Security / VM (start)

last time

two-phase commit: doing operation together

- data is split across several machines
- redo logging — machines know what message to send when rebooting
- state machine to describe protocol — for proving/testing properties
- prepare phase: make promises (can commit/will abort)
- finishing phase: commit if everyone agreed; otherwise abort

quorum consensus: continuing despite failures

- everyone has a copy of shared data
- require quorum (e.g. majority) of nodes
- ask for votes for reads *and* writes
- overlap: guarentee one voter knows about about last update
- everyone in quorum always updates to latest version

started security

last time

two-phase commit: doing operation together

- data is split across several machines
- redo logging — machines know what message to send when rebooting
- state machine to describe protocol — for proving/testing properties
- prepare phase: make promises (can commit/will abort)
- finishing phase: commit if everyone agreed; otherwise abort

quorum consensus: continuing despite failures

- everyone has a copy of shared data
- require quorum (e.g. majority) of nodes
- ask for votes for reads *and* writes
- overlap: guarentee one voter knows about about last update
- everyone in quorum always updates to latest version

started security

a note on grading

hope to have FAT grades this week

- probably should have "you must test with/supplied Makefile will use AddressSanitizer" policy in the future to avoid cases where program totally breaks on the dept. servers I use for testing but probably worked where student was running it

hope to go through last half of quiz comments next week

access control matrix: who does what?

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

access control matrix: who does what?

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs to 1+ *protection domains*:
"user cr4bd"
"group csfaculty"
...

access control matrix: who does what?

objects (whatever type) with restrictions

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs to 1+ *protection domains*:

"user cr4bd"

"group csfaculty"

...

representing access

with objects (files, etc.): *access control list*

- list of protection domains (users, groups, processes, etc.) allowed to use each item

list of (domain, object, permissions) stored "on the side"

- example: AppArmor on Linux
- configuration file with list of program + what it is allowed to access
- prevent, e.g., print server from writing files it shouldn't

representing access

with objects (files, etc.): *access control list*

- list of protection domains (users, groups, processes, etc.) allowed to use each item

list of (domain, object, permissions) stored "on the side"

- example: AppArmor on Linux
- configuration file with list of program + what it is allowed to access
- prevent, e.g., print server from writing files it shouldn't

access control list parts

assign processes to protection domains

- typically: process assigned user + group(s)
- object (file, etc.) access based on user/group

attach lists to objects (files, processes, etc.)

- sometimes very restricted form of list
- e.g. can only specify one user + group

user IDs

most common way OSes identify what *domain* process belongs to:

(unspecified for now) procedure sets user IDs

every process has a user ID

user ID used to decide what process is authorized to do

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

standard programs/library maintain number to name mapping

- `/etc/passwd` on typical single-user systems
- network database on department machines

POSIX groups

```
gid_t getegid(void);
    // process's"effective" group ID

int getgroups(int size, gid_t list[]);
    // process's extra group IDs
```

POSIX also has *group IDs*

like user IDs: kernel only knows numbers

- standard library+databases for mapping to names

also process has some other group IDs — we'll talk later

id

```
cr4bd@power4
: /net/zf14/cr4bd/fall2018/cs4414/hw/fat/grading ; id
uid=858182(cr4bd) gid=21(csfaculty)
        groups=21(csfaculty),325(instructors),90027(cs4414)
```

id command displays uid, gid, group list

names looked up in database

- kernel doesn't know about this database
- code in the C standard library

groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

POSIX file permissions

POSIX files have a very restricted access control list

one user ID + read/write/execute bits for user
    "owner" — also can change permissions

one group ID + read/write/execute bits for group

default setting — read/write/execute

(see docs for chmod command)

POSIX/NTFS ACLs

more flexible access control lists

list of (user or group, read or write or execute or ...)

supported by NTFS (Windows)

a version standardized by POSIX, but usually not supported

POSIX ACL syntax

```
# group students have read+execute permissions
group:students:r-x
# group faculty has read/write/execute permissions
group:faculty:rwx
# user mst3k has read/write/execute permissions
user:mst3k:rwx
# user tj1a has no permissions, even if in group above
user:tj1a:---
```

authorization checking on Unix

checked on system call entry

- no relying on libraries, etc. to do checks

files (open, rename, ...) — file/directory permissions

processes (kill, ...) — process UID = user UID

...

superuser

user ID `0` is special

superuser or *root*

some system calls: only work for uid 0

- shutdown, mount new file systems, etc.

automatically passes all (or almost all) permission checks

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

Unix password storage

typical single-user system: `/etc/shadow`

- only readable by root/superuser

department machines: network service

- Kerberos / Active Directory
- server takes (encrypted) passwords, gives out "tokens" saying "yes, it is this user"
- can cryptographically verify tokens come from server

aside: beyond passwords

/bin/login entirely user-space code

only thing special about it: when it's run

could use any criteria to decide, not just passwords

- physical tokens
- biometrics
- ...

