Sandboxing

CS-576 Systems Security
Instructor: Georgios Portokalidis
Fall 2018
Sandboxing Means Isolation

Why?

Software has bugs
Defenses slip
Untrusted code

Compartmentalization limits interference and damage!

“a sandbox is a security mechanism for separating running programs”
-- wikipedia
Opportunities for Sandboxing: Browsers

- Flash plugin
- JS engine
- Untrusted inputs
- .swf
- .js

Browser

Internet
Opportunities for Sandboxing: Browsers

- Flash plugin
- JS engine
- Untrusted inputs
- .swf
- .js
- Sandboxes
- Browser
- Internet
Untrusted Code in Browsers

Browser

Internet

Flash plugin

JS engine

Rendering engine

Untrusted inputs

.html

.css
Untrusted Code in Browsers

Browser

Flash plugin
JS engine
Rendering engine

Sandbox?

Untrusted inputs

.html
.css

Internet
Sandboxing Methods

VM-based
- Run entire OS in isolation

OS-based
- Process-wide
- Available system calls and capabilities are restricted

Language-based
- Language isolates components

Inline reference monitor
- Integrated into untrusted code during compilation, code generation, or through emulation
- Security checks injected to enforce policy
Sandboxing Methods

VM-based
- Run entire OS in isolation

OS-based
- Process-wide
- Available system calls and capabilities are restricted

Language-based
- Language isolates components

Inline reference monitor
- Integrated into untrusted code during compilation, code generation, or through emulation
- Security checks injected to enforce policy
Let's Refresh What We Know About OSes
OS Access Control of HW
OS Access Control of HW

User space has restricted access to CPU and memory.

User space cannot access HW devices.

- Application
- Application
- Application

Trusted Kernel

- CPU
- MEMORY
- HW DEVICES
OS Access Control of HW

User land

Kernel

Hardware

Application

Application

Application

Trusted Kernel

CPU

MEMORY

HW DEVICES
OS Access Control of HW

User land

Kernel

Hardware

Application

Application

Application

Trusted Kernel

CPU

MEMORY

HW DEVICES

Unprivileged

Privileged
Example of OS-Level Access Control to HW

- Allow "Whisper" to access the following?
  - Notifications
  - Location

- User Account Control helps stop unauthorized changes to your computer.

- "Google Maps" Would Like to Use Your Current Location
  - Don’t Allow
  - OK
Process-level Isolation

Processes cannot directly access each other’s state

User land

Kernel

Trusted Kernel
Process-level Isolation

The kernel can setup inter-process communication

User land

IPC setup

Kernel

Trusted Kernel
Process-level Isolation

The kernel can setup inter-process communication

User land

IPC setup

Kernel

Trusted Kernel
Process-level Isolation

Same for processes owned by different users

- USER1 Application
- IPC channel
- USER2 Application
- IPC channel
- USER3 Application

User land

IPC setup

Kernel

Trusted Kernel
Hardware-based Enforcement

The memory-management unit (MMU) provides virtual memory

Execution rings separate user and kernel space
- Indicated by bits in CPU status register

Processes are isolated into different virtual memory address spaces
Sandboxing Methods

VM-based
- Run entire OS in isolation

OS-based
- Process-wide
- Available system calls and capabilities are restricted

Language-based
- Language isolates components

Inline reference monitor
- Integrated into untrusted code during compilation, code generation, or through emulation
- Security checks injected to enforce policy
Building on Process Isolation

Run code in its own process space to isolate it from browser process

Congratulations you have just executed untrusted code from the Internet!
Building on Process Isolation

Container must have limited privileges
Chromium Sandboxing in Linux

Chromium runs plugins and the rendering engine for each tab in a separate process.

Rendering processes are sandboxed.

Sandboxed processes are managed by a broker process over IPC.

https://chromium.googlesource.com/chromium/src/+/master/docs/linux_sandboxing.md
A helper binary with the setuid bit set is used

The SUID bit causes the execution of the process as root
- Enables access to privileged kernel APIs, such as namespaces

chroot() is used to change the process’ root directory
- Take away file system access from the process

Process is placed in new PID namespace
- Process cannot terminate or signal processes outside the namespace

