TEL AVIV UNIVERSITY

Introduction to Information Security

0368-3065, Spring 2015

Lecture 4:
Process confinement (1/2)

Eran Tromer

Slides credit:
Dan Boneh and John Mitchell, Stanford

Process confinement

Running untrusted code

- We often need to run buggy/unstrusted code:
  - Executable code from untrusted Internet sites:
    - viewers, codecs for media players, “rich content”, “secure banking”, toolbars
    - JavaScript, Java applets, .NET, flash, …
  - Old or insecure applications: ghostview, Outlook
  - Buggy legacy software (sendmail, bind, …)
  - Checking homework exercises
  - Honeypots
  - Digital right management
- <u>Goal</u>: if application <u>misbehaves</u>, stop it.
  - Kill process, alert user, write to log, report to central service…

Confinement

- **Confinement**: ensure application does not deviate from pre-approved behavior
- Can be implemented at many levels:
  - **Hardware**: isolated hardware (“air gap”)
    - Difficult to manage
    - Sufficient?
  - **Processes in OS**
    Isolates a process in a single operating system
    - Separate spaces: virtual memory, view of filesystem
    - System call interface can be controlled (“system call interposition) to
  - **Virtual machines**: isolate OSs on single hardware

  Application-level:
  - Isolating threads sharing same address space:
    - Software Fault Isolation (SFI), e.g., Google Native Code
  - Interpreters for non-native code
    - JavaScript, Java Virtual Machine, .NET CLR

Implementing confinement

- Key component: **reference monitor**
  - **Mediates requests** from applications
    - Implements protection policy
    - Enforces isolation and confinement
  - Must **always** be invoked
    - Every application request must be mediated
  - **Tamperproof**
    - Reference monitor cannot be killed
    - … or if killed, then monitored process is killed too
  - **Small** enough to be analyzed and validated

Simple process confinement

A simple example: chroot

- Often used for “guest” accounts on ftp sites

- To confine the current process, run (as root):

# chroot /home/guest	root dir “/” is now “/home/guest”
# su guest	EUID set to “guest”

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

open(“/etc/passwd”, “r”) $\Rightarrow$

open(“/home/guest/etc/passwd”, “r”)

$\Rightarrow$ application cannot access files outside of jail

Jailkit

Problem: all utility programs (ls, ps, vi) must live inside jail

- **jailkit** project: auto builds files, libs, and dirs needed in jail environment
  - **jk_init**: creates jail environment
  - **jk_check:** checks jail env for security problems
    - checks for any modified programs,
    - checks for world writable directories, etc.
  - **jk_lsh**: restricted shell to be used inside jail
- Restricts only filesystem access. Unaffected:
  - Network access
  - Inter-process communication
  - Devices, users, … (see later)

Escaping from jails

- Early escapes: relative paths

  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 chroot jail as root

- Create device that lets you access raw disk
  ```
  mknod sda b 8 0
  cat malicious-boot-record > sda
  ```
- Send signals to non chrooted process
- Reboot system
- Bind to privileged ports (<1024)
  - fake NFS (network file system) requests from port 111
  - usurp incoming packets to TCP port 80
- Use hard links to files outside the chroot
- Load kernel modules

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 process inside jail
  - root is limited, e.g. cannot load kernel modules

Problems with chroot and jail

- Coarse policies:
  - All-or-nothing access to file system
  - Inappropriate for apps like web browser
    - Needs read access to files outside jail (e.g. for sending attachments in gmail)
- Do not prevent malicious apps from:
  - Accessing network and messing with other machines
  - Trying to crash host OS

