SANDBOXING A LINUX APPLICATION

MARTIN ERTSÅS
Sandboxing a Linux application

Martin Ertsås
martiert@gmail.com
April 7, 2022
Whoami

Martin Ertsås
martiert@gmail.com
@martiert

C++ Programmer
Software Developer
Training Consultant
Engineering Manager

Cisco Systems Norway

Linux based

He/Him

$
Important

Never run code from these slides!

I take no responsibility if you do!
Important

Never run code from these slides!
Important

Never run code from these slides!

I take no responsibility if you do!
What is a sandbox?

“A mechanism to run applications in a controlled and restricted environment, with the goal of mitigating the impact of vulnerabilities” — Martin Ertsås
What is a sandbox?

“A mechanism to run applications in a controlled and restricted environment, with the goal of mitigating the impact of vulnerabilities”
What is a sandbox?

“A mechanism to run applications in a controlled and restricted environment, with the goal of mitigating the impact of vulnerabilities”

— Martin Ertsås
Why Sandbox?

- Untrusted applications
- Running downloaded code
- Application expectations of environment
- They are fun!
Why Sandbox?

- Untrusted applications
Why Sandbox?

• Untrusted applications
• Running downloaded code
Why Sandbox?

- Untrusted applications
- Running downloaded code
- Application expectations of environment
Why Sandbox?

- Untrusted applications
- Running downloaded code
- Application expectations of environment
- They are fun!
Tools available

• Namespaces
• Seccomp
• Cgroups
• ++
Alternative

```c
int main(int argc, char ** argv)
{
    setup_sandbox();
    execvp(argv[1], &(argv[1]));
}
```
Alternative

```c
int main(int argc, char **argv)
{
    setup_sandbox();
    run_application_code();
}
```
static main_func actual_main = nullptr;

int my_main(args) {
    return actual_main(args);
}

int __libc_start_main(main_func main,
                      args) {
    actual_main = main;
    auto actual_start_main = dlsym("__libc_start_main");
    return actual_start_main(
                           my_main,
                           args);
}
Namespaces

- Wraps a global system resources to provide isolation
- Several different Namespace types in Linux
- Some Hierarchical, some not
- Created by calling either clone or unshare
User Namespace

- Isolates users and groups available
- `CLONE_NEWUSER`
- Hierarchical
User Namespace

- NS 0:
  - UID 1000
  - GID 1000

- NS 1:
  - UID 412
  - GID 1000

- NS 3:
  - UID 0
  - GID 0

- NS 2:
  - UID 0
  - GID 0
User Namespace

```c
int my_main(int argc,
            char ** argv,
            char ** argenv)
{
    unshare(CLONE_NEWUSER);
    return actual_main(argc,
                        argv,
                        argenv);
}
```
User Namespace

```c
+uid_t uid = geteuid();
+uid_t gid = getegid();
+
    unshare(CLONE_NEWUSER);
+ofstream ufs("/proc/self/uid_map");
+ufs << 0 << ' ' << uid << ' ' << 1;
+
    ofstream gfs("/proc/self/gid_map");
+gfs << 0 << ' ' << gid << ' ' << 1;
+
    return actual_main(argc,
    argv,
    argenv);
```
User Namespace

```cpp
ofstream ufs("/proc/self/uid_map");
ufs << 0 << ' ' << uid << ' ' << 1;

ofstream deny("/proc/self/setgroups");
+deny << "deny";
+
ofstream gfs("/proc/self/gid_map");
gfs << 0 << ' ' << gid << ' ' << 1;
```
Mount Namespace

- Isolates list of mount points
- `CLONE_NEWNS`
- Can share view of subtrees with the parent process
Mount Namespace

```c
uid_t uid = geteuid();
uid_t gid = getegid();
-unshare(CLONE_NEWUSER);
+unshare(CLONE_NEWUSER | CLONE_NEWNS);
+
set_uid_gid_mappings();
```
Mount Namespace

uid_t gid = getegid();

unshare(CLONE_NEWUSER | CLONE_NEWNS);
+mount(NULL, "/", NULL,
+    MS_PRIVATE | MS_REC, NULL);
+  set_uid_gid_mappings();
Mount Namespace

set_uid_gid_mappings();

+mount("tmpfs", "/tmp", "tmpfs", 0, NULL);
+
+fs::create_directories("/tmp/lib64");
+mount("/lib64", "/tmp/lib64", NULL,
+        MS_REC | MS_BIND, NULL);
+
+fs::create_directories("/tmp/etc");
+mount("/etc", "/tmp/etc", NULL,
+        MS_REC | MS_BIND, NULL);

return actual_main(argc,
Mount Namespace

```c
fs::create_directory("/tmp/etc");
mount("/etc", "/tmp/etc", NULL,
     MS_REC | MS_BIND, NULL);

+fs::create_directory("/tmp/oldroot");
+pivot_root("/tmp", "/tmp/oldroot");
+chdir("/");
+umount2("/oldroot", MS_DETACH);
+fs::remove("/oldroot");
+
return actual_main(argc,
```
DEMO!!!!
PID Namespace

