Promises and pitfalls of sandboxes

“Multiple speed bumps don’t make a wall” (TT)

Robert Swiecki (expressing his own opinions here)
Confidence, Kraków 2017
But why?

- Known to be broken services containment (e.g. image converters)
- Hardening of services of a relatively good quality (e.g. ISC bind)
  - also for resource limitation
  - fuzzing
  - gcc as a service?
- Cloud: VPSes
- IaaS: Infrastructure as a Service
- SaaS: Sandbox as a Service (e.g. hiring pipelines for coders)
- Capture The Flag (CTF) competitions
- Malware research
- Reverse Engineering
- ...
Orthogonality/Layering #1

- Layers of defense

- KVM or CPU/Hardware
- OS/Userland isolation (with CPU/MMU help)
- OS/FS/PID permissions/capabilities
- NS / SW-Sandbox
- Runtime
- Payload
Orthogonality/Layering #2

- Buggy CPU or GFX driver
- OS/Userland isolation (with CPU/MMU help)
- OS/FS/PID permissions/capabilities
- Runtime
- NS / SW-Sandbox
- Payload
from qemu unpriv account to host kernel ring0 - don't use AMD's newest ucode 0x06000832 for Piledriver-based CPUs - goo.gl/L1us8g
Runtime hardening

- ASLR/PIE/NX-stack/CFI/Stack-protector/Fortify-Source
  - Good: Typical CPU/mem penalty <5%
  - Bad: By-passable with memory leaks

- ASAN/MSAN/UBSAN
  - Good: Truly effective at finding security problems
  - Bad: Not security features, can even compromise security

`ASAN_OPTIONS='verbosity=2:log_path=foo' ./setuid`
Legacy mechanisms (rlimits, cgroups)

- **RLimits**: Quite basic
  - Can limit VM size of a process, number of open file-descriptors, and a few more things
  - Per-process only, with the exception of RLIMIT_NPROC

- **Cgroups**: Nicer
  - Per-process, but cumulative resource use and inheritable
  - Confusing design (via multiple /sys files)
Legacy mechanisms (chroot) #1

- Popular during 90’s
  - Good: Easy concept to understand
  - Bad: Only for root (root-equivalent capability), by-passable

```c
mkdir("abc", 0755);
chroot("abc");
chdir("././././././././././././.");
```

(also: namespaces - CLONE_NEWUSER|CLONE_NEWNS)
Legacy mechanisms (chroot) #2

- Doesn’t compartmentalize other aspects of the OS

1. `ptrace(PTRACE_ATTACH, <pid_outside_chroot>, 0, 0);`

2. `process_vm_writev(<pid_outside_chroot>);`

3. `socket(AF_UNIX),
   connect(abstract_socket_namespace_to_a_broker)`
Legacy mechanisms (chroot) #3

- Reduces kernel attack surface minimally only (incl. /dev)

- The **FUTEX** test

**Linux Kernel Futex Local Privilege Escalation (CVE-2014-3153)**

*The futex_requeue function in kernel/futex.c in the Linux kernel through 3.14.5 does not ensure that calls have two different futex addresses, which allows local users to gain privileges via a crafted FUTEX_REQUEUE command that facilitates unsafe waiter modification.*
Legacy mechanisms (capabilities)

- Interesting idea (power-less root)
- Not really used (with exceptions, like ‘ping’)
  - Messy list of capabilities (>60) - require good understanding of interactions within Linux
- Many capabilities are root-equivalent
- Not for regular users (for root only)

```
$ man 7 capabilities
  CAP_SYS_CHROOT
    Use chroot(2)

$ ln /bin/su /tmp/chroot/su
$ chroot /tmp/chroot
$ /su
```
SW/CPU Emulators

- Good: probably no good sides of SW/CPU emulators
- Bad:
  - Slow (faster with JIT)
  - Enormous attack surface: CPU and HW
  - Additional services: Printing interfaces, Network NAT/Bridges
- Truly bad history of security vulnerabilities:
  - Venom CVE-2015-3456
  - Kostya Kortchinsky’s printer service flaw VMSA-2015-0004
  - Bugs in VGA, ETH, USB emulation ...
Ptrace #1

