Software sandboxing: The basics

Diving into the territory of software sandboxing is diving into mostly uncharted territory. The necessary pieces to implement good sandboxing in your software are scattered all-around and the pioneers haven't yet gathered enough knowledge into an unified *mappa mundi* that can guide new sailors through some well understood safe routes. In this blog post I'll offer my own share of experiences that I have acquired while working on sandboxing support for Emilua. Writing style will suffer a little because I'll err on the side of repeating myself too much to avoid any misunderstandings.

Do keep in mind that some Lua code samples here require the unreleased Emilua 0.11 (just grab a recent commit from the repo's development branch).

First, let's get some informal (but useful) definition for sandboxing just to make sure we're on the same page. Here's the definition that was used by Julien Tinnes and Chris Evans at Hack In The Box Malaysia 2009:

> The ability to restrict a process' privileges:
>
> - Programmatically;
> - Without *administrative authority* on the machine;
> - *Discretionary privilege dropping*.

That's a very good definition to keep the ball rolling. Let's quickly iterate over each point individually to make them crystal clear. However keep in mind that the opinions I possess today are a little different from the opinions J. Tinnes and C. Evans had during the 2009 talk (especially around "*is it okay to use superuser APIs*?"), so my explanations will differ a little and guide you towards what *I* consider better practices for 2025.

Programmatic privilege dropping

OSes present different interfaces to users and software developers. System administrators traditionally rely on filesystem permissions to isolate services (UNIX daemons). If we allowed third-party programs to freely change such permissions then it'd nullify the policies the sysadmin was trying to enforce to begin with.

Furthermore third-party programs abstract their own virtual worlds and most of the time UNIX filesystem permissions aren't a good fit to model the security policies such other virtual worlds require. Do you use UNIX permission modes to define who can see your Twitter feed or message you on Identi.ca? Filesystem permissions aren't the only knobs sysadmins possess to restrict access rights, but the reasoning developed here also apply to these other knobs.

Nonetheless a process inevitably runs on top of an OS and there are kernel-exposed resources the process interacts with (e.g. files). It's this interface that matters to the software developer. Web browsers such as Firefox run DRM plugins and it's desirable to run such third-party plugins without allowing them to have full access to every file that Firefox has access to (usually every file in the user's HOME directory). Traditional tools such as `setuidgid` can't help here and their usefulness is limited as interfaces sysadmins turn to. `setuidgid` and similar tools aren't interfaces intended for the software developer to use.