- Isolates process ID number space
- `CLONE_NEWPID`
- First process in the namespace gets PID 1
- unshare does not move the process into the namespace
- Hierarchical
PID Namespace
PID Namespace

```c
uid_t uid = geteuid();
uid_t gid = getegid();

-unshare(CLONE_NEWUSER | CLONE_NEWNS);
+unshare(CLONE_NEWUSER
   + | CLONE_NEWNS
   + | CLONE_NEWPID);

set_uid_gid_mappings();
```
PID Namespace

mount_application(rootfs, argv[1]);

−swap_root();
−int result = actual_main(...);

+pid_t pid = fork();
+if (pid == 0) {
    + swap_root();
    + int result = actual_main(...);
    + _exit(result);
} +
+int status = −1;
+waitpid(pid, &status, 0);
+return status;
PID Namespace

```c
pid_t pid = fork();
if (pid == 0) {
    swap_root();
    +
    + fs::create_directories("/proc");
    + mount("proc", "/proc", "proc",
    +    0, NULL);
    +
    int result = actual_main(...);
    _exit(result);
```
Network Namespace

- Creates a new network stack
- CLONE_NEWNET
Network Namespace

```c
pid_t pid = fork();
if (pid == 0) {
    unshare(CLONE_NEWNET);
    swap_root();
}
```
Network Namespace

- This removes “all” network interfaces
- Use virtual network interfaces
- Use iptables
- Use bridge interfaces
Other namespaces

- Cgroup
- IPC
- Time
- UTS
Seccomp

- Filtering of system calls
- Only allows exit, sigreturn, read and write
Seccomp

- Filtering of system calls
- Only allows `exit`, `sigreturn`, `read` and `write`
- Pretty useless for most applications
Seccomp-BPF

- Uses the Berkeley Packet Filtering
- An in-kernel programming language
Raw usage

```c
struct seccomp_data {
    int nr;
    __u32 arch;
    __u64 instruction_pointer;
    __u64 args[6];
};
```
Raw usage

+sock_filter filter[] = {
+    BPF_STMT(
+        BPF_RET|BPF_K,
+        SECCOMP_RET_ALLOW),
+};
+sock_fprog prog = {
+    .len = std::size(filter),
+    .filter = filter,
+};
+prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
+syscall(SYS_seccomp,
+    SECCOMP_SET_MODE_FILTER,
+    0, &prog);
+
+    int result = actual_main(...);
+    _exit(result);
sock_filter filter[] = {
    BPF_STMT(
        BPF_LD|BPF_W|BPF_ABS,
        offsetof(seccomp_data, arch)
    ),
    BPF_JUMP(
        BPF_JMP|BPF_JEQ|BPF_K,
        AUDIT_ARCH_X86_64, 0, 1)
    ,
    BPF_STMT(
        BPF_RET|BPF_K,
        SECCOMP_RET_ALLOW
    ),
    BPF_STMT(
        BPF_RET|BPF_K,
        SECCOMP_RET_KILL
    ),
};
Raw usage

```
BPF_JUMP(
    BPF_JMP|BPF_JEQ|BPF_K,
    - AUDIT_ARCH_X86_64, 0, 1),
    + AUDIT_ARCH_X86_64, 0, 3),

+BPF_STMT(
    + BPF_LD|BPF_W|BPF_ABS,
    + offsetof(seccomp_data, nr)),
+BPF_JUMP(
    + BPF_JMP|BPF_JEQ|BPF_K,
    + SYS_execve, 0, 1),

BPF_RET(BPF_W|BPF_K, SECCOMP_RET_ALLOW)
BPF_RET(BPF_W|BPF_K, SECCOMP_RET_KILL)
```
BPF_STMT(BPF_LD|BPF_W|BPF_ABS, (offsetof(struct seccomp_data, arch))),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, AUDIT_ARCH_X86_64, 0, 24),

BPF_STMT(BPF_LD|BPF_W|BPF_ABS, (offsetof(struct seccomp_data, nr))),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_recvmsg, 22, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_sendto, 21, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_getsockname, 20, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_bind, 19, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_socket, 18, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_capget, 17, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_getdents64, 16, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_getegid, 15, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_geteuid, 14, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_getpid, 13, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_write, 12, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_munmap, 11, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_arch_prctl, 10, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_mprotect, 9, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_read, 8, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_close, 7, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_mmap, 6, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_fstat, 5, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_openat, 4, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_execve, 3, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_access, 2, 0),
BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, SYS_brk, 1, 1),