- Debugging interface, not a security one
- Good: Surprisingly effective (starting with systrace by N.Provos)
- Bad:
  - slow -> context switches
  - full of security bugs itself
  - messy, inconsistent behavior between different kernel versions

```c
pid: syscall(syscall_no, arg0, arg1, ...)
ptracer: ptrace(PTRACE_SYSCALL, pid, 0, 0);
another process/thread: kill(pid, SIGKILL)
```
Ptrace #2

Ptracer

```c
bool is_entry;

for (;;) {
    int pid = wait(&status);
    ...
    if (WIFSTOPPED(status) &&
        WSTOPSIG(status) == SIGTRAP) {
        is_entry = !is_entry;
        if (is_entry) {
            check_syscall();
        }
    }
}
```

Tracee

```c
int main() {
    syscall1();
    asm("int3");
    syscall2();
}
```

rt_sigreturn changes orig_eax to -1

Since Linux 2.4.6

PTRACE_O_TRACETRACE(syscall1())
PTTRACE #3

Solution: R/O Maps??
1. Modify `fork/vfork -> clone(CLONE_TRACE)`
2. `PTRACE_O_TRACEFORK, PTRACE_O_TRACEVFORK, PTRACE_O_TRAC ECLONE` (v. 2.5)

... unless `clone(CLONE_UNTRACED)` is used -> remove the flag, or invoke the syscall violation procedure
• If ptracer dies -> no more sandboxing
• Since v.3.8 -> \texttt{PTRACE\_O\_EXITKILL}

• Multitude of other problems
  ○ Unclear SIGSTOP semantics (thread stop, thread group stop)
  ○ Spurious SIGTRAP events
  ○ Emulation of process stop state (\texttt{PTRACE\_LISTEN})
  ○ ...
Different syscall tables (e.g. i386 vs x86-64)

- No easy way to differentiate between 32/64-bit syscall tables from `ptrace()`
  - return value from `ptrace(PTRACE_GETREGSET)` returns info about bitness of the process bitness, and not about the syscall table used
  - it’s possible to fetch syscall-inducing instruction (int 0x80 vs syscall vs sysenter) but TOCTOU.
  - Checking the CS segment register might be inconclusive
Native Client (NaCL) #1

- Based on the Russ Cox’ and Bryan Ford’s idea from **vx32**
- User-level sandboxing, makes use of custom ELF loader/verifier and CPU segmentation (**modify_1dt()** on i386) and large mappings (non i386)
Native Client (NaCL) #2

- Limited subset of x86-32, x86-64 and ARM
- SFI - Software Fault Isolation, DFI/CFI - Data/Control Flow Integrity
- naclcall, nacljmp, naclret
- Possible to change CFI (func ptrs), but not to escape the jail

```
nacljmp eax        ->     and eax,0xfffffffffe0
                       jmp eax

nacljmp %eXX,%rZP   ->     and $-32,%eXX
                         add %rZP,%rXX
                         jmp *%rXX
```
Native Client (NaCL) #3

- **Good**
  - Quite effective & rather fast (5-10% slow-down)
  - Based on CPU instruction whitelists
  - Statically pre-verified
  - Ability to apply an external syscall sandbox (e.g. ptrace or seccomp-bpf based)

- **Bad**
  - Writing safe trusted stubs (trampolines) requires great deal of work and attention
  - The whole process is not very straightforward (custom compilers/SDK/gdb)
  - Depends on perfect implementation of white-listed CPU instructions (CPU errata)
  - Lots of restrictions
    - No dynamic/self-modifying/JIT code
    - No assembler inlines
    - No direct access to syscalls/FS/Net
Native Client (NaCL) #4

KVM

Namespace container (Net, Pid, Fs)

Linux process

NaCL jail

Syscall Sandbox

Trusted stub / Runtime Service

CPU?
Seccomp (v1) #1

- Neat idea, but turned out to be immensely hard to work with
- Required brokers for resources, but nothing can be done for memory management
- Chromium Legacy Seccomp Sandbox
  - One of the most complex implementations out there
Seccomp (v1) #2