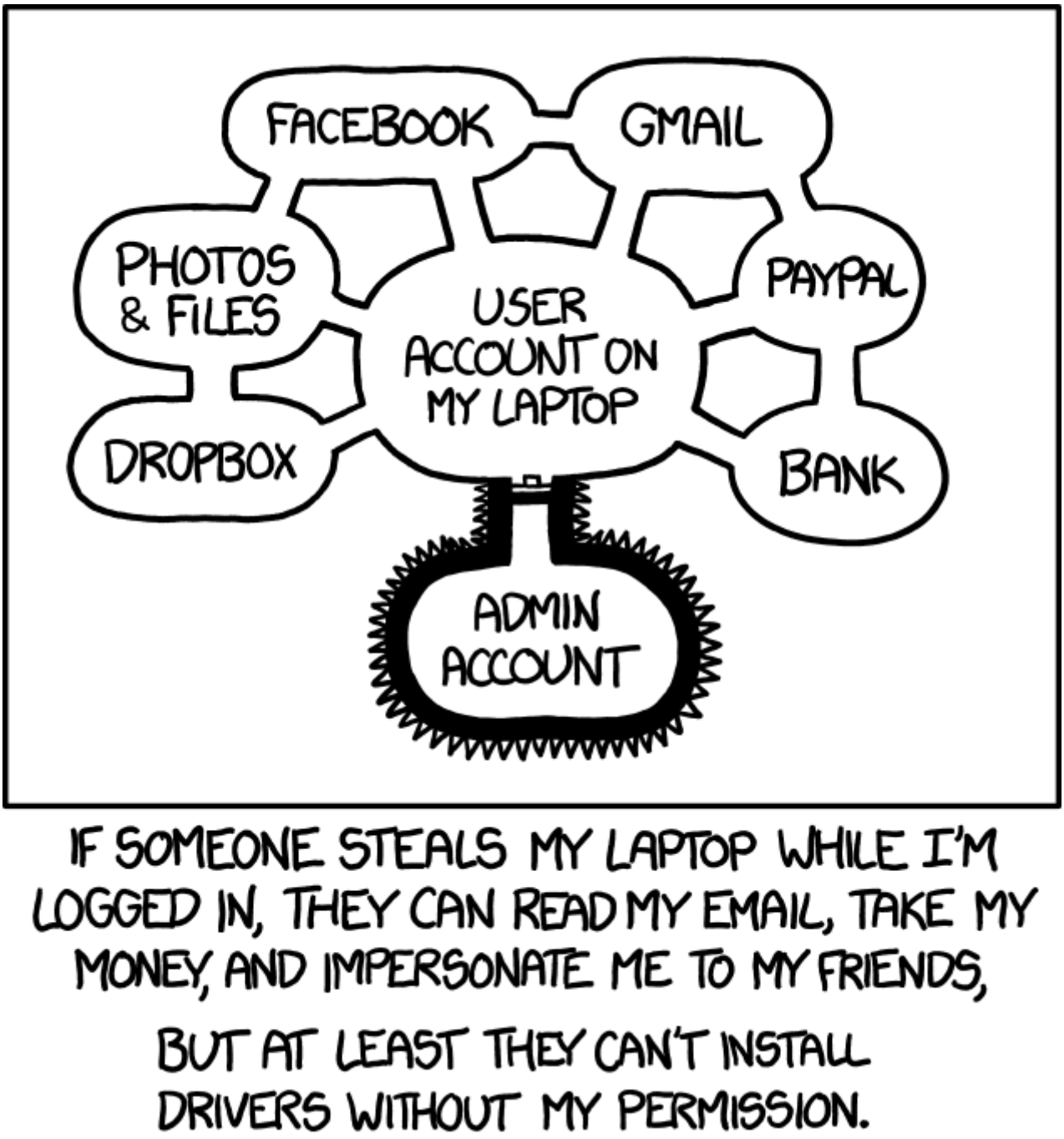

Figure 1. XKCD 1200: Authorization

For programmatic privilege dropping, traditional UNIX interfaces are a poor match, and OSes where this gap actually matters will provide extended interfaces that go beyond traditional UNIX (e.g. FreeBSD's Capsicum and Linux's Seccomp).

Dropping privileges without root

When good interfaces for sandboxing weren't available, programmers found their way to create sandboxes anyway by abusing mechanisms available only to the superuser. The most emblematic technique in this class is a helper suid binary that'll configure a chroot jail.

The obvious problem with these approaches is that they aren't available to all programs. Allowing any program to install suid binaries defeat any security measures. Suid binaries equal to temporally raising privileges to full administrative authority over the system. Privileges should only ever decrease, never increase (principle of least privilege).

Another related concern here is to not design APIs that backfire by exponentially increasing the kernel attack surface. The Docker boom popularized Linux namespaces as a mechanism to cheaply isolate services. However within a nested user namespace, the process runs as superuser (within that namespace), and code paths within the kernel that would normally only be available to the superuser are now available to every user. We have over a decade of kernel code that was never written with this premise in mind. This decision caused security problems in the past, and it's bound to happen again. To quote Andy Lutomirski:

> I consider the ability to use `CLONE_NEWUSER` to acquire `CAP_NET_ADMIN` over *any* network namespace and to thus access the network configuration API to be a huge risk. For example, unprivileged users can program iptables. I'll eat my hat if there are no privilege escalations in there.
>
> — Andy Lutomirski, https://lore.kernel.org/all/CALCETrWYRvqhyCwx5RX6L3TEYCfW0j6ThFUc+ASL7BpxgO5dEQ@mail.gmail.com/

It's fine to allow user namespaces as long as you restrict this interface to trusted containerization tools (e.g. Docker). However Linux namespaces is a terrible interface for software sandboxing. Newer sandboxing interfaces in Linux such as Landlock were carefully designed to not exponentially increase the kernel attack surface as to avoid the disasters we've seen with Linux's user namespaces. Moreover new ways to restrict namespaces within Linux are still being developed and long-term it's a bad bet to rely on them as a general sandboxing mechanism.

The first few years of software sandboxing research I've put into Emilua were solely focused on Linux namespaces. After a lot of frustration the focus shifted towards different solutions. Nowadays Emilua still offers support for Linux namespaces, but the intended use-case now is the creation of containerization tools. For proper sandboxing within Emilua, you'll use mechanisms other than Linux namespaces.

