Practical and Effective Sandboxing for Non-root users

Taesoo Kim and Nickolai Zeldovich

MIT CSAIL
Why yet another sandbox for desktop applications?

- There are many existing sandbox mechanisms
  - Chroot / Lxc (Unix/Linux)
  - Jail (Freebsd)
  - Seatbelt (Mac OS X)
  - VM?
  ...
- Difficult-to-use, requiring root privilege, or slow!
Our tool

TL;DR

$ mbox -- ./downloaded-bin

Network Summary:
> [11279] -> 173.194.43.51:80
> [11279] Create socket(PF_INET,...)
> [11279] -> a00::2607:f8b0:4006:803:0

Sandbox Root:
> /tmp/sandbox-11275
> N:/tmp/index.html

TL;DR

$ mbox -- ./downloaded-bin

... Network Summary:
> [11279] -> 173.194.43.51:80
> [11279] Create socket(PF_INET,...)
> [11279] -> a00::2607:f8b0:4006:803:0

... Sandbox Root:
> /tmp/sandbox-11275
> N:/tmp/index.html

TL;DR

```bash
$ mbox -- ./downloaded-bin
...
```

Network Summary:
- [11279] -> 173.194.43.51:80
- [11279] Create socket(PF_INET,...)
- [11279] -> a00::2607:f8b0:4006:803:0

Sandbox Root:
- /tmp/sandbox-11275
- N:/tmp/index.html

Commit, diff, ignore, list, shell, quit ?>

Protecting the host filesystem from modification
TL;DR

$ mbox -- ./downloaded-bin

...

Network Summary:
> [11279] -> 173.194.43.51:80
> [11279] Create socket(PF_INET,...)
> [11279] -> a00::2607:f8b0:4006:803:0

...

Sandbox Root:
> /tmp/sandbox-11275
> N:/tmp/index.html

[commit, [diff, [ignore, [list, [shell, [quit ?>

Revision-control-system like interface
Without root privilege! TL;DR

```
$ mbox -- ./downloaded-bin
...
Network Summary:
> [11279] -> 173.194.43.51:80
> [11279] Create socket(PF_INET,...)
> [11279] -> a00::2607:f8b0:4006:803:0
...
Sandbox Root:
> /tmp/sandbox-11275
> N:/tmp/index.html
```
Design overview

• **Layered sandbox filesystem**
  - Overlaying the host filesystem
  - Confining modification made by sandboxed processes
  - Persistent storage: in fact, just a regular directory

• **System call interposition**
  - Commodity OSes provide one for non-root users
  - Enabling a variety of applications: installing pkgs, restricting network, build/dev. env ...
Design overview

• **Layered sandbox filesystem**
  - Overlaying the host filesystem
  - Confining modification made by sandboxed processes
  - Persistent storage: in fact, just a regular directory

• **System call interposition**
  - Commodity OSes provide one for non-root users
  - Enabling a variety of applications: installing pkgs, restricting network, build/dev. env ...
Installing packages as normal user

$ mbox -R -- apt-get install git
(-R: emulate a fakeroot environment)

• Mbox provides a writable sandbox layer on top of the host filesystem
  - User owns the sandbox directory
  - Contain newly installed files, and package databases

• Mbox emulates a fakeroot environment
  - Use standard package managers without modification
  - Support: apt-get (Ubuntu), dpkg (Debian), pip (Python)
Running unknown binary safely

$ mbox -n -- ./downloaded-bin
(-n: disable remote network accesses)

- Mbox protects the host filesystem from modifications
- Mbox restricts or monitors network accesses
  - Interpret socket-like system calls
  - Summarize network activity when terminated
Checkpointing filesystem

$ mbox -i -- emacs ~/.emacs
(-i: enable interactive commit-mode)
Checkpointing filesystem

$ mbox -i -- emacs ~/.emacs
(-i: enable interactive commit-mode)
Checkpointing filesystem

$ mbox -i -- emacs ~/.emacs

(-i: enable interactive commit-mode)
Checkpointing filesystem

$ mbox -i -- emacs ~/.emacs
(-i: enable interactive commit-mode)
Build/development environment

$ tree linux-git
...
+-mm--mmmap.c
 +-mlock.c

$ mbox -r outdir -- make
(-r dir: specify a sandbox directory)

- Mbox can separate out the generated obj files
  - make clean == rm -rf outdir
- Mbox can also be used for virtual dev. env.
  - Install packages with standard package managers
Outline

- Motivation / use cases
  - Layered sandbox filesystem
  - System call interposition (using seccomp/BPF)
- Implementation / evaluation
- Related work
- Summary
Sandbox filesystem supports copy-on-write
Sandbox filesystem supports copy-on-write

Sandboxed process

open(".emacs", R)

