Sandboxing

(1) Motivation

- Depending on operating system to do access control is not enough.
  - For example: download software, virus or Trojan horse, how to run it safely?
  - Risks:
    - Unauthorized access to files, services, devices.
    - Installation of unwanted/hostile applications
    - Manipulate your connections
  - Discussion:
    - Q: How do you want to run them?
    - Q: How to confine the untrusted software to restricted environment?
    - Q: How to confine the privileged software to restricted environment?

- We want to have more access controls that are not provided by OS.
  - Sandboxing Technologies

---

(2) Sandboxing Approaches

- Isolation: Building a prison, so the program running in this prison can only SEE a subset of the resources, and they cannot see the rest of the resources.
  - Chroot: (e.g., used by ftp)
  - Virtual Machine (Isolated OS)

- Adding extra access control: the program can SEE everything, but an extra layer of access control is added, such that the program cannot make arbitrary access (even though the OS allows it).
  - Intercept system calls (Janus)
  - Intercept library function calls (Library interposition)
  - Java Sandboxing
  - Some need to change OS, some does not

---

(3) Chroot

- Idea: How to create a limited view for a process, so the process can only see files in the limited scope?
  - Change the meaning of "root directory".
  - For example, set /tmp to be the root directory. Therefore, the access to /etc/shadow is actually an access to /tmp/etc/shadow.

- Advantage
  - The proram cannot escape from the prison.

- Disadvantage
  - Copy all the necessary files to the corresponding directory, e.g., required files, libraries, etc.
  - Difficult to manage when files can be updated.

- How is chroot implemented in Minix?

In file: src/fs/fproc.h

```
struct fproc {
        ...
        struct inode *fp_rootdir; /* pointer to current root dir */
        ...
    }
```

In file: src/fs/stadir.c

```
    PUBLIC int do_chroot()
    {
    /* Perform the chroot(name) system call. */

      register int r;

      if (!super_user) return(EPERM);       /* only su may chroot() */
      r = change(&fp->fp_rootdir, name, name_length);
      return(r);
    }
```

- Security
  - Q: Can you use "cd .." to get out of the prison?
    No. When you call "cd ..", the system call `chdir("..")` will ensure not to pass the root directory.
    In `Minix`, this is guaranteed by checking whether the current directory is the same as the root directory specified in `fp_rootdir`.

  - Only the superuser can use `chroot`, why?
    Without this restriction, anybody can become root.

- How to gain the superuser privilege when normal user can run `chroot`?
  - Hint 1: Assume that you gain the root privilege within the prison, you have to lose the privilege when you get out of the prison (by killing the process), **how can you regain the root privilege**?

    A: using Set-UID shell binaries.

- Hint 2: **How can you gain the root privilege within the prison?** There are two ways to gain root, one way is to exploit vulnerability; the other way is to know the root's password. Assume there is no vulnerability to exploit. How can you get the password? Where does "su" look for the password?

  A: create your own /etc/shadow within the prison.

- How do you find "su" in the prison? Since "su" must be a set-UID program, simply copying it to the prison does not work.

  A : using hard link, e.g., **"ln /bin/su /tmp/su**". "ln" preserves the owner and the permission (including the set-UID bit).

> A **soft link** is a pointer to a filename, and that filename is what points to an inode, which has data and various other information. Because a soft link is a pointer to a filename, it can span file systems and when you delete the filename, the soft link is broken and the data is gone.
>
> A **hard link**, however, points directly to an inode, not a filename. It points to the same inode as the file you're hard linking to. In a case like that, you can delete the original file, and still access the data through the hard link because the hard link is still a pointer to that inode.
>
> Because the hard link is a pointer to the inode, you can't span file systems. That's also why often hard links are considered security risks and only root can create them. Hard links mean that someone can delete a file they're trying to get rid of, but if there's a hard link to it hidden somewhere on the file system that data is still accessible.

- How to break out of `chroot()` prison after becoming root?
  - Hint 1: remember how system prevents you from using "`cd ..`" to get out of prison. Assume that the current directory is "/tmp", the root of the prison. Directly using "cd.." won't work because "/tmp" is the same as `fp_rootdir`.
  - Hint 2: If your current directory is not the same as `fp_rootdir`, you can always conduct "cd ..". However, you do want to do "cd .." at the "root" directory! Maybe we can stay at the "/tmp", the root of the prison and at the same time make `fp_rootdir` not "/tmp". How can we make it happen?
  - Solution: do another `chroot("NewPrison")` within the current prison.
    - You will still stay at `/tmp`,
    - But, `fp_rootdir` now becomes `/tmp/NewPrison`
    - `chdir(..)` will be able to proceed to /, the actual root.
    - Sometimes, `chroot()` also change the current directory to the new root. In this case, you need to open a file handler of a directory outside of the new prison, and then use `fchdir()` to change your current directory to the outside of the new prison.

