CSO2: Accounts

opening a file?

```
open("/u/bradjc/private.txt", O_RDONLY)
```

say, private file on portal

on Linux: makes *system call*

kernel needs to decide if this should work or not

What is a system call?

(will discuss in more detail in next lecture)

Briefly - special syscall *instruction* run w/ *system call number* set

- system call number = open requested
- other arguments passed like a normal function

instruction switches hardware into privileged mode, runs OS function

this function decides if the "open" call is allowed to proceed

only implemented syscalls are supported

handlers written carefully (we hope) to ensure correct permission checking

how does OS decide whether syscall should proceed?

```
open("/u/bradjc/private.txt", O_RDONLY)
```

argument: needs extra metadata

what would be wrong with using...

system call arguments?

where the code calling open came from?

user IDs

most common way OSes identify "who" process belongs to:

- process = instance of running program (w/ own registers+memory)
- (we'll be more specific about processes later)

(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

core part of OS only knows number (not name!)

- core, always loaded part of OS = "kernel"
- the part of the OS with extra privs with hardware
- the part of the OS that enforces program restrictions

effective user ID is used for all permission checks

also some other user IDs

POSIX user IDs

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

process's user identified with unique number

core part of OS only knows number (not name!)

- core, always loaded part of OS = "kernel"
- the part of the OS with extra privs with hardware
- the part of the OS that enforces program restrictions

effective user ID is used for all permission checks

also some other user IDs

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 (= core part of OS) only knows numbers

standard library+databases for mapping to names

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

id command

```
: /u/bjc8c ; id
uid=858545(bjc8c) gid=90002(csfaculty)
groups=90002(csfaculty),150015(slurm-cs-brad-campbell)
```

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

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

…but: user can keep program running with video group
in the background after logout?

id and group id establish identity

who is running a process or accessing a resource

but also need *permissions*

what users can access

POSIX file permissions

POSIX files support a per-file access control list

- interface is very narrow/limited

binary permissions are *read*, *write*, and *execute*

- set for three categories: user, group, other

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

on directories, 'execute' means 'search' instead

permissions encoding

permissions encoded as 9-bit number, can write as octal: `XYZ`

octal divides into three 3-bit parts:

- user permissions (X), group permissions (Y), other permission (Z)

each 3-bit part has a bit for 'read' (4), 'write' (2), 'execute' (1)

- binary number: `0bRWE`

permissions encoding

permissions encoded as 9-bit number, can write as octal: `XYZ`

octal divides into three 3-bit parts:

- user permissions (X), group permissions (Y), other permission (Z)

each 3-bit part has a bit for 'read' (4), 'write' (2), 'execute' (1)

- binary number: `0bRWE`

`700` — user read+write+execute; group none; other none

permissions encoding

permissions encoded as 9-bit number, can write as octal: `XYZ`

octal divides into three 3-bit parts:

- user permissions (X), group permissions (Y), other permission (Z)

each 3-bit part has a bit for 'read' (4), 'write' (2), 'execute' (1)

- binary number: `0bRWE`

`700` — user read+write+execute; group none; other none

`451` — user read; group read+execute; other execute

chmod — exact permissions

```
chmod 700 file
chmod u=rwx,og= file
```

user read write execute; group/others no accesss

```
chmod 451 file
chmod u=r,g=rx,o=x file
```

