SOFTWARE SANDBOXES

CARSTEN WEINHOLD, BJÖRN DÖBEL
Outline

- Why and what to isolate?
- Machine-Level Isolation
  - Virtual Machines
  - OS-level isolation: chroot, BSD Jails, OS Containers, SELinux
- Application-Level Isolation
  - Chromium Architecture
  - Native Client
The Need for Isolation

- Large-scale: Multi-user systems
  - Security: Prevent other users from reading/modifying my data...
  - Sharing: … but allow this for certain exceptions.
- Fair distribution of resources (CPU time / network bandwidth) among users
- Small-scale: Integrate software from differing sources
  - Web browser: websites, plugins
Types of Isolation

- **Fault Isolation**
  - A faulting application shall not take down others.

- **Resource Isolation**
  - Global resources shall be distributed fairly across all users
  - What is fair?

- **Security Isolation**
  - Applications shall not access or modify others' data.
Physical Separation

App 1

App 2

App 3

"air gap"
Physical Separation

- Advantages:
  - Achieves isolation
  - Different OS/software setups

- Disadvantages:
  - Resource overcommit
  - Administration effort
  - Sharing difficult
Virtual Machines

• Idea: better resource utilization by running multiple virtual machines on a single physical
Virtual Machine Monitor

- Provides virtual hardware environment
  - Guest OS runs as on real hardware
  - Intercept (and emulate) privileged instructions
  - Virtual devices

  - Type 1 – Bare metal
    - Runs as OS directly on hardware
    - e.g., VMware ESXi, Xen

  - Type 2 – hosted
    - Part of a native OS (e.g., kernel module)
    - e.g., KVM, VirtualBox
Isolation with VMs

- **Advantages**
  - Isolation
  - Better resource utilization
  - Different OS/SW setups

- **Disadvantages**
  - Management
  - Slight Performance overhead
  - Sharing still difficult

Many more implementation issues: See lectures on Microkernel-Based Operating Systems and Microkernel Construction
Multi-User System

- Unix path name resolution
  - Each process has a lookup root (default: /)
  - `open("/foo/bar/baz")` traverses file system hierarchy starting from this root

- (Limited) ACLs to manage access rights
  - Single group/owner not sufficient for complex access policies

- Idea: Restrict users/programs' access to parts of the file system → `chroot`
Chroot Example

- **Process A:**
  - Global file system access
  - `open("/bin/ls")` → returns file descriptor to `/bin/ls`

- **A creates process B:**
  ```c
  pid = fork();
  if (pid == 0) // child
  {
    chdir("/var/Domain1");
    chroot("/var/Domain1");
    setuid(some_user);
    execve("program B");
  }
  ```
Chroot Example

- Process B now has /var/Domain1 set as its lookup root
  - open("/bin/ls") returns file descriptor to /var/Domain1/bin/ls

- Ideally, no access to anything outside /var/Domain1 possible for process B

- Sharing between users:
  - Make files/directories visible in different locations (e.g. linking)
Chroot is not Isolation

- Chroot is meant to restrict file access of well-behaving applications
  - Intended for software testing

- No restrictions on
  - Loading kernel modules
  - Opening network connections
  - Reading `/dev/kmem`
  - Tracking other processes (e.g., through `ps` / `top`)
Breaking out of chroot

• Step 1: Become root
  • Find an exploit as described in last week’s lecture

• Step 2:

```c
fd = open(".", O_RDWR);
mkdir("./tmpdir", 0755);
chroot("./tmpdir");
fchdir(fd);
for (i = 0; i < 1024; ++i)
  chdir("..");
chdir(".");
```
Breaking out of chroot

Starting as process B, chroot'ed to /var/Domain1...

```c
fd = fopen(".", O_RDWR);
→ fd now contains valid file descriptor of /var/Domain1
```
Breaking out of chroot