- Other Problems of `chroot()`
  - The compartmentalization does not extend to the process or networking spaces
  - Once you become root within the prison, you can control the processes outside of the prisons
    - Sending signals to other process
    - Debug other process using `ptrace(2)` syscall.

(4) FreeBSD Jail

- When a process is placed in a jail, it, and any descendents of the process created after the jail creation, will be in that jail.
- A process may be in only one jail, and after creation, it cannot leave the jail.
- Jails are created when a privileged process calls the `jail()` syscall.
- Restrictions in a jail
  - File system: similar to `chroot()`, but fixed the security problems with the `chroot()`.
  - Process: processes within he jail cannot interact or even verify the existence of processes outside the jail – processes within the jail are prevented from delivering signals to processes outside the jail, as well as connecting to those processes with debuggers.
  - Network: each jail is bound to a single IP address. Processes within the jail may not make use of any other IP address for outgoing or incoming connections.
  - Other restrictions
    - Modifying the running kernel by direct access and loading kernel modules is prohibited
    - Modifying any of the network configuration, interfaces, addresses, and routing table is prohibited.
    - Mounting and unmouting file systems is prohibited
    - Creating device nodes is prohibited.
    - Accessing raw, divert, or routing sockets is prohibited.
    - etc.

---

(5) Janus (meaning: Roman God of doors and gates)

- Motivation:
  - An extra layer of access control (in addition to the one provided by the OS).
  - Another way to think: an extension of the OS reference monitor.
  - Regulating the invocation of systems calls and the arguments.
  - Intercept system calls.

- Janus by Goldberg, Wagner et al. from Berkeley
  - Firewall between an application and the OS
  - Please read both papers on Janus

- Restricting system calls (system call interposition)
  - *How to restrict system calls?*
    - Early Janus: Using system call interposition via debugging support provided by OS, e.g. *ptrace*, */proc*.
    - Current Janus: Modify operating system: loadable kernel module.

- User-level Approach
  - The application runs until it performs a system call. At this point, it is put to sleep, and the tracing process wakes up. The tracing process determines which system call was attempted, along with the arguments to the call. It then determines whether to allow or deny this system call based on the policy. *How to achieve this?*

  - ***truss***: a program that traces system calls and signals of a process.
    - Quote from the manual of truss:

"The truss utility executes the specified command and produces a trace of the system calls it performs, the signals it receives, and the machine faults it incurs. Each line of the trace output reports either the fault or signal name or the system call name with its arguments and return value(s)."

- `% truss -p pid`
- Use the `ptrace`(2) system call.
- `ptrace(2)` cannot trace a few system calls without tracing all the rest as well.

ptrace(2): allows a parent process to control the execution of a child process or another process.

"The `ptrace()` function allows a parent process to observe and control the execution of another process, and examine and change its core image and registers. Its primary use is for the implementation of breakpoint debugging. The parent can initiate a trace by calling fork(2) and having the resulting child do a PTRACE_TRACEME, followed (typically) by an exec(3). Alternatively, the parent may commence trace of an existing process using PTRACE_ATTACH.

- ***/proc:*** request callbacks on a per-system call basis.
  - ***/proc*** virtual file system
  - Allows direct control of the traced process's memory.
  - Can request callbacks on a per-system call basis.

- Sample security policy
  - Example: **path allow read,write /tmp/***
  - Concentrate on the ***open*** system call, and always allow ***read*** and ***write*** calls. Why?
  - The application is placed in a particular directory; it cannot ***chdir*** out of this directory.
  - The application is allowed read access to certain carefully controlled files referenced by absolute pathnames, such as shared libraries and global configuration files.

- Kernel-level Approach
  - An example sequence of events
    - A sandboxed process makes a system call *open("foo");* this traps into the kernel at the system call entry point.
    - A hook at the system call entry point redirects control to *mod_janus*, since open is sensitive system call.
    - *mod_janus* notifies *janus* that a system call has been request and puts the calling process to sleep.
    - *Janus* wakes up and requests all relevant details about the call from *mod_janus*, which it uses to decide whether to allow or deny the call. It then notifies *mod_janus* of its decision.
    - If the call is allowed, control is returned to the kernel proper and system call execution is resumed as normal. If *janus* decides to deny the call, an error is returned to the calling process and the system call is aborted.

