Secure Architecture Principles

Slides credit: Dan Boneh
What Happens if you can’t drop privilege?

- In what example scenarios does this happen?
  - A service loop
  - E.g., ssh

- Solution?
  - Privilege separation
  - Identifying operations that need privileges
  - Separate original code into master (privileged) and slave (unprivileged)

- Example: ssh
Privilege Separation

● Process:
  ● Step 1: Identify which operations require privilege
  ● Step 2: rewrite programs into 2 or more parts

● Approach:
  ● Manual
    ● Have been done on security-critical programs, e.g., ssh
    ● Labor-intensive and may miss privileged operations
  ● Automatic (e.g., Privtrans [Brumley-Song-2004])
    ● Automatic inference of privileged operations using a few initial annotations
    ● Automatic source-to-source rewriting
      ● Privileged code move into master
      ● Unprivileged code move into slave
      ● Stubs for inter communication
Automatic Privilege Separation

- **Step 1**: automatic inference of privileged data and operations
  - Given some initial annotations of privileged data and/or operations, infer what other data/operations are privileged
  - Idea: can be viewed as a form of static taint analysis
  - Approach:
    - Define qualifier \_priv\_ and \_unpriv\_
    - Operations on \_priv\_ results in \_priv\_

```c
int _priv_ a;
int _priv_ b;
int _priv_ c;
int _priv_ g;
```
Automatic Privilege Separation

- Step 2: automatic source-to-source transformation
  - Move privileged data and code to Master
  - For call to privileged functions, change the call site to a wrapper function which marshals the args on slave side and sends them to Master’s stub
  - Similar stubs on returns for unprivileged return values
Privilege Separation at Runtime

Slave

Main Execution

RPC Request
RPC Reply

Master

Privileged Server

State Store

Dawn Song
Summary: Privilege Separation

- Only master is privileged, usually much smaller
- Slave is unprivileged
- Bug in slave cannot harm master, cannot gain privilege
- How to protect master from a compromised slave?
  - Fault isolation: e.g., running in different processes
Setuid programming

- Be Careful with Setuid 0!
  - Root can do anything; don’t get tricked
  - Principle of least privilege – change EUID when root privileges no longer needed
Non-Language-Specific Vulnerabilities

// Part of a setuid program
if (access("file", W_OK) != 0) {
    exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof (buffer));

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access(&quot;file&quot;, W_OK)</td>
<td>Returns 0 if the user invoking the program has write access to “file”</td>
</tr>
<tr>
<td>(it checks the real uid, the actual id of the user, as opposed to the effective uid, the id associated with the process)</td>
<td></td>
</tr>
<tr>
<td>open(&quot;file&quot;, O_WRONLY)</td>
<td>Returns a handle to “file” to be used for writing only</td>
</tr>
<tr>
<td>write(fd, buffer ...)</td>
<td>Writes the contents of buffer to “file”</td>
</tr>
</tbody>
</table>
Time-of-Check-to-Time-of-Use (TOCTTOU)

// Part of a setuid program
if (access("file", W_OK) != 0) {
    exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof (buffer));

// After the access check
symlink("/etc/passwd", "file");

// Before the open, "file" points to the password database

access("file", W_OK)  // Returns 0 if the user invoking the program has write access to "file"
open("file", O_WRONLY) // Returns a handle to "file" to be used for writing only
write(fd, buffer ...)  // Writes the contents of buffer to "file"
symlink("/etc/passwd", "file") // Creates a symlink from "file" to "/etc/passwd". A symbolic link is a reference to another file, so in this case the attacker causes "file" (which they have privileges for) to point to "/etc/passwd". The program then opens "/etc/passwd" instead of "file".
...;
...
exec( );

if (access("file", W_OK) != 0) {
    exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));

Owner 0
SetUID

program

Owner 25
-rw-r--r--
"file"

Owner 0
-rw-------
"/etc/passwd"

symlink("/etc/passwd", "file")

symlink("/etc/passwd", "file")

RUID 25
EUID 25

RUID 25
EUID 0

if (access("file", W_OK) != 0) {
    exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));

Access check

Launch program

Writes to /etc/passwd
The Flaw?

- Code assumes FS is unchanged between `access()` and `open()` calls – Never assume anything…
- An attacker could change file referred to by “file” in between `access()` and `open()`
  - Eg. `symlink("/etc/passwd", "file")`
  - Bypasses the check in the code!
  - Although the user does not have write privileges for `/etc/passwd`, the program does (and the attacker has privileges for file, so they are allowed to create the symbolic link)
TOCTTOU Vulnerability

- In Unix, often occurs with file system calls because system calls are not atomic
- But, TOCTTOU vulnerabilities can arise anywhere there is mutable state shared between two or more entities
  - Example: multi-threaded Java servlets and applications are at risk for TOCTTOU
Unix summary

- Good things
  - Some protection from most users
  - Flexible enough to make things possible
- Main limitation
  - Too tempting to use root privileges
  - No way to assume some root privileges without all root privileges
Access control in Windows (since NTFS)

- Some basic functionality similar to Unix
  - Specify access for groups and users
    - Read, modify, change owner, delete
- Some additional concepts
  - Tokens
  - Security attributes
- Generally
  - More flexibility than Unix
    - Can define new permissions
    - Can give some but not all administrator privileges
Identify subject using SID

- Security ID (SID)
  - Identity (replaces UID)
    - SID revision number
    - 48-bit authority value
    - variable number of Relative Identifiers (RIDs), for uniqueness
  - Users, groups, computers, domains, domain members all have SIDs
Process has set of tokens