System call interposition

for process-level confinement

System call interposition

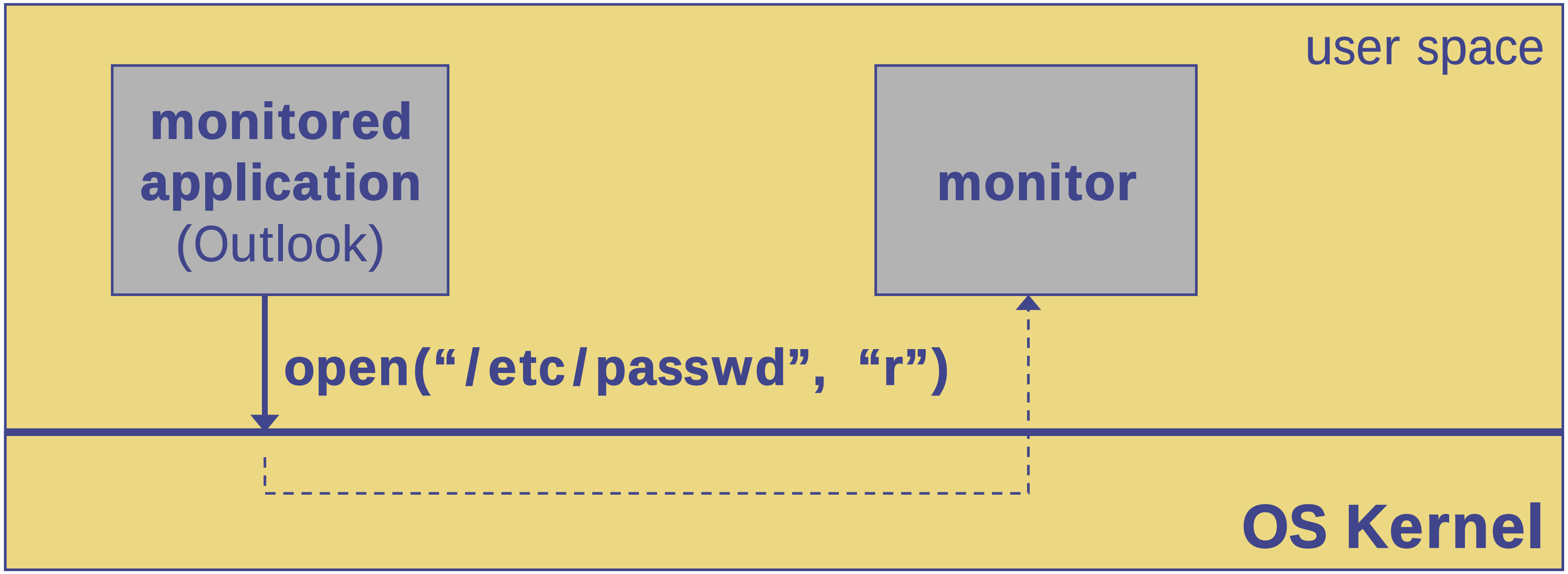
- Observation: to damage host system (i.e. make persistent changes) app must make system calls
  - To delete/overwrite files: unlink, open, write
  - To do network attacks: socket, bind, connect, send
- Monitor app system calls and block unauthorized calls
- Implementation options:
  - Completely kernel space (e.g. GSWTK)
  - Completely user space
    - ~~Capturing system calls via dynamic loader (LD_PRELOAD)~~
    - Dynamic binary rewriting (program shepherding)
  - Hybrid (e.g. Systrace)

Initial implementation (Janus)

- Linux ptrace: process tracing

  tracing process calls: **ptrace (… , pid_t pid , …)**

  and wakes up when **pid** makes sys call.



- Monitor kills application if request is disallowed

Complications

- Monitor must maintain all OS state associated with app
  - current-working-dir (CWD), UID, EUID, GID
  - Whenever app does “cd path” monitor must also update its CWD
    - otherwise: relative path requests interpreted incorrectly
- If app forks, monitor must also fork
  - Forked monitor monitors forked app
- Monitor must stay alive as long as the program runs
- Unexpected/subtle OS features: file description passing, core dumps write to files, process-specific views (chroot, /proc/self)

Problems with ptrace

- ptrace is too coarse for this application
  - Trace all system calls or none
    - e.g. no need to trace "close" system call
  - Monitor cannot abort sys-call without killing app

