Computer Security

05. Confinement – Application Sandboxes

Paul Krzyzanowski

Rutgers University

Spring 2019

The sandbox

sand•box, ’san(d)-"bäks, *noun*. Date: 1688
: a box or receptacle containing loose sand: as
a: a shaker for sprinkling sand on wet ink **b:** a box that contains sand for children to play in

- A restricted area where code can play in
- Allow users to download and execute untrusted applications with limited risk
- Restrictions can be placed on what an application is allowed to do in its sandbox
- Untrusted applications can execute in a trusted environment

Jails & containers are a form of sandboxing
… but we want to focus on giving users the ability to run apps

Application sandboxing

via system call hooking & user-level validation

System Call Interposition

System calls interface with system resources

- An application must use system calls to access any resources, initiate attacks … and cause any damage
  - Modify/access files/devices:

    creat, open, read, write, unlink, chown, chgrp, chmod, …
  - Access the network:

    socket, bind, connect, send, recv

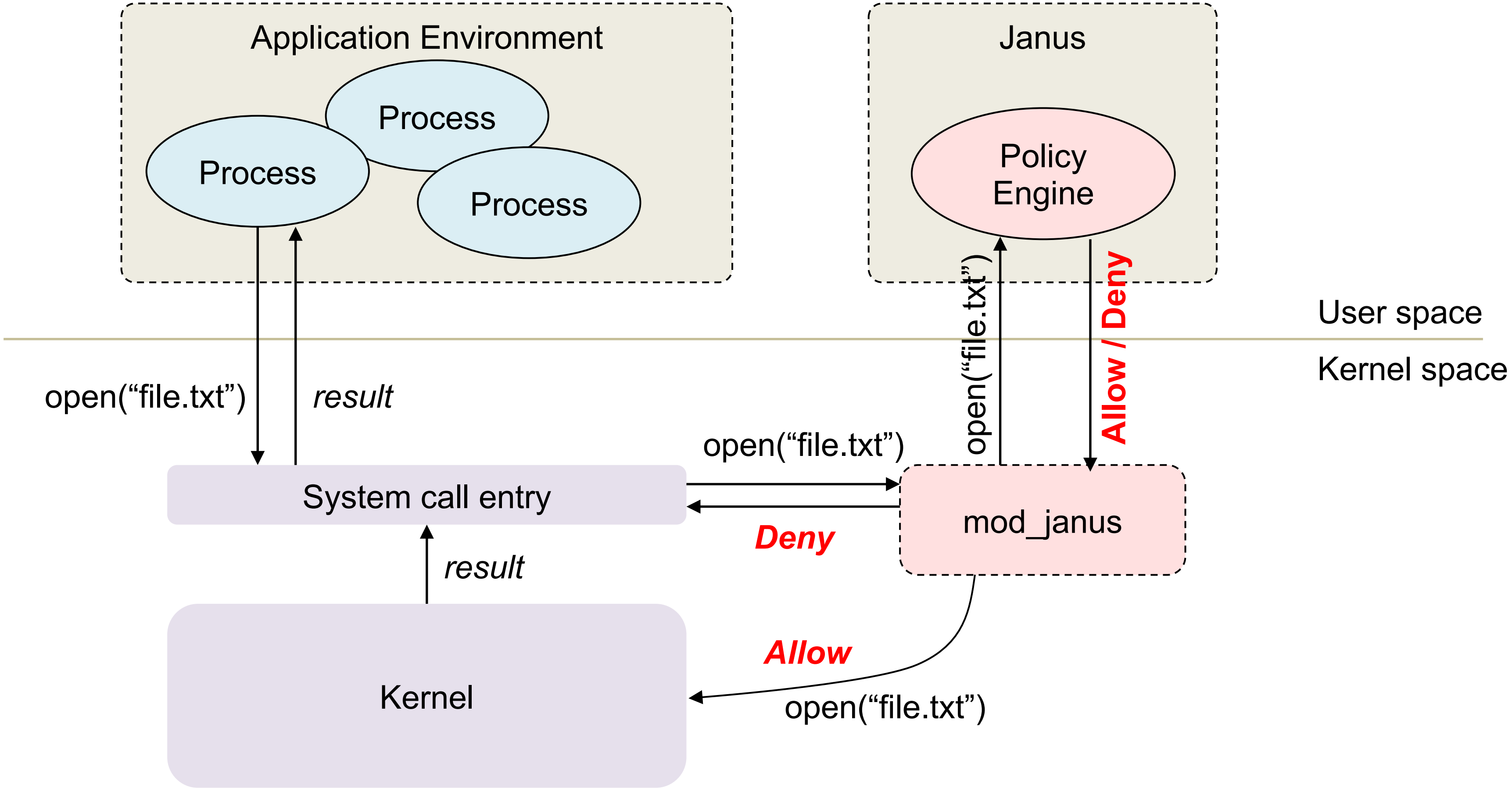
- **System call interposition (hooking)**
  - Intercept, inspect, and approve an app's system calls