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

changing user IDs

```
int setuid(uid_t uid);
```

if superuser: sets effective user ID to arbitrary value

- and a "real user ID" and a "saved set-user-ID" (we'll talk later)

system starts in/login programs run as superuser

- voluntarily restrict own access before running shell, etc.

sudo

```
tj1a@somemachine$ sudo restart
Password: *********
```

sudo: run command with superuser permissions

started by non-superuser

recall: inherits non-superuser UID

can't just call `setuid(0)`

set-user-ID sudo

extra metadata bit on *executables*: set-user-ID

if set: `exec` system call changes effectve user ID to owner of executable

`sudo` program: owned by root, marked set-user-ID

marking setuid: `chmod u+s`

set-user ID gates

set-user ID program: gate to higher privilege

controlled access to extra functionality

make authorization/authentication decisions *outside the kernel*

way to allow normal users to do *one thing that needs privileges*

- write program that does that one thing — nothing else!
- make it owned by user that can do it (e.g. root)
- mark it set-user-ID

want to allow only some user to do the thing

- make program check which user ran it

uses for setuid programs

mount USB stick

- setuid program controls option to kernel mount syscall
- make sure user can't replace sensitive directories
- make sure user can't mess up filesystems on normal hard disks
- make sure user can't mount new setuid root files

control access to device — printer, monitor, etc.

- setuid program talks to device + decides who can

write to secure log file

- setuid program ensures that log is append-only for normal users

bind to a particular port number < 1024

- setuid program creates socket, then becomes not root

set-user-ID program v syscalls

hardware decision: some things only for kernel

system calls: *controlled* access to things kernel can do

decision about how can do it: in the kernel

kernel decision: some things only for root (or other user)

set-user-ID programs: controlled access to things root/... can do

decision about how can do it: made by root/...

set-user ID programs are very hard to write

what if stdin, stdout, stderr start closed?

what if the PATH env. var. set to directory of malicious programs?

what if `argc == 0`?

what if dynamic linker env. vars are set?

what if some bug allows memory corruption?

...

a delegation problem

consider printing program marked setuid to access printer

> decision: no accessing printer directly
> printing program enforces page limits, etc.

command line: file to print

can printing program just call open()?

a broken solution

```
if (original user can read file from argument) {
    open(file from argument);
    read contents of file;
    write contents of file to printer
    close(file from argument);
}
```

hope: this prevents users from printing files than can't read

problem: race condition!

a broken solution / why

setuid program	other user program
	create normal file `toprint.txt`
check: can user access? (yes)	—
	`unlink("toprint.txt")`
	`link("/secret", "toprint.txt")`
`open("toprint.txt")`	—
read ...	—

time-to-check-to-time-of-use vulnerability

TOCTTOU solution

temporarily ‘become’ original user

then open

then turn back into set-uid user

this is why POSIX processes have multiple user IDs

can swap out effective user ID temporarily

practical TOCTTOU races?

can use symlinks *maze* to make check slower

symlink `toprint.txt` → `a/b/c/d/e/f/g/normal.txt`
symlink `a/b` → `../a`
symlink `a/c` → `../a`
...

gives more time to sneak in unlink/link or (more likely) rename

aside: real/effective/saved

POSIX processes have *three* user IDs

effective — determines permission — `geteuid()`

- jo running sudo: geteuid = superuser's ID

real — the user who started the program — `getuid()`

- jo running sudo: getuid = jo's ID

saved set-user-ID — user ID from *before* last exec

- effective user ID saved when a set-user-ID program starts
- jo running sudo: = jo's ID
- no standard get function, but see Linux's `getresuid`

process can swap or set effective UID with real/saved UID

aside: real/effective/saved

POSIX processes have *three* user IDs

effective — determines permission — `geteuid()`

- jo running sudo: geteuid = superuser's ID

real — the user who started the program — `getuid()`

- jo running sudo: getuid = jo's ID

saved set-user-ID — user ID from *before* last exec

- effective user ID saved when a set-user-ID program starts
- jo running sudo: = jo's ID
- no standard get function, but see Linux's `getresuid`

process can swap or set effective UID with real/saved UID

- idea: become other user for one operation, then switch back

why so many?

two versions of Unix:

System V — used effective user ID + saved set-user-ID

BSD — used effective user ID + real user ID

POSIX commitee solution: keep both

aside: confusing setuid functions

setuid — if root, change all uids; otherwise, only effective uid

seteuid — change effective uid

- if not root, only to real or saved-set-user ID

setreuid — change real+effective; sometimes saved, too

- if not root, only to real or effective or saved-set-user ID

...

more info: Chen et al, "Setuid Demystified"

`https://www.usenix.org/conference/11th-usenix-security-symposium/setuid-demystified`

also group-IDs

processes also have a real/effective/saved-set group-ID

can also have set-group-ID executables

same as set-user-ID, but only changes groupo

ambient authority

POSIX permissions based on user/group IDs process has

- correct user/group ID — can read file
- correct user ID — can kill process

permission information "on the side"

- separate from how to identify file/process

sometimes called *ambient authority*

"there's authorizationin the air..."

alternate approach: ability to address = permission to access

representing access

with objects (files, etc.): *access control list*

- list of protection domains (users, groups, processes, etc.) allowed to use each item

list of (domain, object, permissions) stored "on the side"

- example: AppArmor on Linux
- configuration file with list of program + what it is allowed to access
- prevent, e.g., print server from writing files it shouldn't

capabilities

token to identify = permission to access

typically *opaque* token

some capability list examples

file descriptors

list of open files process has acces to

page table (sort of?)

list of physical pages process is allowed to access

some capability list examples

file descriptors

list of open files process has acces to

page table (sort of?)

list of physical pages process is allowed to access

list of what process can access *stored with process*

handle to access object = key in permitted object table

impossible to skip permission check!

sharing capabilities

capability-based OSes have ways of sharing capabilities:

inherited by spawned programs

- file descriptors/page tables do this

send over local socket or pipe

- usually supported for file descriptors!
- (look up `SCM_RIGHTS` — how it works different for Linux v. OS X v. FreeBSD v. ...)

Capsicum: practical capabilities for UNIX (1)

Capsicum: research project from Cambridge

adds capabilities to FreeBSD by extending file descriptors

opt-in: can set process to require capabilities to access objects

- instead of absolute path, process ID, etc.

capabilities = fds for each directory/file/process/etc.

more permissions on fds than read/write

- execute
- open files in (for fd representing directory)
- kill (for fd reporesenting process)
- ...

Capsicum: practical capabilities for UNIX (2)

capabilities = no global names

no filenames, instead fds for directories

- new syscall: `openat(directory_fd, "path/in/directory")`
- new syscall: `fexecv(file_fd, argv)`

no pids, instead fds for processes

- new syscall: `pdfork()`

alternative to per-process tables

file descriptors: different in every process

- use special functions to move between processes

alternate idea: same number in every process

- one big table

sharing token = copy number

but how to control access? make numbers hard to guess

example: use random 128-bit numbers

sandboxing

sandbox — restricted environment for program

idea: dangerous code can play in the sandbox as much as it wants

can't do anything harmful

sandbox use cases

buggy video parsing code that has buffer overflows

browser running scripts in webpage

autograder running student submissions

...

sandbox use cases

buggy video parsing code that has buffer overflows

browser running scripts in webpage

autograder running student submissions

...

(parts of) program that don't need to have user's full permissions

no reason video parsing code should be able open() my taxes

can we have a way to ask OS for this?

Google Chrome architecture

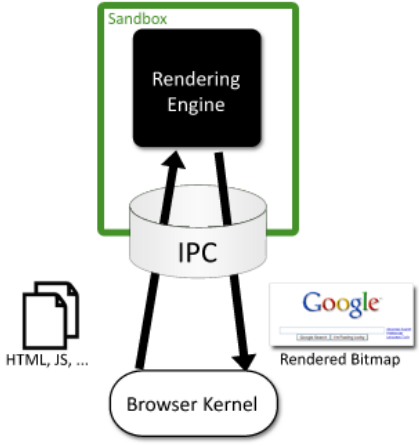


Figure 1: The browser kernel treats the rendering engine as a black box that parses web content and emits bitmaps of the rendered document.

sandboxing mechanisms

create a new user with few privileged, switch to user

- problem: creating new users usually requires sysadmin access
- problem: every user can do too much
- e.g. everyone can open network connection?