user read; group read/execute; others execute

chmod — adjusting permissions

`chmod u+rx foo`

add user read and execute permissions

leave other settings unchanged

`chmod o-rwx,u=rx foo`

remove other read/write/execute permissions

set user permissions to read/execute

leave group settings unchanged

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
user:tj1a:---

# POSIX acl rule:
    # user take precedence over group entries
```

POSIX ACLs on command line

`getfacl file`

`setfacl -m 'user:tj1a:---' file`

add line to ACL

`setfacl -x 'user:tj1a' file`

REMOVE line from acl

`setfacl -M acl.txt file`

add to acl, but read what to add from a file

`setfacl -X acl.txt file`

remove from acl, but read what to remove from a file

ACLs establish permissions

identity: *who* is running a process or accessing a resource

permissions: who has what access to a file or resource

now need *enforcement*

ensure that permissions are implemented correctly

authorization checking on Unix

request made to core part of OS = system call

handler for system calls checks permissions

- no relying on libraries, etc. to do checks

files (open, rename, …) — file/directory permissions include UID or GID

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

…

superuser

user ID `0` is special

superuser or *root*

(non-Unix) or Administrator or SYSTEM or ...

some OS funtionality: only work for uid `0`

shutdown, mount new file systems, etc.

automatically passes all (or almost all) permission checks

superuser v kernel mode

processor has two modes

- kernel mode (what core part of OS uses)
- user mode (every thing else)

programs running as *superuser still in user mode*

- just change in how OS acts when program asks for things

superuser : OS :: kernel mode : hardware

certain hardware instructions/operations only enabled in kernel mode

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
- server gives tokens: “yes, really 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 as superuser and 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()` syscall changes effective user ID to owner's ID

- "extra" user IDs track what original user was

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

- sudo's code: if (original user allowed) ...; else print error

marking setuid: `chmod u+s`

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 programs are very hard to write

what if stdin, stdout, stderr start closed?

what if signals setup weirdly?

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?

...

privilege escalation

privilege escalation — vulnerabilities that allow more privileges

code execution/corruption in utilities that run with high privilege

e.g. buffer overflow, command injection

login, sudo, system services, …

bugs in system call implementations

logic errors in checking delegated operations

backup slides

authorization v authentication

authentication — who is who

authorization v authentication

authentication — who is who

authorization — who can do what

probably need authentication first...

authentication

password

hardware token

...

some security tasks (1)

helping students collaborate in ad-hoc small groups on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (2)

letting students assignment files to faculty on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (3)

running untrusted game program from Internet?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

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

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/...

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

one (bad?) idea: setuid program to read grade for assignment

`./print_grade assignment`

outputs grade from `all-grades/assignment/USER.txt`

a very broken setuid program

print_grade.c:

```
int main(int argc, char **argv) {
    char filename[500];
    sprintf(filename, "all-grades/%s/%s.txt",
            argv[1], getenv("USER"));
    int fd = open(filename, O_RDWR);
    char buffer[1024];
    read(fd, buffer, 1024);
    printf("%s:␣%s\n", argv[1], buffer);
}
```

HUGE amount of stuff can go wrong

examples?

other privileged escalation issues

sudo problem: trusted code that's supposed to enforce restriction can be fooled into not really enforcing it

also can occur in other contexts:

system call letting program access things it shouldn't?

browser letting web page javascript access things it shouldn't?

web application giving users access to files they shouldn't have?

mobile phone OS allowing location access without location permission?

...

another very broken setuid program (setup)

allow users to print files, but only if less than 1KB

another very broken setuid program

print_short_file.c:

```
int main(int argc, char **argv) {
    struct stat st;
    if (stat(argv[1], &st) == -1) abort();
    // make sure argv[1] is owned by user running this
    if (st.st_uid != getuid()) abort();
    // and that it's less than 1 KB
    if (st.st_size >= 1024) abort();
    char command[1024];
    sprintf(command, "print␣%1000s", argv[1]);
    system(command);
    return EXIT_SUCCESS;
}
```

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 …	—

link: create new directory entry for file

- another option: rename, symlink ("symbolic link" — alias for file/directory)
- another possibility: run a program that creates secret file (e.g. temporary file used by password-changing program)

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`
...

lots of time spent following symbolic links when program opening toprint.txt

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

exercise

which (if any) of the following would fix for a TOCTTOU vulnerability in our setuid printing application? (assume the Unix-permissions without ACLs are in use)

[A] **both before and after** opening the path passed in for reading, check that the path is accessible to the user who ran our application

[B] after opening the path passed in for reading, using `fstat` with the file descriptor opened to check the permissions on the file

[C] before opening the path, verify that the user controls the file referred to by the path **and** the directory containing it