Discretionary privilege dropping

Actually sandboxes might also be defined as:

> A restricted, controlled execution environment that prevents potentially malicious software [...] from accessing any system resources except those

> for which the software is authorized.
>
> — Committee on National Security Systems (CNSS) Glossary 2022, https://www.cnss.gov/CNSS/issuances/Instructions.cfm

There's no actual consensus over what traits are required for some code to be considered sandboxed and definitions are usually very loose. These definitions don't require the properties we've been discussing so far. Therefore a different term altogether might come in handy. J. Tinnes suggested "*discretionary privilege dropping*". That's the type of sandboxing we'll be looking into for this article.

Discretionary privilege dropping doesn't replace system administration policies. Rather they complement each other and should be adopted in tandem.

Practical sandboxing: processes

Now we're hopefully on the same page. Sandbox for us mean the same thing: discretionary privilege dropping. How do we go from an unsandboxed program to a sandboxed one on existing real-world OSes? In every mainstream OS today, the privilege boundary lies at the process level. Credentials are associated with each process and that's what the kernel checks to decide whether the process can acquire new resources using ambient authority.

Linux is actually different and associates credentials at the thread level, but a design rooted at the thread level cannot work, and that's why glibc will do extra work to synchronize credentials across threads even if the kernel is sloppy about it. GNOME developers thought they could work at the thread level just to be proved wrong with CVE-2023-43641.

Adam Langley actually described a mechanism that in theory can work at the thread level, but in practice is economically too costly and I don't think it'll ever work:

> So that's what we do: each untrusted thread has a trusted helper thread running in the same process. This certainly presents a fairly hostile environment for the trusted code to run in. For one, it can only trust its CPU registers - all memory must be assumed to be hostile. Since C code will spill to the stack when needed and may pass arguments on the stack, all the code for the trusted thread has to carefully written in assembly.
>
> The trusted thread can receive requests to make system calls from the untrusted thread over a socket pair, validate the system call number and perform them on its behalf. We can stop the untrusted thread from breaking out by only using CPU registers and by refusing to let the untrusted code manipulate the VM in unsafe ways with `mmap`, `mprotect` etc.
>
> — https://www.imperialviolet.org/2009/08/26/seccomp.html

Let's not theorize over what alternative designs could work. For today, processes is what we got.

Once we compartmentalise our program as separate processes, we can proceed to the next steps:

- Assigning different privileges to each compartment (the processes).
- Handling communication among the compartments.

Researchers from FreeBSD's Capsicum already had the right mental model to develop sandboxes for well over a decade:

> Compartmentalised application development is, of necessity, distributed application development, with software components running in different processes and communicating via message passing.
>
> — Capsicum: practical capabilities for UNIX, Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway

The means to drop privileges are different on every platform, so we'll skip this for now and get back to it later. First let's focus on the problem of *distributed application development*.

The actor model and capability-based security

The actor model is one of the most well known patterns for the development of distributed systems. Erlang is perhaps its most iconic user. However Erlang's interest in the actor model lies in high availability and fault tolerance. Nonetheless it's still useful to look into widely used models even if we're not interested in high availability nor fault tolerance.

It's common for many explanations of the actor model to quickly step into the world of mathematics (which is fine). However many of them quickly become lost into the world of abstraction and forget about computers entirely (which is not fine). So let's just use a summary of the points we care about in the actor model:

- Actors can manage their own internal state.
- Actors can spawn other actors.
- Actors can send messages to other actors.
- Actors can include the addresses of other actors in messages.

If we summarize the actor model into concrete design choices within our programming language or framework, here's what we care about:

- There is a function to create actors. This function returns the address of the new actor.
- The address of an actor can be used to send messages.
- The address of an actor can also be a message or part of a larger message.
- There is a function to receive messages. This function read messages that are enqueued for the calling actor.

- It's possible to retrieve the address of the current actor.
- Actors share no memory with each other.
- An actor doesn't run in parallel to itself. If an actor is currently running in thread A, it can't also be running in thread B. However it's fine for actors to jump from one thread to another (as in work-stealing threaded task schedulers). That's the same property that Boost.Asio describe as strands.