Starting as process B, `chroot`ed to `/var/Domain1...

```c
fd = fopen(".", O_RDWR);
→ fd now contains valid file descriptor of `/var/Domain1`

mkdir("./tmpdir", 0755);
→ creates new directory 'tmpdir' below current one
Breaking out of chroot

chroot("./tmpdir")
→ sets B's resolution root to /var/Domain1/tmpdir
→ so B can't access anything above, right?

But we still have a file descriptor pointing outside!

fchdir(fd);
→ sets the current working directory to /var/Domain1
→ this is POSIX-certified behavior
Breaking out of chroot

- Now `chdir("..")` in a long loop
- At some point we will hit the real root directory
- Now finally
  ```c
  chroot(".");
  ```
  sets B's resolution root to `/.`
- Mission accomplished.
• Based on chroot + kernel modifications

• Prohibited:
  • Loading kernel modules
  • Modify network configuration
  • (Un-)mount file systems
  • Create device nodes
  • Access kernel runtime parameters (sysctl)

• Permitted:
  • Run programs within jail (working directory...)
  • Signalling processes within a jail
  • Modification of in-jail file system
  • Bind sockets to TCP/UDP ports defined at jail creation
Jails Implementation

• **Added jail system call**
  • Create jail structure → unmodifiable after setup
  • Attached to every process
    - Only processes within a jail can add processes to it
    - No breaking out of chroot
  • Adapted other system calls
    - Limit PID/GID/TID-based system calls
• Had to adjust some drivers
  • e.g., virtual terminal needs to belong to specific jails
Access Control Theory

- **Discretionary Access Control (DAC)**
  - Security (isolation) enforced based on object-subject relationship
  - Linux: File System → file ownership

- **Mandatory Access Control (MAC)**
  - Isolation based on object – (subject x operation) relationship
  - e.g., Program A with UID X may read a file;
    Program B with UID X may also write it
  - Linux: File System ACLs (limited to 3 operations)

- **Role-Based Access Control (RBAC)**
  - Subjects can have dynamic roles assigned
  - Access based on object-role relationship

- **Principle of Least Privilege**
• RBAC for Linux (co-developed by NSA...)
• Type Enforcement
  • Processes are placed in dedicated sandboxes (domains)
  • Fine-grained configuration per domain
    - Which files can be accessed? (And how?)
    - Which network ports can be bound to?
    - Can the app render to an X11 window?
    - Can the app fork() new processes? In which domain?
SELinux Policies

- Policy files define
  - User roles
    \[\text{user} \ j\oe \rightarrow \text{role} \ \text{user}\_t\]
  - Object types
    \[\text{dir} \ /\text{etc}/\text{selinux} \rightarrow \text{policy}\_\text{src}\_t\]
  - Permissions
    \[\text{r}\_\text{dir}\_\text{file} (\text{user}\_t, \text{policy}\_\text{src}\_t) \rightarrow \text{user}\_t \text{may read} \text{policy}\_\text{src}\_t\]

- `checkpolicy` compiler generates loadable kernel module to enforce rules
SELinux Architecture
LSM: Linux Security Modules

- Loadable Kernel Modules

- `struct` security_operations {
  [..]
  int (*file_open) (struct file *,
                  const struct cred *);
  [..]
};

- `extern int` register_security(
  struct security_operations*);
static int do_entry_open(struct file *f, ..., const struct cred *cred)
{
    [...] error = security_file_open(f, cred);
    if (error) { ... }
    [...]
Containers

- Jails, SELinux: security isolation + some fault isolation
  - Process cannot modify state outside its jail
  - Fine-grained SELinux policies may also limit fault propagation
    - But configuration is a mess...

- Resource isolation still missing

- Enter: container-based virtual machines
  - Recent gain in popularity:
    https://linuxcontainers.org
    http://www.docker.com
Containers

- Full virtualization is expensive
  - Implementation overhead
    - Need to have pass-through drivers available
  - Management overhead
    - VM configuration in addition to setup of guest OS
  - Runtime overhead (though small)

- Often we don't need all features
  - Many use cases warrant "A Linux installation"
• Jails-like Linux modification
  • Extended chroot
    – Chroot barrier: prevent breaking out
  • PID / resource name spaces + filtering
  • Network isolation
    – only bind apps to predefined set of IP addresses / ports

• Share libraries / kernel across VM instances
VServer: Resource Isolation

- Goal: Fair distribution of resources (e.g. CPU time)
- But what is fair?
  - Fair share → each VM gets the same amount of compute time
  - Proportional Share → VMs with more processes get larger amount of resources
- Linux: Completely Fair Scheduler (CFS)
  - All processes get the same amount of time
  - No notion of process-VM mappings
VServer: Token-Bucket Scheduler

- Each VM has a bucket
- Every timer tick removes a token from VM's bucket
- If bucket is empty: remove all VM's processes from run queue until threshold of tokens has been refilled
- Refill: over time according to some policy
- Allows to implement proportional and fair share
• Network: use existing Linux traffic shaping mechanisms
  • Bandwidth reservations
  • Shares → specify how non-reserved bandwidth is distributed between VMs

• Disk: rely on Linux disk scheduler to do the right thing