Read

Sandbox filesystem

Host filesystem

.emacs
Sandbox filesystem supports copy-on-write

Sandboxed process

open(".emacs", RW)
Sandbox filesystem supports copy-on-write

Sandboxed process

open(".emacs", RW)

Sandbox filesystem

.emacs

Copy

Host filesystem

.emacs
Sandbox filesystem supports copy-on-write

Sandboxed process

open(".emacs", RW)

Read/Write

Copy

Sandbox filesystem

.emacs

Host filesystem

.emacs
Copy-on-write by rewriting path arguments

Sandboxed process

```
open(".emacs", RW)
```

Read/Write

Sandbox filesystem

```
.emacs
```

Copy

Host filesystem

```
.emacs
```

,tmp/sbox/
Copy-on-write by rewriting path arguments

Sandboxed process

open(".emacs", RW)

Read/Write

/tmpp/sbox/home/taesoo/.emacs

Copy

/tmp/sbox/

Sandbox filesystem

.emacs

Copy

Host filesystem

.emacs
All subsequent read/write should happen on the sandbox filesystem

Sandboxed process

open(".emacs", RW)
...
open(".emacs", R)

Read

Sandbox filesystem

.emacs

Host filesystem

.emacs
All subsequent read/write should happen on the sandbox filesystem.

Sandboxed process

open(".emacs", RW)
...
open(".emacs", R)

/tmp/sbox/home/taesoo/.emacs

/tmp/sbox/

Sandbox filesystem

.emacs

Host filesystem

.emacs
Sandbox filesystem keeps track of deleted files

Sandboxed process

`unlink(".emacs")`

...
Sandbox filesystem keeps track of deleted files

Sandboxed process

/unlink(".emacs")

... 

/open(".emacs", R)

Hashmap of deleted files

/.emacs

Mbox

Hashmap of deleted files

/.emacs
Mbox doesn't have to interpose on every system call

```c
fd = open(".emacs", R)    fd = open(".emacs", RW)
read(fd, buf, size)       write(fd, buf, size)
```

- After redirecting the path in open(), we don't have to interpose on read/write() system calls
- Mbox needs to interpose on 48 system calls getting a path argument to provide a layered sandbox filesystem
Mechanism: system call interposition

• Ptrace is a common technique, but slow
  – Interpose entry/exit of every system call
  – Serialize system calls of child processes

• Using seccomp/BPF (>= Linux 3.5)
  – Seccomp is a security mechanism for isolating a process by allowing a certain set of system calls
  – Seccomp/BPF uses BPF (Berkeley Packet Filter) to specify rules for filtering system calls
BPF program for interposition

User space

Kernel

Mbox
BPF program for interposition

User space

Kernel

Seccomp/BPF

BPF_STMT(LD, OFF_SYSCALL)
BPF_JUMP(#open, 0, 1)
BPF_STMT(RET, TRACE)
...
BPF_STMT(RET, ALLOWED)

BPF

Mbox

① prctl()
BPF program for interposition

```
BPF_STMT(LD, OFF_SYSCALL)
BPF_JUMP(#open, 0, 1)
BPF_STMT(RET, TRACE)
...
BPF_STMT(RET, ALLOWED)
```
BPF program for interposition

```c
BPF_STMT(LD, OFF_SYSCALL)
BPF_JUMP(#open, 0, 1)
BPF_STMT(RET, TRACE)
...
BPF_STMT(RET, ALLOWED)
```
BPF program for interposition

```
open("/a", RW)
exec()
```

```
BPF_STMT(LD, OFF_SYSCALL)
BPF_JUMP(#open, 0, 1)
BPF_STMT(RET, TRACE)
...  
BPF_STMT(RET, ALLOWED)
```
BPF program for interposition

Sandboxed process

User space

Kernel

Seccomp/BPF

Mbox

BPF_STMT(LD, OFF_SYSCALL)
BPF_JUMP(#open, 0, 1)
BPF_STMT(RET, TRACE)
...  
BPF_STMT(RET, ALLOWED)

BPF_STMT(RET, ALLOWED)

prctl()  
①  
exec()  
②  
wait()  
③  
④  
EVENT_SECCOMP
BPF program for interposition

```
open("/a", RW)
[382x423]"/a" → "'/tmp/sbox/a"
exec()
ptrace (PEEK/POKE)
```

```
BPF_STMT(LD, OFF_SYSCALL)
BPF_JUMP(#open, 0, 1)
BPF_STMT(RET, TRACE)
...
BPF_STMT(RET, ALLOWED)
```
More story to come ...

- How to avoid time-of-check-to-time-of-use?
- How to avoid replicating OS state?
- ...