For Emilua, this design translates into 3 functions:

```
spawn_vm(module) -> actor
actor.send(msg)
inbox.receive() -> msg
```

If you can learn just 3 functions, you can code for the actor model. Let's go over some examples now:

Creating actors

```
-- `module1.lua` will be the entry-point
-- for the execution of `new_actor1`
local new_actor1 = spawn_vm{
    module = 'module1' }

local new_actor2 = spawn_vm{
    module = 'module2' }
```

Sending messages

```
new_actor1:send(1)
new_actor1:send('ping')
new_actor1:send{ arg1 = 1, arg2 = 2 }
```

Receiving messages

```
local inbox = require 'inbox'

while true do
    local msg = inbox:receive()
    handle(msg)
end
```

Sending messages with addresses

```
new_actor1:send{ arg1 = 1, arg2 = 2, send_result_to = new_actor2 }
new_actor1:send{ arg1 = 3, arg2 = 4, send_result_to = inbox }
```

If we decide to use the actor model for sandboxing, then each process will be an actor. UNIX domain sockets can be used for actor messaging. Upon spawning a new actor, we setup socket inheritance so we can communicate with it. The socket will be the actor address. We also need to be able to include the addresses of other actors in messages, but this is also covered because it's possible to send file descriptors over UNIX domain sockets. The inbox file descriptor is never sent to other actors (i.e. we have a MPSC channel).

Emilua has many implementations for the actor model, so we must explicitly instruct it to use subprocesses upon spawning a new actor:

Creating IPC-based actors

```
local new_actor1 = spawn_vm{
    module = 'module1', subprocess = {} }

local new_actor2 = spawn_vm{
    module = 'module2', subprocess = {} }
```

This design also solves another problem for our sandboxing concerns: handing resources over to restricted processes. "*Everything is a file (descriptor)*" is one of the most well known phrases within the UNIX culture. If we can send file descriptors then we have a really broad range of resources that we can work with from sandboxed processes. To mention just a few:

- Files.
- Directories.
- Pipes.
- Sockets.
- Device nodes (e.g. `/dev/random`, GPU communication, ...).
- Shared memory (memfds).
- Process handles — pidfds, procdescs.
- Sync objects (e.g. eventfd).
- eBPF programs.

These are the resources we care about when we sandbox programs. These are the resources we'll take into account when we develop our security models. If we can prove that we aren't leaking file descriptors to the wrong actors then we can use the actor model. Fortunately there's a well researched model that solves this problem for us: capability-based security. There's even a programming language based on the actor model and capability-based security: the Pony programming language.

There's only one small gap that we need to fill to combine both models: capability based security assumes unforgeable tokens, but the actor model uses addresses (which are forgeable). In our case, this problem was already solved by the use of channels instead of addresses. The API stays the same and nobody will notice a thing. Now we can use capabilities to reason about questions such as:

- Is it possible for actor A to have effective access to resource X?

- How can we design a layout that makes it impossible for any sandboxed actor to simultaneously have access to files and sockets?

As for the usage of file descriptors as capabilities, the rule of thumb would be to avoid ioctls, but we'll be back to this topic later.

The actor model is simple to use, but very powerful. The ability to include the addresses of other actors in messages means arbitrarily variable topologies. On most of my own projects, I restrict myself to tree topologies, but the moment a tree becomes unfit for my project, it'll be easily replaced by a different topology. So far I haven't stumbled on a single sandboxed application that can't be modeled using actors.