Process is placed in new network namespace
- Restrict network access of process

Finally drop super-user privileges
Process Sandbox: User Namespaces

User namespaces are an unprivileged API

Used as an alternative to SUID sandbox

A process is placed a new namespace

Isolates:
- Filesystem
- Network
- PID
- IPC
User Namespaces

A newly launched process can be put in a new namespace

- Through the clone() system call

Available namespaces

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Constant</th>
<th>Isolates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cgroup</td>
<td>CLONE_NEWCGROUP</td>
<td>Cgroup root directory</td>
</tr>
<tr>
<td>IPC</td>
<td>CLONE_NEWIPC</td>
<td>System V IPC, POSIX message queues</td>
</tr>
<tr>
<td>Network</td>
<td>CLONE_NEWNET</td>
<td>Network devices, stacks, ports, etc.</td>
</tr>
<tr>
<td>Mount</td>
<td>CLONE_NEWNS</td>
<td>Mount points</td>
</tr>
<tr>
<td>PID</td>
<td>CLONE_NEWPID</td>
<td>Process IDs</td>
</tr>
<tr>
<td>User</td>
<td>CLONE_NEWUSER</td>
<td>User and group IDs</td>
</tr>
<tr>
<td>UTS</td>
<td>CLONE_NEWUTS</td>
<td>Hostname and NIS domain name</td>
</tr>
</tbody>
</table>

Reading material: [https://lwn.net/Articles/531114/](https://lwn.net/Articles/531114/)
Process Sandbox: SECCOMP BPF

Filters the kernel APIs available to a process

Used together with previous sandboxes

Aims to protect the kernel from a malicious process

Available system calls are defined using Berkeley packet filters

- Filters are compiled to a program that enforces policy
Programs consist of instructions that can check the values of various system calls and their arguments

- Cannot dereference pointers

BPF can be hard to write and looks like assembly

Example:

- a BPF load operation (BPF_LD), for a word (BPF_W), using the value in the instruction as an offset into the data area (BPF_ABS)
  
  BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, arch)))

- a jump-if-equal instruction (BPF_JMP | BPF_JEQ) that compares the value in the instruction, which is known as "k", (BPF_K) to the value in the accumulator. So, if the architecture is x86-64, this jump will skip the next instruction (the offset of "1" for the jump true destination), otherwise it will execute it ("0" for jump false)

  BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K , AUDIT_ARCH_X86_64 , 1, 0)
SECCOMP BPF Programs

What you can do
Filter specific system calls
  • Or particular arguments to them
Define what action to take when an invalid call is made

What you can’t do
Dereference pointer arguments
Remove an installed filter
Libsecomp

Offers a simpler API for writing filters

```c
int seccomp_rule_add(uint32_t action, int syscall, unsigned int arg_cnt, ...);

seccomp_rule_add(SCMP_ACT_ALLOW, SCMP_SYS(close), 0);
```
#include <stdio.h>  /* printf */
#include <unistd.h>  /* dup2: just for test */
#include <seccomp.h> /* libseccomp */

int main() {
    printf("step 1: unrestricted\n");

    // Init the filter
    seccomp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_KILL);  // default action: kill

    // setup basic whitelist
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(rt_sigreturn), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);

    // setup our rule
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(dup2), 2,
                     SCMP_A0(SCMP_CMP_EQ, 1),
                     SCMP_A1(SCMP_CMP_EQ, 2));

    // build and load the filter
    seccomp_load(ctx);
    printf("step 2: only 'write' and dup2(1, 2) syscalls\n");

    // Redirect stderr to stdout
    dup2(1, 2);
    printf("step 3: stderr redirected to stdout\n");

    // Duplicate stderr to arbitrary fd
    dup2(2, 42);
    printf("step 4: !! YOU SHOULD NOT SEE ME !!\n");

    // Success (well, not so in this case...)
    return 0;
}
Limitations of OS and VM-based Sandboxing

Context switches between broker and sandboxed processes can be expensive
Sandboxing Methods

VM-based
- Run entire OS in isolation

OS-based
- Process-wide
- Available system calls and capabilities are restricted

Language-based
- Language isolates components

Inline reference monitor
- Integrated into untrusted code during compilation, code generation, or through emulation
- Security checks injected to enforce policy
Example: JS/Java

The language and the runtime environment/VM is enforcing security

- Memory safe languages
- Memory corruption or leakage is not possible (at least in theory)

Access control done at the API level, for example:

- Which files can be loaded
- Which frames are accessible through the DOM
- Where can code be loaded from
- The VM acts as a reference monitor
Sandboxing Methods

VM-based
- Run entire OS in isolation

OS-based
- Process-wide
- Available system calls and capabilities are restricted

Language-based
- Language isolates components

Inline reference monitor
- Integrated into untrusted code during compilation, code generation, or through emulation
- Security checks injected to enforce policy
Sandboxing Unsafe Languages

