory entries have a fixed size, can move the last entry into the slot left by the removed entry, and shrink the size of the directory file

file I/O

- in a conventional file system, all the data structures are on disk
- before a data structure can be used, it must be brought into memory, either as a whole, or in part
- whenever (part of) a data structure is modified, it must be written back to disk
- to make this easy, many data structures (e.g. inodes, directory entries) can be designed to fit into a block, so access is to the data structures within a block
- I/O is also needed to bring file data into memory, and write changed file contents back to disk