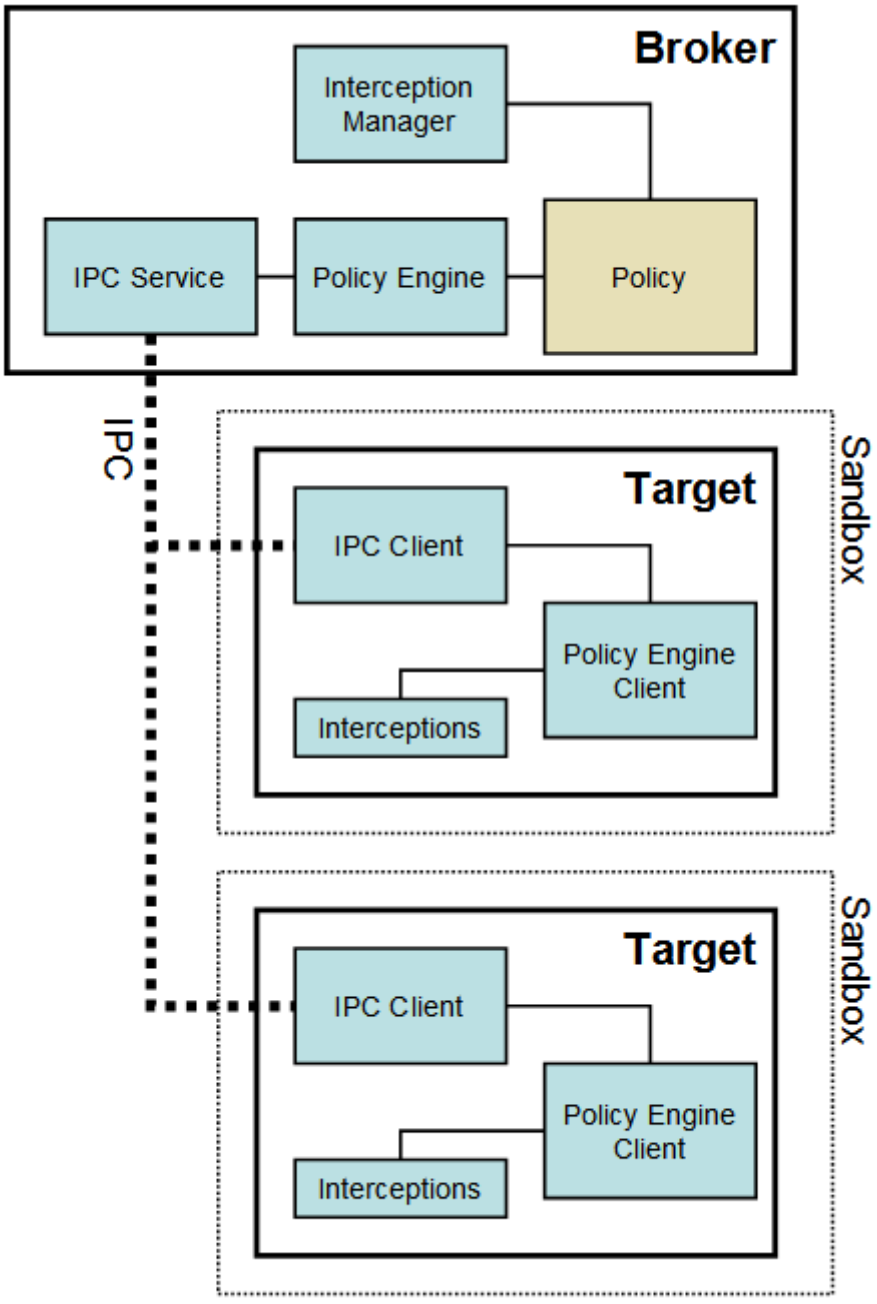

Figure 2. Chromium Sandbox Topology Diagram

If you need guidelines on how to develop distributed applications using the actor model, you'll enjoy several decades of R&D that've gone into it. Whether you prefer books, small tutorials, face-

to-face classes, study groups, or many other learning approaches, you'll likely find something of use.

File descriptors as capabilities

Now that we have messaging solved with the actor model, let's jump into sandboxing (security models) again. There are other properties an object must have so it can be modeled as a capability. A capability isn't only a reference to a resource, but the associated access rights as well. Owning a capability is the same as also having access rights to perform actions. With this in mind, we need to ponder:

- Can file descriptors be modeled as capabilities?
- What precautions must we take to use file descriptors as capabilities?

Generally UNIX systems run permission checks to grant or deny access only when a new file descriptor is created, not when existing file descriptors are used. This behavior is compatible with capabilities. Here's the code for a sample program:

```
#include <fcntl.h>
#include <stdio.h>

int main()
{
  int fd = open("/root", O_RDONLY);
  if (fd == -1) {
    perror("open");
  } else {
    printf("success\n");
  }
}
```

And the output when I run the program as root:

```
success
```

And the output when I run the program as any other user:

```
open: Permission denied
```

This is just the UNIX behavior I was mentioning. Now let's run some shell command as root:

```
# grep -Ee '^nobody:' </etc/shadow
nobody:!*:19642::::::
```

And the same command as a different user:

```
$ grep -Ee '^nobody:' </etc/shadow
-bash: /etc/shadow: Permission denied
```

Nothing surprising here. It's just the same behavior. Now let's run `grep` as an unprivileged user, but making sure it inherited a file descriptor opened by root:

```
# setpriv --reuid=1000 --regid=1000 --init-groups grep -Ee '^nobody:' </etc/shadow
nobody:!*:19642::::::
```

As stated earlier, UNIX systems generally don't run permission checks when performing actions on existing file descriptors. That's why `grep` succeeded to read the file contents in the example. This was true for the previous example (the action read on a regular file), but will it always be true? We could be afraid of new kernel versions. They could always introduce a new syscall that breaks this convention. However early in UNIX history the concept of suid binaries was introduced, and that's a legacy that'll keep haunting kernel developers to make sure they don't break this convention. Let's explore suid binaries now.