Pointers can be used to potential read/write arbitrary memory

Memory accesses need to be isolated first
- Can rarely rely on HW to contain memory operations
- Software checks are introduced in application code
Software-fault Isolation

Run multiple programs in the same address space that run in isolation

Each program runs in a different logical fault domain

Programs can access memory within their domain
  - Ensures memory secrecy and integrity

Code within a domain cannot call/jump to code in other domains
  - Unless through secure interfaces
Software-fault Isolation

Programs can only access memory within their domain

- Ensures memory secrecy and integrity
Software-fault Isolation

Programs can only access memory within their domain
- Ensures memory secrecy and integrity

Code within a domain cannot call/jump to code in other domains
- Unless through secure interfaces
Software-fault Isolation

Programs can only access memory within their domain
  ▪ Ensures memory secrecy and integrity

Code within a domain cannot call/jump to code in other domains
  ▪ Unless through secure interfaces

Modify programs during compilation or by rewriting to enforce these properties
Constraining Memory Accesses

Through boundary checking

\[
\text{cmp } 0x0300 \\
\text{if less Error} \\
\text{cmp } 0x04AA \\
\text{if greater Error} \\
\text{write } x
\]
Constraining Memory Accesses

We can improve the boundary checks

- By allocating domains in aligned memory ranges
- Using bit masking to help with checking

```
tmp := x & FF00
cmp tmp 0300
if not equal Error
write x
```
Constraining Memory Accesses

Further improvements
- Do not detect error
- Constrain memory access to domain

```plaintext
tmp := x & 00FF
tmp := tmp | 0300
write tmp
```
Eliminating temporary registers is not always a good idea

Malicious code could bypass masking operations

```
...  
x := x & 00FF  
x := x | 0300  
write x
```

Time Of Check
Time Of Use (TOCTOU)
Constraining Memory Accesses

Can malicious code bypass checks with temporary registers?

```plaintext
tmp := x & 00FF

tmp := tmp | 0300

write tmp
```
Can malicious code bypass checks with temporary registers?

tmp := x & 00FF

tmp := tmp | 0300

write tmp

• tmp has not been initialized and will probably cause the program to crash.
• Can be forced to crash by setting tmp to bad address (e.g., 0xffffffff) after write
Constraining Control Flow

Sandboxes are mainly used to constrain untrusted code so obviously this is a general problem.

...  
jmp ptr  →  ?
Constraining Control Flow

Similar tricks can be applied

... jmp ptr

... call ptr

... ret

... tptr := ptr & 00FF
... tptr := tptr | 0300
... jmp tptr

... tptr := ptr & 00FF
... tptr := tptr | 0300
... call tptr

?
Constraining Control Flow

Naive approach

\[
\begin{align*}
\text{ret} & \quad \rightarrow \\
pop \ tptr & \\
tptr & := tptr \& \ 00FF \\
tptr & := tptr \mid 0300 \\
jmp \ ptr & 
\end{align*}
\]
CISC Trouble

Constraining within the domain is not enough

- Instructions may be hidden within instructions in CISC programs
Pseudo Fixed-size Instructions

Align every “pseudo” instruction on a 32-byte boundary

- 0x1F bits are always zero

Force pointer so it can only point to a pseudo instruction

```c
pop tptr
tptr := tptr & 00E0
tptr := tptr | 0300
jmp ptr
```
Benefits of SFI

No context switches

Faster if run-time checks are faster than context switching
Google Native Client (NaCL)

A sandboxing technology for running a subset of Intel x86, ARM, or MIPS native code in a sandbox

https://developer.chrome.com/native-client

NaCL programs are compiled with modified compiler

Supports subset of language

Produce sandboxed programs
Escaping Sandboxes

Exploitation of a sandboxed component grants limited control

But sandboxes may have bugs

Multiple exploits in different components are usually required

In 2012’s pwnium competition 14 bugs where needed to take down chrome

- http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html
Multiple Layers of Sandboxes

Original process

Sandboxed component

Sandboxed process
Other Use Cases for Isolation

Process-level Isolation from the OS is frequently used to realize the principle of least privilege in servers

Examples: SSH, Web servers
SSH

Connections

SSH listening process

Runs as root

Authenticate

SSH request serving process

fork()

SSH request serving process

SSH request serving process

How is access control done here?
SSH

- Connections
- Runs as root
- SSH listening process
- Authenticate
- setuid()/seteuid()
- SSH request serving process
- fork()
- SSH request serving process
- SSH request serving process
- Process drop privileges and run as the authenticated user