- Security problems: **race conditions**
  - <u>Example</u>: symlink: me -> mydata.dat

time

proc 1: open("me")

monitor checks and authorizes

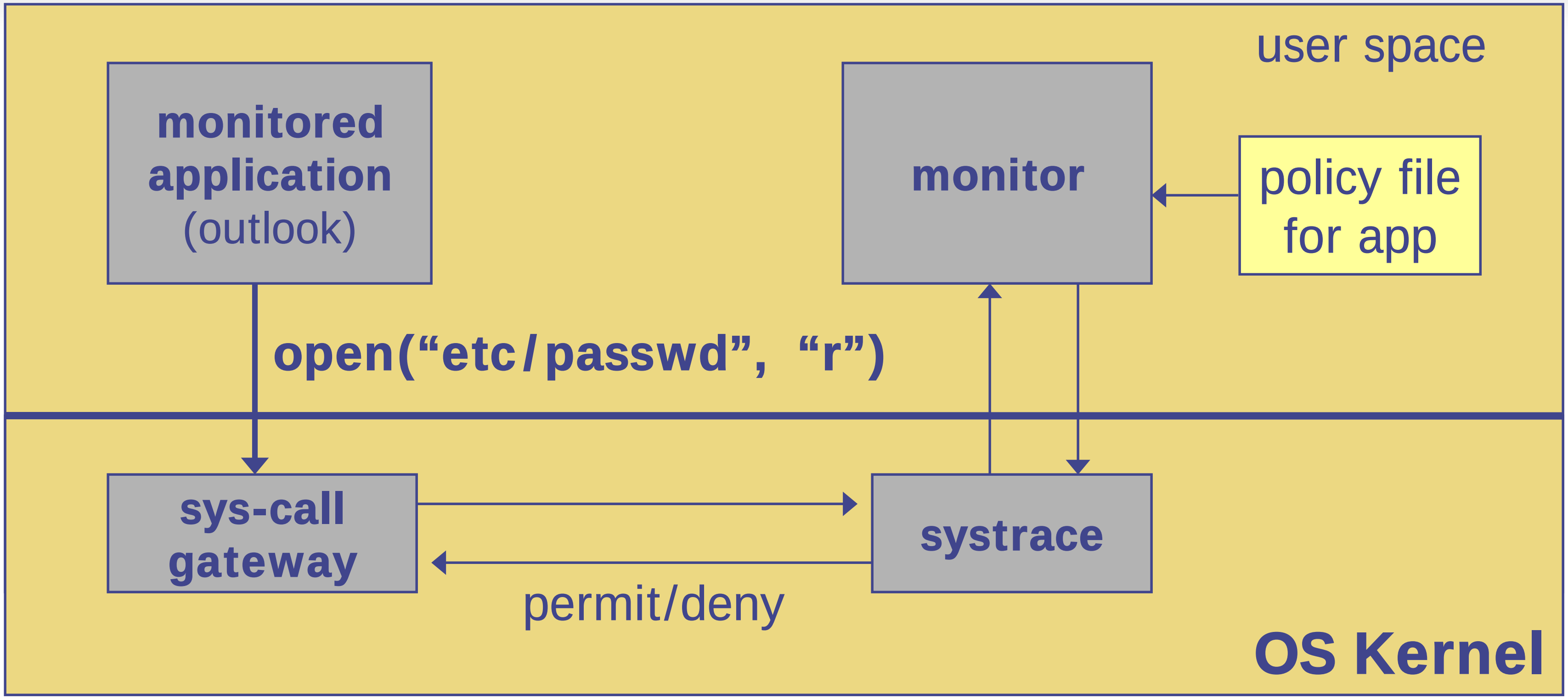
proc 2: me -> /etc/passwd

OS executes open("me")

not atomic

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

Improved system call interposition: Systrace



- Systrace only forwards monitored sys-calls to monitor (saves context switches)
- Systrace resolves sym-links and replaces sys-call path arguments by full path to target
- When app calls execve, monitor loads new policy file
- Fast path in kernel for common/easy cases, ask userspace for complicated/rare cases

Systrace policy

- Sample policy file:

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

- Specifying policy for an app is quite difficult
  - Systrace can auto-gen 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 main reason this approach is not widely used