One-process Seccomp-v1 Sandbox

Process (thread group)

Thread #1
Payload
Seccomp mode 1

Thread #2
Resource broker
No seccomp

IPC (mem)
Resource brokering

*Resources are File-Descriptors* *(with exceptions)*
ptrace/seccomp-bpf (but not seccomp v1)

![Diagram](image)
There were a few ideas about pushing syscall evaluators into kernel before (e.g. in the perf's subsystem - ftrace).

Authors came up with two ideas:

- Reusing BPF - Berkeley Packet Filter(s) VM
- Letting the userland to create the full evaluator operating on a simple struct

```c
struct seccomp_data {
    int nr;
    __u32 arch; /* NO PID and TID!!! */
    __u64 instruction_pointer;
    __u64 args[6];
};
```
Seccomp-bpf #2

SECCOMP_RET_KILL    /* kill the task immediately */
SECCOMP_RET_TRAP    /* disallow and force a SIGSYS */
SECCOMP_RET_ERRNO   /* returns an errno */
SECCOMP_RET_TRACE   /* pass to a tracer or disallow */
SECCOMP_RET_ALLOW   /* allow */

- **SECCOMP_RET_TRACE** - no tracer → syscall disallowed

- If multiple filters - all evaluated, and the “worst” return value wins

- No loops!
Seccomp-bpf #3

struct sock_filter {
    uint16_t code;  /* the opcode */
    uint8_t jt;     /* if true: jump displacement */
    uint8_t jf;     /* if false: jump displacement */
    uint32_t k;     /* immediate operand */
};

/* load the syscall number */
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, offsetof(struct seccomp_data, nr)),
/* allow read() */
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_read, 0, 1),
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW)
/* deny anything else */
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL)
VALIDATE_ARCHITECTURE,

LOAD_SYSCALL_NR,
SYSCALL(__NR_exit, ALLOW),
SYSCALL(__NR_exit_group, ALLOW),
SYSCALL(__NR_write, JUMP(&l, write_fd)),
SYSCALL(__NR_read, JUMP(&l, read)),
DENY,

LABEL(&l, read),
ARG(0),
...

JNE(STDIN_FILENO, DENY),
ARG(1),
JNE(buf, DENY),
ARG(2),
JGE(sizeof(buf), DENY),
ALLOW,

LABEL(&l, write_fd),
ARG(0),
JEQ(STDOUT_FILENO, JUMP(&l, w_buf)),
JEQ(STDERR_FILENO, JUMP(&l, w_buf)),
DENY,
Kafel (config language)

```c
#define mysyscall -1
POLICY sample {
    ALLOW {
        kill(pid, sig) {
            pid == 1 && sig == SIGKILL
        }
        mysyscall(arg1, myarg2) {
            arg1 == 42 &&
            myarg2 != 42
        }
    }
}
USE sample DEFAULT KILL
```

Chromium BPF-DSL (C++ API)

```c
EvaluateSyscall(int sysno) const OVERRIDE {
    if (sysno == __NR_socketpair) {
        const Arg<int> domain(0), type(1)
        return If(domain == AF_UNIX &&
                   (type == SOCK_STREAM ||
                    type == SOCK_DGRAM), Error(EPERM)).
                   Else(Error(EINVAL));
    }
    return Allow();
}
```
Seccomp-bpf #6

- Implementers tend to forget to check the (syscall) architecture in use

```c
struct sock_filter filter[] = {
    VALIDATE_ARCHITECTURE,
}
```

- Seccomp-bpf cannot check user-land arguments (FS paths, connect())
  - Use ptrace() or namespaces

```c
syscall(__NR_open, "/etc/passwd", O_RDONLY);
```

- Decompiled seccomp-bpf code is *rather unreadable* (for verification)
- Syscalls vary between architectures (no “one policy for all”), OpenSSH
Namespaces #1