BPF_STMT(BPF_RET|BPF_K, SECCOMP_RET_KILL),
BPF_STMT(BPF_RET|BPF_K, SECCOMP_RET.Allow),
Raw usage

-BPF_JUMP(  
-   BPF_JMP|BPF_JEQ|BPF_K,  
-   SYS_exit_group, 22, 0),  
+BPF_JUMP(  
+   BPF_JMP|BPF_JEQ|BPF_K,  
+   SYS_exit_group, 0, 2),  
+BPF_STMT(  
+   BPF_LD|BPF_W|BPF_ABS,  
+   offsetof(struct seccomp_data, args)),  
+BPF_JUMP(  
+   BPF_JMP|BPF_JEQ|BPF_K,  
+   0, 22, 23),  
BPF_STMT(  
   BPF_JMP|BPF_JEQ|BPF_K,  
   SYS_sendto, 21, 0),
libseccomp usage

```c
set_new_root(rootfs);
+
+scmp_filter_ctx ctx =
+  seccomp_init(SCMP_ACT_KILL);
+seccomp_load(ctx);
+seccomp_release(ctx);

int result = actual_main(...);
return result;
```
libseccomp usage

```c
set_seccomp_arch(SCMP_ARCH_X86_64);
+ seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
+     SCMP_SYS(execve), 0);
+ seccomp_load(ctx);
seccomp_release(ctx);
```
libseccomp usage

```c
scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

allow_libseccomp(ctx, SCMP_SYS(execve));
allow_libseccomp(ctx, SCMP_SYS(exit_group));
allow_libseccomp(ctx, SCMP_SYS(recvmsg));
allow_libseccomp(ctx, SCMP_SYS(sendto));
allow_libseccomp(ctx, SCMP_SYS(getsockname));
allow_libseccomp(ctx, SCMP_SYS(getdents64));
allow_libseccomp(ctx, SCMP_SYS(geteuid));
allow_libseccomp(ctx, SCMP_SYS(getegid));
allow_libseccomp(ctx, SCMP_SYS(getpid));
allow_libseccomp(ctx, SCMP_SYS(write));
allow_libseccomp(ctx, SCMP_SYS(munmap));
allow_libseccomp(ctx, SCMP_SYS(arch_prctl));
allow_libseccomp(ctx, SCMP_SYS(mprotect));
allow_libseccomp(ctx, SCMP_SYS(read));
allow_libseccomp(ctx, SCMP_SYS(close));
allow_libseccomp(ctx, SCMP_SYS(mmap));
allow_libseccomp(ctx, SCMP_SYS(openat));
allow_libseccomp(ctx, SCMP_SYS(access));
allow_libseccomp(ctx, SCMP_SYS(brk));

seccomp_load(ctx);
seccomp_release(ctx);
```
libseccomp usage

```c
seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
                  SCMP_SYS(execve), 0);
seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
                  SCMP_SYS(exit_group), 0);
seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
                  SCMP_SYS(exit_group), 1,
                  SCMP_A0(SCMP_CMP_EQ, 0));
seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
                 SCMP_SYS(recvmsg), 0);
seccomp_rule_add(ctx, SCMP_ACT_ALLOW,
                 SCMP_SYS(reply), 0);
```

How do I compare strings?
How do I compare strings?

- You don’t
- seccomp runs before value copied to the kernel
- Could compare memory location
DEMO!!!!
CGroups

• Controls a group of processes and their access to resources
• Limiting capabilities
• Monitoring capabilities
• Pseudo-filesystem API
CGroups v1

- First cgroups implementation
- Multiple controllers
- No development synchronization between controllers
CGroups v1

- First cgroups implementation
- Multiple controllers
- No development synchronization between controllers
- Will ignore v1
Cgroups v2

- Take 2
- This time they had to get it right, right?
Cgroups v2

- Take 2
- This time they had to get it right, right?
- They have done a lot better
- One unified hierarchy
- Similar APIs for controllers
- Not all controllers available in v2
What can be controlled?

- Memory usage
- CPU usage
- CPU core access/pinning
- suspend/restore
- block device access
- Monitoring performance and cpu access
- Number of processes that might be created
- RDMA access
- huge pages usage
- Device creation
- Tagging network packets
- Prioritize network devices
Enabling/disabling controller

• Two files to look at
• `cgroups.controllers` lists available controllers in a cgroup
• `cgroups.subtree_control` lists controllers enabled in this cgroup
Enabling/disabling controller

- Two files to look at
  - cgroups.controllers lists available controllers in a cgroup
  - cgroups.subtree_control lists controllers enabled in this cgroup
- A controller is only available in a cgroup if it’s enabled in the parent cgroup
Demo!!!
Limiting memory
CPU
cpuset
Events
Cgroup type
Other tools

• Cgroups
• SELinux
Thank you!