Privileged Programs

Acquiring Privileges

- How can privileged operations be performed?
- More precisely, how can an unprivileged process request that something privileged take place?
- What is privilege?

Types of Privilege

- Hardware restriction—ask the OS (or maybe it just isn't allowed)
- OS restriction—use a privileged process
- Application restriction—application-dependent

What is a Privileged Process?

- One that has access to some resource not generally available
- Doesn't necessarily correspond to `root` or `Administrator`
- More secure systems have many types of privilege

File System-Related Privileges

- Who can read from or write to certain files?
- Example: a mail system on a multi-user computer system

Mailer Daemon: Sending Mail

- In principle, an unprivileged operation
- For convenience, have one well-written mail daemon per system
  - Accept mail from mail user agents (MUAs)
  - Add some header lines (MessageID, Received, maybe From)
  - For remote mail, create a network connection to the receiving machine
  - For local mail, try to open a mailbox owned by that other user
  - Attempt delivery; queue and retry if necessary
- Use “privilege” to protect its own queue files.
- And: use “privilege” to write to someone else’s mailbox

Which security attribute is invoked here?

Mailer Daemon: Sender Security

Confidentiality	Protect the confidentiality of the email itself
Integrity	Prevent mail from being tampered with
Availability	Prevent mailer crashes and email deletion; deliver mail

The Lessons of Mailer Security

- There’s no hardware privilege
- For remote mail, the mailer has no resources unavailable to other users
- Mailer protection (except for local delivery) is just another case of protecting one user from another

What if Sending Remote Mail is Privileged?

- Many sites block direct outbound access to port 25
- Why? Firewalls, spam senders, and “bots”
- How can we restrict which users can connect to which port?
- Either make network access go through the file system —`/dev/tcp/25/another.host`—or add a separate permission mechanism

What Are Other Privileges?

What Are Other Privileges?

- Override DAC (or override it for specific purposes)
- Mount a file system in a restricted fashion
- Mount a file system with no restrictions
- Operate on file as owner
- Change MAC label
- Set time
- Assign privileges
- Linux has about 40 privileges via its `capabilities`

What is the Principle of Least Privilege

- No subject should have more privileges than it needs
- Obvious reason: it can't misuse abilities it doesn't have
- ☞ Very important in case of application compromise
- Difficult to do properly, since one privilege often implies another
- Example: if I can overide the DAC "write" privileges, I can overwrite an executable that a more privileged user will invoke, and thus get that user's privileges

How Processes Get Privileges

- Inheritance
- File attributes
- Ask a privileged process to perform the operation for you

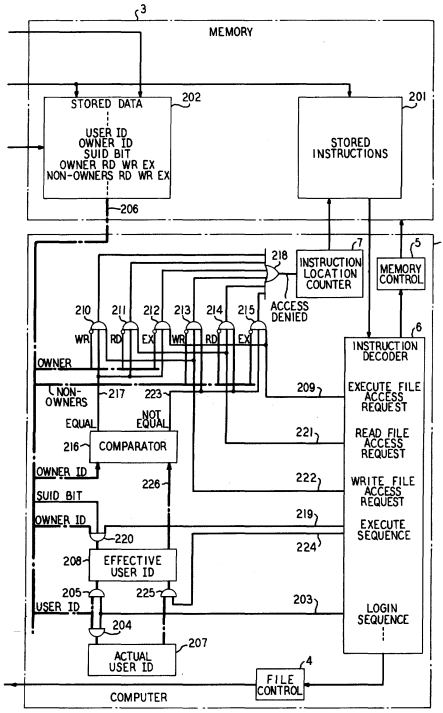
Inheritance

- Many privileges are inherited from parent process (necessary in Unix, where almost every command is run in a separate process)
- ☞ The source of a lot of danger!
  - Example: Unix uid
  - Sometimes associated with username: the login mechanism sets initial privileges, and your shell inherits them
  - Obviously, child processes cannot inherit privileges the parent doesn't possess

File Attributes: SetUID (SUID)

- Fundamental privilege acquisition mechanism in Unix
- Invented in 1973 by Dennis Ritchie
- Patented—U.S. Patent 4,135,240, issued January 1979
- Major step towards *principle of least privilege*
- A serious security risk if used improperly

FIG. 1B


3
MEMORY
STORED DATA
USER ID
OWNER ID
SUID BIT
OWNER RD WR EX
NON-OWNERS RD WR EX
202
201
STORED INSTRUCTIONS
206
1
7
5
218
INSTRUCTION LOCATION COUNTER
MEMORY CONTROL
210
211
212
213
214
215
ACCESS DENIED
WR
RD
EX
WR
RD
EX
6
INSTRUCTION DECODER
OWNER
NON-OWNERS
217
223
209
EXECUTE FILE ACCESS REQUEST
EQUAL
NOT EQUAL
221
READ FILE ACCESS REQUEST
216
COMPARATOR
OWNER ID
226
222
WRITE FILE ACCESS REQUEST
SUID BIT
OWNER ID
219
EXECUTE SEQUENCE
220
224
208
EFFECTIVE USER ID
205
225
203
LOGIN SEQUENCE
USER ID
204
207
ACTUAL USER ID
4
FILE CONTROL
COMPUTER

What is SUID?