- Concept borrowed from Plan9 (*from outer space*)
- Some aspects of the OS can be unshared from other processes
  - Uids, Hostname, Fs tree, Net context, Pid tree, Cgroups...
- Since ~3.16 it’s possible, with `CLONE_NEWUSER`, to unshare context for an unprivileged user
  - This enable huge attack surface, many priv-esc’s in the past
    - Access to raw sockets for various protocols
    - Ability to mount some filesystems (bugs in overlayfs)
    - Chroot escape trick?
    - Quite complex semantics wrt clone flag exclusion (e.g. no `CLONE_THREAD | CLONE_NEWNS`)
  - Can be disabled with kernel patches
Namespaces #2

clone(CLONE_NEWUSER | CLONE_THREAD | CLONE_NEWNS)

chroot("/sth")

execve("/sbin/su")

Process
Chroot = /

O-UID = 1337
O-UID = 1337
I-UID = 0
I-UID = 0

Chroot = /
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth

O-Uid = 0
Chroot = /sth
Namespaces #3

- It shrinks the kernel attack surface *(the futex problem)* minimally only
- It expands this attack surface in some other places
  - Can be avoided by careful setup of namespaces
    - i. Enable namespaces
    - ii. Setup chroot, hostname, net etc.
    - iii. Drop capabilities
    - iv. Somehow block `CLONE_NEWUSER` (can be by `chrooting`)
    - v. Run sandboxed process
  - `firejail`, `nsjail`, `minijail0`, `docker/lxc`
Namespaces + Syscall whitelist + resource limits

- Eg: NS + Seccomp-bpf + Cgroups

OS: Eth, Full filesystem root tree, all PIDs, all syscalls, whole memory

Payload

NS container
- Some eth
- Some FS tree
- Some PIDs

Syscall whitelist

Resource Limits
- memory
- # of pids
KVM

- Direct access to a subset of CPU instructions
  - Many still need to be emulated (attack surface!!)
- If devices or services (printing servers) are simulated (some can be exposed directly via IOMMU) → attack surface!!
Others: Xen, Capsicum, LSM

- **Xen**
  - Creation of domains: privileged (Dom\textregistered) and unprivileged (Dom\textsuperscript{U})
  - Personal opinion: usage declining bc of KVM in Linux
  - Problems: attack surface - non trivial IO API exposed by the Dom\textsuperscript{0}

- **Capsicum**
  - Working motto: “Practical capabilities for UNIX”
  - Resources as file-descriptors
  - Linux implementation: LSM + Seccomp-bpf

- **LSM**
  - Yama, AppArmor, SELinux
  - Typically try to limit access to resources (e.g. filesystem paths)
  - Protection of the kernel attack surface doesn’t seem to be priority (the futex problem)
## The futex test

<table>
<thead>
<tr>
<th>Technology</th>
<th>Futex test</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlimits, cgroups, chroot, capabilities</td>
<td>FAILS</td>
</tr>
<tr>
<td>ptrace syscall whitelist</td>
<td>PASSES</td>
</tr>
<tr>
<td>seccomp</td>
<td>PASSES</td>
</tr>
<tr>
<td>seccomp-bpf</td>
<td>PASSES</td>
</tr>
<tr>
<td>NaCL</td>
<td>PASSES</td>
</tr>
<tr>
<td>LSM</td>
<td>FAILS</td>
</tr>
<tr>
<td>Capsicum</td>
<td>FAILS</td>
</tr>
<tr>
<td>KVM / SW Emulators</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Conclusions

- Many features shouldn’t be called sandboxes these days
  - chroot, rlimits, capabilities
- Attack surface is what matters
- Not every protection/hardening method is a layer (or, a strong layer)
- There’s no golden bullet: practically all sandboxing Linux kernel facilities or external projects suffer from non trivial flaws, or hard to overcome practical problems (e.g. NaCL)
- Combination of a few of those features (if these are solving independent problems) might actually produce something useful (effective)
- Creating safe and functional sandboxes for Linux is a truly non-trivial job, where corner-cases are common
Q&A