  • Disk is less about isolation, more about optimizing accesses
Linux namespaces

• All modern container implementations based on Linux namespaces

• Virtualizes these resources:
  • Mount (mnt), process (pid), network (net)
  • Inter-process communication (ipc)
  • Host and domain names (UTS)
  • User IDs (user), Control group (cgroup), time

• Basis of Docker, LXC, Rkt, Singularity, …
Application-level Isolation
Web Browser, ca. 2008
Monolithic Browsers

- Web pages communicate through DOM
  - Unrelated page can inspect and modify data
  - Access Control: Same-Origin Policy
    - http://www.example.com
    - http://www.example.com/p2
    - https://www.example.com
- Web pages may include data from different sources (e.g., iframes)
- User credentials stored by browser
  - May be (mis-)used by other pages
- Per-page isolation infeasible: web apps need multiple pages
  - Calendar window
  - Email compose window
  - ...
Chromium: Isolating Web Programs
Chromium: Isolating Web Programs

- News
  - DOM
  - HTML
  - Network
  - UI
  - Plugins

- Mail
  - DOM
  - HTML
  - Network
  - UI
  - Plugins

- Calendar
  - DOM
  - HTML
  - Network
  - Storage
  - UI

Browsing Instance
Web Processes

- No direct storage access
- Single thread of execution
- chroot to empty temp dir

- Might require FS access
- Fault isolation

Rendering Process

Mail

Calendar

DOM

HTML

JS

UI

Network

Storage

Browser Kernel

Plugins
• Isolate web pages into OS processes

• Difficult:
  • determine exact boundaries...
  • … while maintaining compatibility

• Gain:
  • Security & Fault Isolation between web pages
  • Performance → parallel rendering possible
  • Accountability

• Enter unlimited possibilities of cloud wonderland...
Problems with Plugins

- **Goals:**
  - Native code execution (JIT or interpreted)
  - Access to local resources (disk, …)

- **Problems:**
  - Circumvent browsers' security mechanisms
  - Arbitrary code execution possible

- **Solutions**
  - Ask for user approval before running plugin
  - Language-level security (e.g. Java Class Loader) → often open up new attack surface
  - Process Isolation → protects web pages, can still exploit system call interface
Native Client (NaCL)

- Allow plugins (NaCl modules) compiled to native x86 code

- **Inner Sandbox**: limit execution to module's code and data

- **Outer Sandbox**: System Call Policy Enforcement (think: SELinux)
NaCL: App Model

Web Browser
- DOM
- HTML
- JS
- Image Viewer

NaCl Module
- Native Imaging Application
- Service Runtime
- System Calls

Plugin API
NaCL Modules

- NaCl module and service runtime in same address space
  - Module code must not break out of its text/data region
  - But we need well-defined ways to
    - Perform system calls (if policy permits)
    - Communicate with web page through plugin API

- Solution: Dedicated compiler (adapted GCC) that enforces rules on NaCl modules
NaCL Module Rules

- Once loaded, the binary is not writable
  - Enforced using mprotect()
  - Prevents self-modifying code

- Binary is statically linked
  (start address == 0, entry point = 64 kB)
  - No dynamically loaded code → allows static validation during startup
  - Predefined starting point required for load-time validation
  - Address restrictions: later
NaCL Module Rules

- All indirect control transfers use a nacljmp pseudo-instruction
  - Disable ret / function pointers → harden stack smashing

- The binary is padded up to the nearest page with at least one \texttt{hlt} instruction
  - Prevent jump to arbitrary address → will trigger \texttt{hlt}
NaCL Module Rules

- The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary
  - Alignment restrictions for indirect jumps (coming soon)

- All valid instruction addresses are reachable by disassembly that starts at the base address
  - Need access to all code for analysis

- All direct control transfers target valid instructions
  - Prevent jump into middle of instruction
• Problem: x86 code may jump to arbitrary address (e.g., using `ret` or `jmp *%<register>`)  
• NaCl: Alignment makes sure that every 32-byte aligned address is a valid instruction  
• Use `nacljmp` instead of indirect control flow:  
  ```c
  and *%<reg>, 0xFFFFFFFFFE0
  jmp *%<reg>
  ```  
• Result: code only contains jumps to valid targets  
• Disallowed instructions  
  • x86 segment modifications  
  • `ret`  
  • `syscall / int 0x*`  
• No support for POSIX signals  
  • They use the SS segment themselves  
• Remaining issue: controlled calls into/out of the sandbox
NaCL Data Flow Integrity

- NaCl code may jump into trampoline (32-byte aligned)
- Each 32-byte aligned word is either
  - An entry to a service routine call
    - mmap / sbrk
    - thread creation
    - Plugin API calls
  - Or a HLT instruction
- Trampoline may contain unsafe code
NaCL Summary

- Plugins in isolated process
- Compiler enforces
  - Reliable Disassembly
- Sandbox enforces
  - Data Integrity
  - Control Flow Integrity
  - No unsafe instructions

Result: We can play Quake in the browser!

Update: Works with Javascript now, too!


• Reis, Gribble "Isolating Web Programs in Modern Browser Architectures", EuroSys 2009

• Yee et al. "Native Client: A Sandbox for portable, untrusted x86 native code", IEEE Security & Privacy 2009