- When the program is executed, it acquires the privileges of the file's owner
- This feature is available for all uids
- Similar feature for groups: setgid
- If a file is setuid `root`, it executes with `root` privileges

Setting and Querying the SUID Bit

- Set:
  `chmod u+s file`
- Query:
  `ls -l file`

  The “x” for owner is replaced by a “s”

```
$ ls -l /bin/su
-rwsr-xr-x 2 root root 71288 Feb 27 2013 /bin/su
```

What Does This Do?

```
$ cp /bin/sh .
$ ls -l sh
-r-xr-xr-x  1 smb  smb  109768 Sep 15 22:49 sh
$ chmod u+s sh
$ ls -l sh
-r-sr-xr-x  1 smb  smb  109768 Sep 15 22:49 sh
```

What Does This Do?

```
$ cp /bin/sh .
$ ls -l sh
-r-xr-xr-x  1 smb  smb  109768 Sep 15 22:49 sh
$ chmod u+s sh
$ ls -l sh
-r-sr-xr-x  1 smb  smb  109768 Sep 15 22:49 sh
```

- It created a setuid shell
- Anyone who executed that shell would have all of my privileges
- Not a good thing to do. . .

How Did I Do That Safely?

```
$ (umask 077; mkdir f)
$ cd f
$ ls -ld .
drwx------  2 smb  smb  512 Sep 18 22:49 .
```

Safely Doing Dangerous Things

- Create a directory that no one else can access
- Use `umask` to do it atomically
- Create the dangerous file in a locked directory
- Only I (and `root`) can get to that directory

Safely Doing Dangerous Things

- Create a directory that no one else can access
- Use `umask` to do it atomically
- Create the dangerous file in a locked directory
- Only I (and `root`) can get to that directory
- To get to that dangerous shell, an attacker would either need to have my privileges—in which case the attack buys nothing—or `root` privileges, in which case I've already lost the game.

Combining Permission Settings

- Use some permissions to restrict acccess
- Use SUID or SetGID to grant more authority to invoker
- What does this do?

```
$ ls -l shutdown
-r-sr-xr--  1 root  operator  14463 Sep  2 01:38 shutdown
```

A Restricted shutdown Command

Note the permissions:

```
-r-sr-xr-- 1 root operator
```

- `r-s` SUID `root`
- `r-x` Executable by group “operator”
- `r--` Readable but not executable by others

The command runs with `root` permissions, but only a select few can get those permissions.

Why is SUID Good?

- Available to all users; does not require special privilege
- Used by mailers, printer daemons, games, etc
- Conceptually simple way to provide controlled interface to some resources

Why is SUID Bad?

- Available to all users; does not require special privilege
- Writing *secure* SUID programs is *hard*
- Too easy to give away permissions
- Per-user permissions aren’t granular enough

Peter Neumann on SUID

It is precisely BECAUSE it allows easy implementation that it is so frequently misused—by people who don't know better. Use of "setuid" opens up the possibility of a variety of security flaws, including Trojan horses, search-path traps, etc., and tends to substantially widen the perimeter of trust. I'm not sure that anyone knows how to characterize "proper use" completely—if it is indeed possible at all.

RISKS Digest 4:53, 1 March 1987

Fred Grampp and Robert Morris on SUID

SUID programs should be used only when there is no other way to get a desired result. On most *UNIX* systems, perhaps a dozen SUID programs, excluding games, are really needed. A lax attitude about SUID programs, combined with a 'quick and dirty' programming style, can produce disasters. . . It is difficult, when users are writing all but the most trivial programs, to determine in advance that the program will be correct. Programs sometimes do the most amazing things in unforeseen circumstances.

UNIX Operating System Security
AT&T Bell Laboratories Technical Journal 63:8, Part 2, October 1984

Linux and SUID

```
$ find /*bin /usr/*bin -perm -4000 -print
/bin/su
/bin/ping
/bin/fusermount
/bin/mount
/bin/umount
/bin/ping6
/sbin/mount.ecryptfs_private
/sbin/mount.nfs
/usr/bin/newuidmap
/usr/bin/chsh
/usr/bin/pkexec
/usr/bin/newgidmap
/usr/bin/at
/usr/bin/newgrp
/usr/bin/passwd
/usr/bin/chfn
/usr/bin/sudo
/usr/bin/gpasswd
```

Also: 16 setgid programs; 7 with other privileges (mostly for networking)

- Most are in two categories: mounting external devices and changing data fields in `/etc/passwd`
- A few, e.g., sudo, are about letting certain users have more privileges, at least temporarily
- All of these have to do their own permission-checking—and do it very carefully!

What is the Problem with SUID?

- The bad guy is running the program and supplying the inputs
- The bad guy controls the environment
- Many subtle traps!