with capabilities, just discard most capabilities

- just close capabilities you don't need
- run rendering engine with only pipes to talk to browser kernel

otherwise: system call filtering

- disallow all 'dangerous' system calls

Linux system call filtering

`seccomp()` system call

"strict mode": only allow read/write/_exit/sigreturn

- current thread gives up all other privileges
- usage: setup pipes, then communicate with rest of process via pipes

alternately: setting a whitelist of allowed system calls + arguments

- little programming language (!) for supported operations

browsers use this to protect from bugs in their scripting implementations

- hope: find a way to execute arbitrary code? — not actually useful

sandbox browser setup

create pipe

spawn subprocess (“rendering engine”)

put subproces in strict system call filter mode

send subprocesses webpages + events

subprocess sends images to render back on pipe

sandboxing use case: buggy video decoder

```
/* dangerous video decoder to isolate */
int main() {
    EnterSandbox();
    while (fread(videoData, sizeof(videoData), 1, stdin) > 0) {
        doDangerousVideoDecoding(videoData, imageData);
        fwrite(imageData, sizeof(imageData), 1, stdout);
    }
}

/* code that uses it */
    FILE *fh = RunProgramAndGetFileHandle("./video-decoder");
    for (;;) {
        fwrite(getNextVideoData(), SIZE, 1, fh);
        fread(image, sizeof(image), 1, fh);
        displayImage(image);
    }
```

talking to the sandbox

browser kernel sends commands to sandbox

sandbox sends commands to browser kernel

idea: commands only allow necessary things

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sandbox

- send user input

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "

- send

needs filtering — at least no `file:` (local file) URLs

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sar

- send user input

can still read any website!
still sends normal cookies!

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to

- send user input

files go to download directory only
can't choose arbitrary filenames

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sandbox

- send user input

browser kernel displays file choser
only permits files selected by user

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

other model: every node has a copy of data

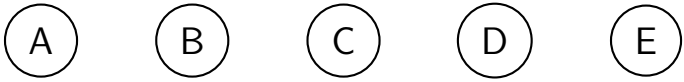
goal: work despite a few failing nodes

just require "enough" nodes to be working

for now — assume fail-stop

nodes don't respond or tell you if broken

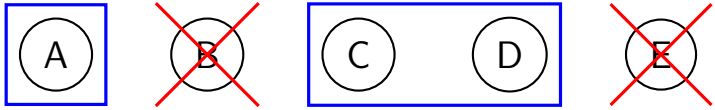
quorums (1)



perform read/write with vote of any *quorum* of nodes

any quorum enough — okay if some nodes fail

quorums (1)



perform read/write with vote of any *quorum* of nodes

any quorum enough — okay if some nodes fail

if A, C, D agree: that's enough

B, E will figure out what happened when they come back up

quorums (2)

requirement: quorums overlap

overlap = *someone in quorum* knows about every update

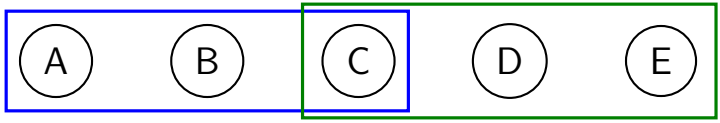
e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

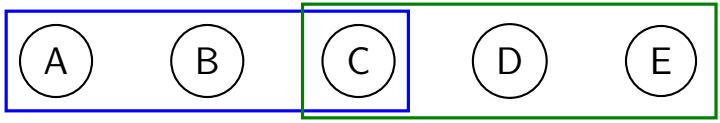
e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates

quorums (3)

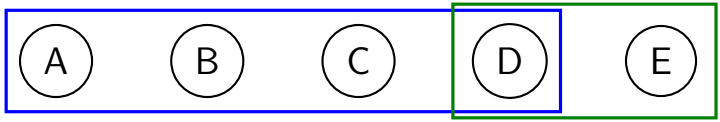
sometimes vary quorum based on operation type

example: update quorum = 4 of 5; read quorum = 2 of 5

requirement: read *overlaps* with last update

compromise: better performance sometimes, but tolerate less failures

quorums (3)



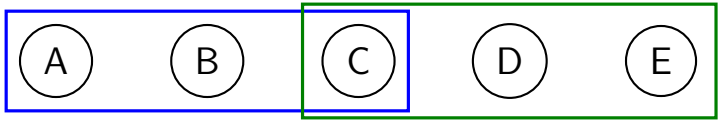
sometimes vary quorum based on operation type

example: **update** quorum = 4 of 5; **read** quorum = 2 of 5

requirement: read *overlaps* with last update

compromise: better performance sometimes, but tolerate less failures

quorums



details very tricky

- what about coordinator failures?
- how does recovery happen?
- what information needs to be logged?
- "catching up" nodes that aren't part of several updates

full details: lookup Raft or Paxis

quorums for Byzantine failures

just overlap not enough

problem: node can give inconsistent votes

tell A "I agree to commit", tell B "I do not"

need to confirm consistency of votes with other notes

need *supermajority*-type quorums

f failures — $3f + 1$ nodes

full details: lookup PBFT