Example: Janus

- **Policy file** defines allowable files and network operations
- Dedicated policy per process
  - Policy engine reads policy file
  - Forks
  - Child process *execs* application
  - All accesses to resources are screened by Janus
- OS system call entry point contains a *hooks*
  - Redirects control to `mod_Janus`
  - Module tells the user-level Janus process that a system call has been requested
    - Process is blocked
    - Janus process queries the module for details about the call
    - Makes a policy decision

Example: Janus

App sandboxing tool implemented as a loadable kernel module



Implementation Challenge

Janus has to mirror the state of the operating system!

- If process forks, the Janus monitor must fork
- Keep track of the network protocol
  - socket, bind, connect, read/write, shutdown
- Does not know if certain operations failed
- Gets tricky if file descriptors are duplicated
- Remember filename parsing?
  - We have to figure out the whole dot-dot (..) thing!
  - Have to keep track of changes to the current directory too
- App namespace can change if the process does a *chroot*
- What if file descriptors are passed via Unix domain sockets?
  - *sendmsg, recvmsg*
- Race conditions: **TOCTTOU**

Application sandboxing

via integrated OS support

Linux seccomp-BPF

- Linux capabilities
  - Dealt with things a root user could do
  - No ability to restrict access to regular files

- Linux namespaces
  - Chroot functionality – no ability to be selective about files

Seccomp-BPF =
SECure COMPuting with Berkeley Packet Filters

- Allows the user to attach a system call filter to a process and all its descendants
  - Enumerate allowable system calls
  - Allow/disallow access to specific files & network protocols

- Used extensively in Android

Linux seccomp-BPF

- Uses the Berkeley Packet Filter (BPF) interpreter
  - seccomp sends “packets” that represent system calls to BPF
- BPF allows us to define rules to inspect each request and take an action
  - *Kill the task*
  - *Disallow & send SIGSYS*
  - *Return an error*
  - *Allow*
- Turned on via the `prctl()` – process control – system call

Seccomp is not a complete sandbox but is a tool for building sandboxes

- Needs to work with other components
  - Namespaces, capabilities, control groups
- Potential for comprehension problems – BPF is very low level

Apple Sandbox

- Create a list of rules that is consulted to see if an operation is permitted

- Components:
  - Set of libraries for initializing/configuring policies per process
  - Server for kernel logging
  - Kernel extension using the TrustedBSD API for enforcing individual policies
  - Kernel support extension providing regular expression matching for policy enforcement

- sandbox-exec command & sandbox_init function
  - sandbox-exec: calls *sandbox_init()* before *fork()* and *exec()*
  - `sandbox_init(kSBXProfileNoWrite, SANDBOX_NAMED, errbuf);`

Apple sandbox setup & operation

sandbox_init:

- Convert human-readable policies into a binary format for the kernel
- Policies passed to the kernel to the TrustedBSD subsystem
- TrustedBSD subsystem passes rules to the kernel extension
- Kernel extension installs sandbox profile rules for the current process

Operation: intercept system calls

- System calls hooked by the TrustedBSD layer will pass through **`Sandbox.kext`** for policy enforcement
- The extension will consult the list of rules for the current process
- Some rules require pattern matching (e.g., filename pattern)

Apple sandbox policies

Some pre-written profiles:

- Prohibit TCP/IP networking
- Prohibit all networking
- Prohibit file system writes
- Restrict writes to specific locations (e.g., /var/tmp)
- Perform only computation: minimal OS services

Browser-based application sandboxing

Web plug-ins

- External binaries that add capabilities to a browser
- Loaded when content for them is embedded in a page
- Examples: Adobe Flash, Adobe Reader, Java

Challenge:

How do you keep plugins from doing bad things?