Confusing a Program

```
$ PS1='% ' bash
% ulimit -f 0
% echo foo >/tmp/foo
File size limit exceeded
$ ls -l /tmp/foo
-rw-r--r--  1 smb  wheel  0 Sep 19 00:04 /tmp/foo
```

What if this happens to the passwd command?
This is one example of problems from *inherited environment*

Inherited Environments

Processes inherit many things from parent processes

- UID and GID
- Open files
- `ulimit` values
- Linux capabilities
- Environment variables
- More...

Which will cause trouble? How do you know?

How Do You Know?

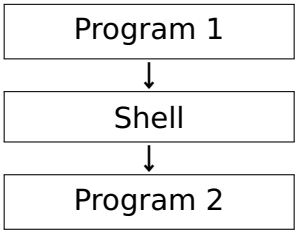
- How do you know what inherited attributes will cause trouble?
- The only solution: *thorough* knowledge of your environment
- Thus: avoid complexity
- But even simple, necessary things can be complex
- From the man page for `strftime`:
  “The environment variables TZ and LC_TIME are used.”
- What is LC_TIME? What happens if it’s set to an unexpected value?

Shells are Horribly Complex

- Shells look at all sorts of environment variables and can do unexpected things if some are set
- Traditional and known: `PATH` can affect which program is actually executed
- Traditional and little known: `IFS` specifies characters that will split a line into different arguments
- `ENV` and `BASH_ENV`: specifies a file of shell commands that is (sometimes) read before anything else is done
- In fact, there are more than 50 environment variables that are used by bash—do you know them all?

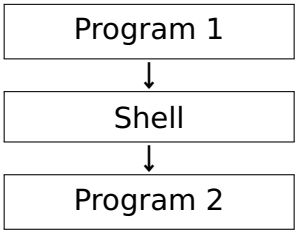
Invoking a Program via a Shell

What You Think is Happening



Invoking a Program via a Shell

What You Think is Happening



What is Actually Happening

The Alternative: Message-Passing

- A program runs with certain permissions
- It sets up some sort of local communications channel
- Other programs send messages to that channel, and receive responses
- Used by Windows, some Unix subsystems

Practical Difficulties

- How does the initial program start?
- What sorts of channels are available?
- Can you control access to those channels?
- What are the messages and responses like?

Initial Startup

- Very much OS-dependent
- On some systems, any user can have a program started at boot time:
  ```
  $ crontab -l
  @reboot echo ‘hostname‘ reboot | mail smb
  ```
- Sometimes program is invoked automatically when channel is contacted

A “Daemon Daemon”

- Have an always-running “daemon daemon”
- It listens for requests for other daemons, and fires up the proper program with the proper privilege
- (Also used for network daemons)
- Must be very privileged, since it can run programs as any other user

Some Types of Channels

- Local sockets (“UNIX-domain sockets”)
- Message-passing queues
- Controlled RPC

Access Control

- Different channels have different permission mechanisms
- Very much OS-dependent
- Getting these right is just as important as file permissions

UNIX-Domain Sockets

- Appear in file system
- *Not* accessed like regular files; use “socket” primitives instead
- Permissions on this “file” not always honored—version-depenent!
- Solution: set directory permissions instead

```
# ls -ld private
drwx------  2 postfix  wheel  512 Sep 10 23:31 private
# ls -ld private/maildrop
srw-rw-rw-  1 postfix  wheel  0 Sep 10 23:31 private/maildrop
```

Unix-Domain Sockets on Linux

In the Linux implementation, pathname sockets honor the permissions of the directory they are in. Creation of a new socket fails if the process does not have write and search (execute) permission on the directory in which the socket is created.

On Linux, connecting to a stream socket object requires write permission on that socket; sending a datagram to a datagram socket likewise requires write permission on that socket. POSIX does not make any statement about the effect of the permissions on a socket file, and on some systems (e.g., older BSDs), the socket permissions are ignored. Portable programs should not rely on this feature for security.

Passing Credentials

Linux programs can pass “credentials” and security labels over Unix-domain sockets

```
struct ucred {
    pid_t pid;    /* Process ID of the sending process */
    uid_t uid;    /* User ID of the sending process */
    gid_t gid;    /* Group ID of the sending process */
};
```

via `SCM_CREDENTIALS` and `SCM_SECURITY`.

Note carefully: this implies that the receiving application has to do the right thing with this data

Why is Message-Passing Good?

- Bad guys can’t invoke the privileged commands
- No opportunity to control the environment
- Less opportunity for certain harmful programming mistakes

Why is Message-Passing Bad?

- Fundamentally, you're writing network servers
- We know from experience how hard it is to get them right!
- You have to design a language for the channel, and perhaps marshall/unmarshall arguments

Capabilities

- Capability: a bit-string that gives access to some resource
- Not the same as Linux “capabilities”!
- Sometimes, a cryptographically protected string
- Can be copied
- Effectively attribute-based access control
- Issues: acquisition, forgery, revocation
- Rarely used in practice, except for Web cookies

What is the Real Issue with Acquiring Privileges?

- Granting selective access is *hard*
- *Never* trust anything that can be controlled by the enemy
- Make sure you know the enemy's powers. . .

Questions?

(Peregrine falcons, Riverside Church, May 16, 2019)