- **Security context**
  - Privileges, accounts, and groups associated with the process or thread
  - Presented as set of tokens
- **Security Reference Monitor**
  - Uses tokens to identify the security context of a process or thread
- **Impersonation token**
  - Used temporarily to adopt a different security context, usually of another user
Object has security descriptor

● Security descriptor associated with an object
  ● Specifies who can perform what actions on the object

● Several fields
  ● Header
    ● Descriptor revision number
    ● Control flags, attributes of the descriptor
      ● E.g., memory layout of the descriptor
  ● SID of the object's owner
  ● SID of the primary group of the object
  ● Two attached optional lists:
    ● Discretionary Access Control List (DACL) – users, groups, …
    ● System Access Control List (SACL) – system logs, ..
Example access request

Access request: write
Action: denied

- User Mark requests write permission
- Descriptor denies permission to group
- Reference Monitor denies request
  (DACL for access, SACL for audit and logging)

Priority:
- Explicit Deny
- Explicit Allow
- Inherited Deny
- Inherited Allow
Impersonation Tokens  (compare to setuid)

- Process adopts security attributes of another
  - Client passes impersonation token to server
- Client specifies impersonation level of server
  - Anonymous
    - Token has no information about the client
  - Identification
    - server obtain the SIDs of client and client's privileges, but server cannot impersonate the client
  - Impersonation
    - server identify and impersonate the client
  - Delegation
    - lets server impersonate client on local, remote systems
Creating Jail
Creating Jail: chroot

- “Change root” - changes the root directory of current process
- Example (within a process):

```bash
chroot /tmp/guest
su guest
```

- The root directory for this process (“/”) is now resolved to “/tmp/guest”, and the EUID is set to “guest” (handled by the operating system)
Creating Jail: chroot

- If done properly, the process can no longer access files outside of “/tmp/guest”
  
  Eg: `open("/etc/passwd", "r")`  \(\Rightarrow\)
  
  - `open("/tmp/guest/etc/passwd", "r")`

- Must consider other processes which it may need to invoke (eg. from /usr/bin)- these must be provided for the environment
Escaping from jails

Early escapes: relative paths

```python
open( "../../etc/passwd", "r") =>
open("/tmp/guest/../../etc/passwd", "r")
```
Many ways to escape jail as root

- Create device that lets you access raw disk
- Send signals to non chrooted process
- Reboot system
- Bind to privileged ports
FreeBSD jail

Stronger mechanism than simple chroot

To run:  jail jail-path hostname IP-addr cmd

- calls hardened chroot (no "../.." escape)
- can only bind to sockets with specified IP address and authorized ports
- can only communicate with processes inside jail
- root is limited, e.g. cannot load kernel modules
FreeBSD jail

User 1 running as root within their own jail.

User 2 running as root within their own jail.

Actual system, with an actual root user
Problems with chroot and jail

Coarse policies:

- All or nothing access to parts of file system
- Inappropriate for apps like a web browser
  - Needs read access to files outside jail
    (e.g. for sending attachments in Gmail)

Does not prevent malicious apps from:

- Accessing network and messing with other machines
- Trying to crash host OS
System Call Interposition
System call interposition

Observation: to damage host system (e.g. persistent changes) app must make system calls:
- To delete/overwrite files: `unlink`, `open`, `write`
- To do network attacks: `socket`, `bind`, `connect`, `send`

Idea: monitor app’s system calls and block unauthorized calls
Implementation options

- Completely kernel space (e.g. GSWTK)
- Completely user space (e.g. program shepherding)
- Hybrid (e.g. Systrace)
• ptrace (process trace) allows one process to
  • monitor signals sent to and from the controlled process, as well as state in
    registers, file descriptors, and memory
  • control another process by altering its internal state
Initial implementation (Janus) [GWTB’96]

Linux **ptrace**: process tracing

process calls: `ptrace(..., pid_t pid, ...)`
and wakes up when `pid` makes sys call.

Monitor kills application if request is disallowed
Complications

- If app forks, monitor must also fork
  - forked monitor monitors forked app
- If monitor crashes, app must be killed
- Monitor must maintain all OS state associated with app
  - current-working-dir (CWD), UID, EUID, GID
  - When app does “cd path” monitor must update its CWD
    - otherwise: relative path requests interpreted incorrectly

```c
cd("/tmp")
open("passwd", "r")

cd("/etc")
open("passwd", "r")
```
Problems with ptrace

**Ptrace** is not well suited for this application:

- Trace all system calls or none
  - inefficient: no need to trace “close” system call
- Monitor cannot abort sys-call without killing app

Security problems: **race conditions**

- Example: symlink: “file” □ “/etc/passwd”
  
  Proc 1: open(“file”)
  monitor checks and authorizes
  Proc 2: “file” □ “/etc/passwd”
  OS executes open(“file”)

Classic TOCTOU bug: time-of-check / time-of-use

Problems with ptrace: time not atomic
Which of the following sequence of events is a TOCTOU exploit?

P1: open(“temp”, “w”)  
P2: temp □ “important.txt”  
monitor checks P1’s write request  
OS opens “temp” (if monitor approves)

P2: temp □ “important.txt”  
P1: open(“temp”, “w”)  
monitor checks P1’s write request  
OS opens “temp” (if monitor approves)

P1: open(“temp”, “w”)  
monitor checks P1’s write request  
OS opens “temp” (if monitor approves)  
P2: temp □ “important.txt”  
OS opens “temp” (if monitor approves)