Isolation and Reference Monitor

Slide credit: Dan Boneh
Confinement (I): Hardware

- **Hardware**: run application on isolated hw (air gap)
Confinement (II): Firewall

- Firewall: isolate internal network from the Internet
Confinement (III): VM

- **Virtual machines**: isolate OS’s on a single machine
Confinement (IV): Processes

- **Processes:**
  - Isolate a process in a single operating system
  - System Call Interposition
Confinement (V): SFI

- **Threads:** Software Fault Isolation (SFI)
  - Isolating threads sharing same address space
Implementing confinement: Reference Monitor

Key properties:

– **Mediates requests** from applications
  - Implements protection policy
  - Enforces isolation and confinement

– Must **always** be invoked (complete mediation)
  - Every application request must be mediated

– **Tamperproof/fail safe**
  - Reference monitor cannot be killed
  - or if killed, then monitored process cannot accessing anything requiring reference monitor’s approval

– **Small** enough to be analyzed and validated
Creating Jail: chroot

Often used for “guest” accounts on ftp sites

To use do:  (must be root)

```
chroot /tmp/guest
```
```
“/tmp/guest”
```
```
su guest
```

root dir “/” is now “/tmp/guest”

EUID set to “guest”

Now “/tmp/guest” is added to file system accesses for applications in jail

```
open(“/etc/passwd”, “r”)
```
```
⇒
```
open(“/tmp/guest/etc/passwd”, “r”)
```

⇒ application cannot access files outside of jail
Escaping from jails

Early escapes: relative paths

```c
open( "../../etc/passwd", "r") ⇒
open("/tmp/guest/../../etc/passwd", "r")
```

**chroot** should only be executable by root.

– otherwise jailed app can do:
  
  • create dummy file "/aaa/etc/passwd"
  • run **chroot** "/aaa"
  • run **su root** to become root

(bug in Ultrix 4.0)
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
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
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)
Linux ptrace: process tracing

process calls: ptrace (... , pid_t pid , ...) and wakes up when pid makes sys call.
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
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: me → mydata.dat
  
  proc 1: open(“me”)  
  monitor checks and authorizes  
  proc 2: me → /etc/passwd  
  OS executes open(“me”)  

  **not atomic**

Classic **TOCTOU bug**: time-of-check / time-of-use
Non-Language-Specific Vulnerabilities

- int openfile(char *path) {
  struct stat s;
  if (stat(path, &s) < 0)
    return -1;
  if (!S_ISRREG(s.st_mode)) {
    error("only regular files allowed!");
    return -1;
  }
  return open(path, O_RDONLY);
}

- stat(path, &s) writes information about the file at “path” to the struct s
- S_ISRREG(s.st_mode) determines whether the file whose information is in “s” is regular
- open(path, O_RDONLY); opens the file at “path” in read only mode
The Flaw?

- Code assumes FS is unchanged between `stat()` and `open()` calls – Never assume anything...
- An attacker could change file referred to by `path` in between `stat()` and `open()`
  - From regular file to another kind
  - Bypasses the check in the code!
  - If check was a security check, attacker can subvert system security
- Time-Of-Check To Time-Of-Use (TOCTTOU) vulnerability
  - Meaning of `path` changed from time it is checked (`stat()`) and time it is used (`open()`)

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
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)
Alternate design: systrace

- systrace only forwards monitored sys-calls to monitor (efficiency)
- systrace resolves sym-links and replaces sys-call path arguments by full path to target
- When app calls `execve`, monitor loads new policy file
Ostia: a delegation architecture

Previous designs use filtering:
- Filter examines sys-calls and decides whether to block
- Difficulty with syncing state between app and monitor (CWD, UID, ..)
  - Incorrect syncing results in security vulnerabilities (e.g. disallowed file opened)

A delegation architecture:
Ostia: a delegation architecture

[GPR’04]

- Monitored app disallowed from making monitored sys calls
  - Minimal kernel change  (... but app can call close() itself )

- Sys-call delegated to an agent that decides if call is allowed
  - Can be done without changing app
    (requires an emulation layer in monitored process)

- Incorrect state syncing will not result in policy violation

- What should agent do when app calls execve?
  - Process can make the call directly.  Agent loads new policy file.
Policy

Sample policy file:

```
path allow /tmp/
path deny /etc/passwd
network deny all
```

Manually specifying policy for an app can be difficult:

- Systrace can auto-generate policy by learning how app behaves on “good” inputs
- If policy does not cover a specific sys-call, ask user
  ... but user has no way to decide

Difficulty with choosing policy for specific apps (e.g. browser) is the main reason this approach is not widely used
NaCl: a modern day example

- game: untrusted x86 code

- Two sandboxes:
  - outer sandbox: restricts capabilities using system call interposition
  - Inner sandbox: uses x86 memory segmentation to isolate application memory among apps