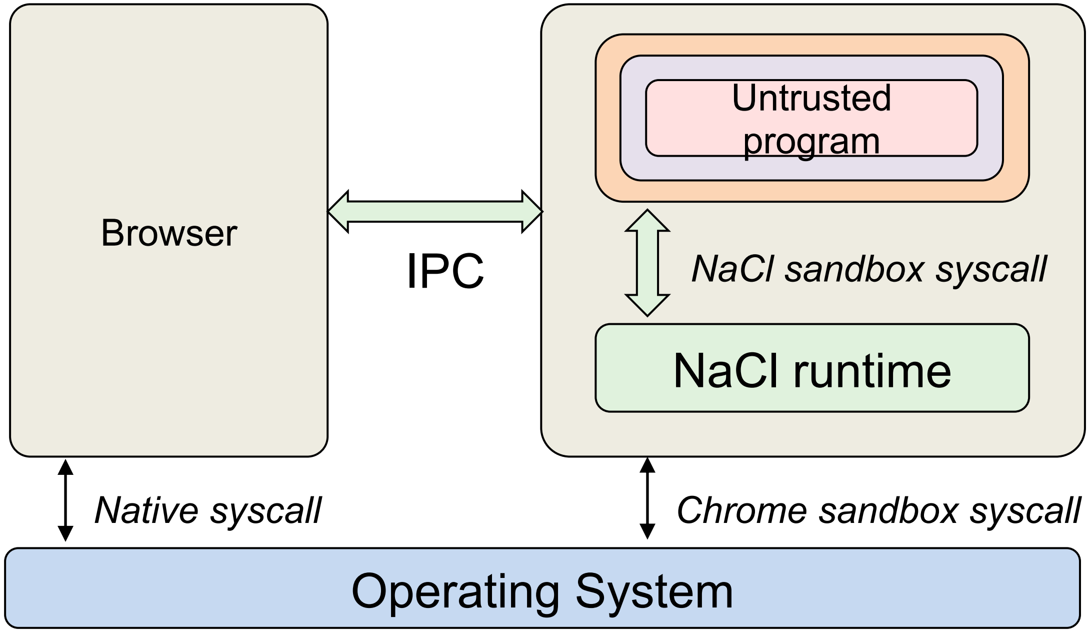
Chromium Native Client (NaCl)

- Browser plug-in designed for
  - Safe execution of platform-independent untrusted native code in a browser
  - Compute-intensive applications
  - Interactive applications that use resources of a client
- Two types of code: **trusted & untrusted**
  - **Trusted** code does not run in a sandbox
  - **Untrusted** code has to run in a sandbox
- Untrusted native code
  - Built using NaCl SDK or any compiler that follows alignment rules and instruction restrictions
    - GNU-based toolchain, custom versions of gcc/binutils/gdb, libraries
    - Support for ARM 32-bit, x86-32, x86-64, MIPS32
    - Pepper Plugin API (PPAPI): portability for 2D/3D graphics & audio
  - NaCl statically verifies the code to check for use of privileged instructions

Chromium Native Client (NaCl)

Two sandboxes

- Outer sandbox: restricts capabilities using system call interposition
- Inner sandbox: uses x86 segmentation to isolate memory among apps
  - Uses static analysis to detect security defects in code; disallow self-modifying code



Portability

- Portable Native Client (PNaCl)
  - Architecture independent
  - Developers compile code once to run on any website & architecture
  - Compiled to a *portable executable* (**pexe**) file
  - Chrome translates pexe into native code prior to exectution

Java sandbox

Java Language

- Type-safe & easy to use
  - Memory management and range checking
- Designed for an interpreted environment: JVM
- No direct access to system calls

Java Sandbox

1. **Bytecode verifier**: verifies Java bytecode before it is run
    - Disallow pointer arithmetic
    - Automatic garbage collection
    - Array bounds checking
    - Null reference checking
2. **Class loader**: determines if an object is allowed to add classes
    - Ensures key parts of the runtime environment are not overwritten
    - Runtime data areas (stacks, bytecodes, heap) are randomly laid out
3. **Security manager**: enforces *protection domain*
    - Defines the boundaries of the sandbox (file, net, native, etc. access)
    - Consulted before any access to a resource is allowed

JVM Security

- Complex process
- 20+ years of bugs … hope the big ones have been found!
- Buffer overflows found in the C support library
  - C support library buggy in general
- Generally, the JVM is considered insecure
  - But Java in general is pretty secure
    - Array bounds checking, memory management
    - Security manager with access controls
  - Use of native methods allows you to bypass security checks

The end