Privilege Separation and Isolation

Dawn Song

dawnsong@cs.berkeley.edu

The Story Continues…

- **Programs have bugs and may not find all the bugs ahead of time**
- **What can we do?**
  - **Build security mechanisms to minimize damage**
- **Examples**
  - **Priviledge separation to prevent priviledge escalation**
  - **Isolation to protect other parts of the program and other programs**
  - **Sandboxing to limit the damage it does to the system**
  - **General concept: reference monitor**

Privileged Programs

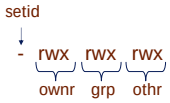
- **Privilege management is coarse-grained in today's OS**
  - **Root can do anything**
- **Many programs run as root**
  - **Even though they only need to perform a small number of priviledged operations**
- **What's the problem?**
  - **Privileged programs are juicy targets for attackers**
  - **By finding a bug in parts of the program that do not need privilege, attacker can gain root**

What Can We Do?

- Drop privilege as soon as possible
- Ex: a network daemon only needs privilege to bind to low port # (<1024) at the beginning
  - Solution?
  - Drop privilege right after binding the port
- What benefit do we gain?
  - Even if attacker finds a bug in later part of the code, can't gain privilege any more
- How to drop privilege?

Unix file security

- Each file has owner and group
- Permissions set by owner
  - Read, write, execute
  - Owner, group, other
  - Represented by vector of four octal values



- Only owner, root can change permissions
  - This privilege cannot be delegated or shared
- Setid bits
  - Talk about this in a sec

Effective user id (EUID)

- Each process has three Ids
  - Real user ID (RUID)
    - » same as the user ID of parent (unless changed)
    - » used to determine which user started the process
  - Effective user ID (EUID)
    - » from set user ID bit on the file being executed, or sys call
    - » determines the permissions for process
      - file access and port binding
  - Saved user ID (SUID)
    - » So previous EUID can be restored
- Real group ID, effective group ID, used similarly

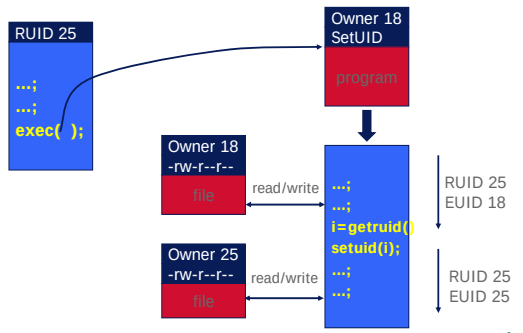
Process Operations and IDs

- **Root**
  - ID=0 for superuser root; can access any file
- **Fork and Exec**
  - Inherit three IDs, except exec of file with setuid bit
- **Setuid system calls**
  - seteuid(newid) can set EUID to
    - » Real ID or saved ID, regardless of current EUID
    - » Any ID, if EUID=0
- **Why do we need to save previous EUID?**
- **Details are actually more complicated**
  - Several different calls: setuid, seteuid, setreuid

Setid bits on executable Unix file

- **Three setid bits**
  - Setuid – set EUID of process to ID of file owner
  - Setgid – set EGID of process to GID of file
  - Sticky
    - » Off: if user has write permission on directory, can rename or remove files, even if not owner
    - » On: only file owner, directory owner, and root can rename or remove file in the directory

Example



Setuid programming

- **Be Careful!**
  - Root can do anything; don' t get tricked
  - Principle of least privilege – change EUID when root privileges no longer needed
- **Setuid scripts**
  - This is a bad idea
  - Historically, race conditions
    - » Begin executing setuid program; change contents of program before it loads and is executed

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 (priviledged) and slave (unprivileged)

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
    - » 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

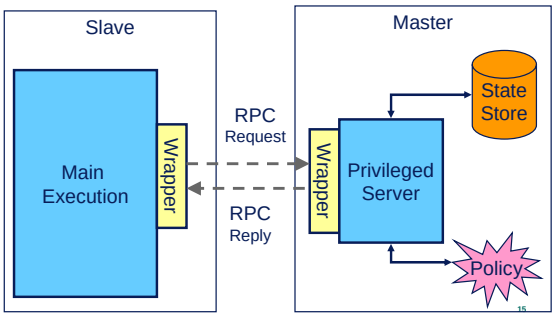
- **Step 1: automatic inference of privileged data and operations**
  - **Given some initial annotations of privileged data and/or operations, infer what other data/operations are privileged**
  - **Idea: can be viewed as a form of static taint analysis**
  - **Approach:**
    - » **Define qualifier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

```
int _priv_ a;
Int _priv_  f();
int b = f(a);          _priv_ b
c= c+b;                _priv_ c
g(c);                  _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



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

Fault Isolation

- Fault Isolation
  - The fault in one program or one part of the code cannot harm other programs or other parts of the code
  - Very important for security in running untrusted or untrustworthy code
  - "Harmness"
    - » E.g., read/write memory it's not supposed to
- Hardware fault isolation
  - Processes
  - What properties/protection does process provide?
    - » Memory protection
    - » Other resources such as file handles are separated as well
  - Works well for some applications

Disadvantage of Hardware Fault Isolation

- Process inter communication is expensive
  - Add significant performance overhead if often
- Why is process inter communication expensive?
  - Trap from user to kernel back to user
  - Context switch is expensive
    - » Flush TLB, cache miss, etc.
  - Often 2-3 orders of magnitude more expensive than normal procedure call

How to Address This?

- **Software Fault Isolation (SFI)**
- **Question:**
  how to protect a piece of code from harming other parts of the program even though they run in the same address space?

Administravia

- **Hw5 is out**

Motivation

- **Today's systems are designed to be extensible**
  - **OS kernel module/drivers**
  - **Browser plug-ins**
- **Extension accounts for over x% of Linux kernel code**
  - **x=70 [Chou et. al.]**
- **Windows XP desktops**
  - **Over 35,000 drivers with over 120,000 versions [Swift et. al.]**
- **Drivers cause 85% of reported failures in Windows XP [Swift et. al.]**

Desired Properties of Extensible Architecture

- **Efficiency**
- **Security model: extension code may be**
  - **Malicious**
  - **Buggy**
- **Protection**
  - **Extension should not read and/or write to certain regions in host ← Isolation, sandbox**
    - » **Do no harm to others**
    - » **Why do we care about Read?**
  - **Other more sophisticated security policies**
  - **Need more efficient mechanisms than hardware fault isolation**

Software Fault Isolation

- **Idea: insert code in extension code to ensure certain security properties**
- **SFI [Wahbe et. al. 93]**
  - **Software fault isolation**
  - **Security property to guarantee: Extension code only writes and jumps to dedicated data and code region**
  - **How to ensure this?**