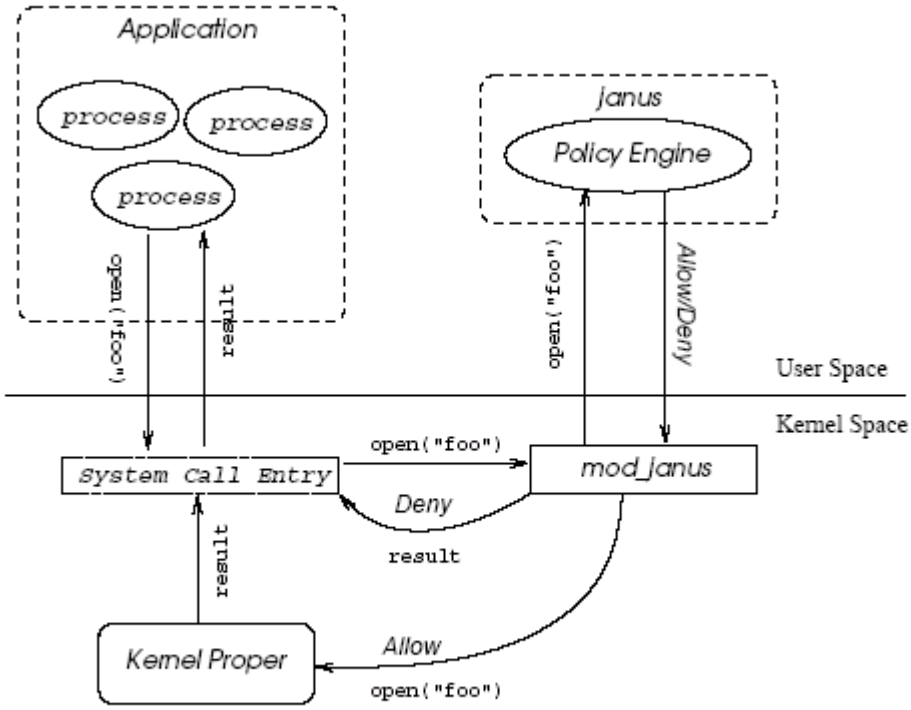

Figure 1. System Call Interposition in Janus

---

(6) Library Interposition (using dynamic library)

- Motivation: intercept library calls, so a layer of access control can be enforced on library function invocation.

- Library interposition Techniques
  - Using the **LD_LIBRARY_PATH** environment variable
    - **LD_LIBRARY_PATH** tells the system where to look for dynamic link libraries.

```
% setenv LD_LIBRARY_PATH  "your new library"
```

  - Using **LD_PRELOAD** environment variable
    - **LD_PRELOAD** preload the specific library function to avoid future searches.

```
% cc -o malloc_interposer.so -G -Kpic malloc_interposer.c
% setenv LD_PRELOAD $cwd/malloc_interposer.so
```

- *Is this scheme secure?* **(DISCUSSION)**
  - Not very secure and can be by-passed.
  - A program can change the environment variables directly.
  - A program can use static linking to avoid the search.

(7) Virtual Machine

- Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller virtual machines (VMs), each running a separate operating system instance.
  - Virtual machines are isolated from one another; the execution of one does not affect the others.
  - We can run untrusted programs in a new virtual machine.

- History of VM
  - 1960s and 1970s:
    - Hardware was expensive (mainframe).
    - Be able to run Multiple OS (by multiple people) on expensive hardware.
    - IBM VM/370.
  - Later:
    - Hardware is cheap.
    - There is no need to run multiple OS by multiple people on the same machine.
    - VM research becomes nearly dead.
  - Current:
    - There are so many operating systems.
    - There is a great need to run multiple OS by ONE person on the same machine.
    - Security.
  - Other Usages:
    - Testing for various operating systems without changing machines.
    - Configuring machines (8M memory space vs. 32M memory space).
    - Security.
    - Virtual networking.

- Two Types of virtual machines:
  - Pure software emulation
  - VMM (Virtual Machine Monitor): a majority of the virtual processor's instructions are executed on the real processor.
    - Most of the instructions of virtual machines can be directly run on the real processor
    - Privileges instructions cannot be directly run, they will trap to VMM, and VMM will then emulates the privileged operations for the virtual machines.

- Two Types of VMM (Use Figures)
  - Type I: runs on a bare machine. It is an OS.
  - Type II: runs as an application (host OS, guest OS), e.g., Vmware.