Please, check the paper!
Implementation

- **Mbox:** a prototype for Linux (>= 3.5, x86-64)
  - Using seccomp/BPF and ptrace
  - Extending strace 4.7
  - 1,500 Lines of code
  - Distributions: Ubuntu 12.04 and Arch 64bit
Performance evaluation

- How much overhead does Mbox exhibit?
- How much faster is seccomp/BPF than ptrace?
Benchmark

- Following the benchmark from Apiary
- Run each benchmark in three configurations
  - Normal
  - Mbox with ptrace
  - Mbox with seccomp/BPF
Mbox imposes modest end-to-end performance overhead

<table>
<thead>
<tr>
<th>Task</th>
<th>Normal</th>
<th>Mbox Seccomp/BPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octave</td>
<td>2.1s</td>
<td>2.1s</td>
</tr>
<tr>
<td>Zip</td>
<td>15.6s</td>
<td>17.4s</td>
</tr>
<tr>
<td>Untar</td>
<td>13.6s</td>
<td>16.4s</td>
</tr>
<tr>
<td>Build Linux (-j1)</td>
<td>43.6s</td>
<td>49.7s</td>
</tr>
</tbody>
</table>

- 0.1% ~ 20.9% overhead
- Octave: a computation-heavy workload
  - Exhibits negligible performance overhead (0.1%)
  - Spends 98% of its execution in userspace
Seccomp/BPF reduces the interposition overhead

<table>
<thead>
<tr>
<th>Task</th>
<th>Normal</th>
<th>Mbox</th>
<th>Ptrace</th>
<th>Seccomp/BPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octave</td>
<td>2.1s</td>
<td>2.1s</td>
<td>0.1%</td>
<td>2.1s</td>
</tr>
<tr>
<td>Zip</td>
<td>15.6s</td>
<td>21.2s</td>
<td>36.5%</td>
<td>17.4s</td>
</tr>
<tr>
<td>Untar</td>
<td>13.6s</td>
<td>19.0s</td>
<td>40.3%</td>
<td>16.4s</td>
</tr>
<tr>
<td>Build Linux (-j1)</td>
<td>43.6s</td>
<td>53.2s</td>
<td>21.9%</td>
<td>49.7s</td>
</tr>
</tbody>
</table>

- Compare overheads of using ptrace and seccomp/BPF
- Seccomp/BPF reduces overhead up to 24.5%
Seccomp/BPF has better concurrency than ptrace

<table>
<thead>
<tr>
<th>Task</th>
<th>Normal</th>
<th>Mbox</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Ptrace</td>
</tr>
<tr>
<td>Build Linux (-j1)</td>
<td>43.6s</td>
<td>53.2s</td>
</tr>
<tr>
<td>Build Linux (-j4)</td>
<td>21.7s</td>
<td>45.6s</td>
</tr>
</tbody>
</table>

- When compiling the Linux kernel with 4 parallel jobs, performance improves 64.9% compared to ptrace
- By avoiding unnecessary serialization of system calls, multiple processes execute system calls concurrently
Seccomp/BPF has better concurrency than ptrace

<table>
<thead>
<tr>
<th>Task</th>
<th>Normal</th>
<th>Mbox</th>
<th>Ptrace</th>
<th>Seccomp/BPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Linux (-j1)</td>
<td>43.6s</td>
<td>53.2s</td>
<td>21.9%</td>
<td>49.7s</td>
</tr>
<tr>
<td>Build Linux (-j4)</td>
<td>21.7s</td>
<td>45.6s</td>
<td>110.1%</td>
<td>31.5s</td>
</tr>
</tbody>
</table>

- When compiling the Linux kernel with 4 parallel jobs, performance improves 64.9% compared to ptrace
- By avoiding unnecessary serialization of system calls, multiple processes execute system calls concurrently
Related work

- **Layered filesystems**: UnionFS [Quigley '06] / Aufs
  - Following unification rules / copy-on-write
    → Require no modifications in commodity OSes

- **System call interposition**: Ostia [Garfinkel '04]
  - Enforcing security policies / studied common pitfalls
    → Summarize our experience of using seccomp/BPF

- **Namespace**: Plan9 [Pike '90] / Lxc container (Docker)
  - Private namespace for each process
    → Enabling various applications via system call interposition
Summary

**Mbox:** a lightweight sandboxing mechanism

- *Layered* sandbox filesystem
- Revision-control-system like sandbox usage model
- Interposing on system calls with *seccomp/BPF*
- Enabling a variety of applications for *non-root* users

http://pdos.csail.mit.edu/mbox
Questions (if you don't have any)

- What if files are modified by other processes running outside of Mbox?
- Why 20% on tar? just rewriting path arguments doesn't seem to be demanding work.
- How complicated the BPF program? Why not implement everything in BPF then?
- Why does Mbox support only 64bit? and is Mbox ready for users (not developers)?
- Can Mbox be used for A, B and C ... ?