In the last example we had the superuser using the syscall `setresuid` to change the process credentials. Now we'll walk into the opposite direction, creating a child privileged process from an unprivileged one. This is allowed only for suid binaries, so the privileged process will only ever run programs trusted by the sysadmin. One of such programs is `su`:

```
$ setsid su </dev/null 2>&1 | cat
Password: su: Authentication token manipulation error
```

This example shows that we can easily trick suid binaries to read or write into any file descriptors that we have by using simple fd inheritance. For this example, it wrote the string "`Password: su: Authentication token manipulation error`" using the credentials of a privileged process. If the credentials of the writer process had any importance for the security of the system, every UNIX system would be broken already. Thefefore new interfaces are always designed in a way that the credentials of the writer process don't matter at all.

As a recent example to further stress the point, some of the last syscalls that Linux introduced were related to filesystem mounting. The initial versions of the contributed patchset were rejected due to the use of the syscall `write` in operations that'd use the credentials of the calling process for permission checks. Eventually the contributor changed the design by using the new syscall `fsconfig` and the patchset was accepted.

It's important to notice that kernel developers will respect the convention whether suid binaries are allowed on our Linux distro or not. Even if we block suid binaries from our OS entirely, we can still assume that no attacker will be able to gain new privileges by using our process as a proxy to perform some dangerous write operation (the attacker could just write into the file descriptor directly instead and the effects would've been the same).

The exception to this rule are ioctls. Performing ioctls on fds received from untrusted processes is always dangerous. Emilua relies on Boost.Asio for async IO, and Boost.Asio used to rely on `FIONBIO` when it shouldn't. After a few email exchanges, I managed to persuade Christopher Kohlhoff to change this behavior and now Boost.Asio will do the right thing as long as you're at least on Boost 1.86.

Awesome. We can indeed model file descriptors as capabilities, but we weren't the first to reach this conclusion.

FreeBSD's Capsicum

Capsicum is an interface to better support the use of file descriptors as capabilities that is part of FreeBSD since its 9.0 release. One of the facilities offered by Capsicum is the function `cap_enter`. `cap_enter` drops process privileges by disabling ambient authority entirely.

```
local new_actor3 = spawn_vm{
    module = 'module3',
    subprocess = {
        -- the runtime will run this Lua
        -- code before it attempts to
        -- initialize complex libraries
        -- or acquire system resources
        init = 'C.cap_enter()'
    } }
```

That's it. One function call and we dropped privileges. All system accesses will have to be performed through open file descriptors. If we don't already have access to some resource, the only way to get it now is through `inbox`. If we attempt to open files, `open` will fail because ambient authority is disabled. If we attempt to connect a socket to some endpoint, the operation will fail because ambient authority is disabled. That's the beauty of Capsicum: we deny access to external resources because the names themselves that could be used to refer to resources become unavailable.

When the Capsicum research was published, the following table was also presented:

Table 1. Sandboxing mechanisms employed by Chromium

Operating system	Model	Line count	Description
Windows	ACLs	22350	Windows ACLs and SIDs
Linux	`chroot`	605	`setuid` root helper sandboxes renderer
Mac OS X	Seatbelt	560	Path-based MAC sandbox
Linux	SELinux	200	Restricted sandbox type enforcement domain

Operating system	Model	Line count	Description
Linux	`seccomp`	11301	`seccomp` and userspace syscall wrapper
FreeBSD	Capsicum	100	Capsicum sandboxing using `cap_enter`

At that time, its researchers have modified Chromium to make use of Capsicum and compared how much effort was required to make use of each sandboxing mechanism within Chromium. Capsicum required only 100 lines of code. Compare that to seccomp's 11301 lines or Windows' 22350 lines. The other mechanisms compared didn't actually restrict the sandboxes significantly and can be disregarded. If you're only going to study one sandboxing mechanism in your life, it should be Capsicum. To this date, I have yet to see a better sandboxing mechanism than Capsicum.

Capsicum also provides finer grained access control to file descriptors. As an example, one may use Capsicum to allow one process to wait on a semaphore, but not to post on it. A file descriptor is usually created with all rights assigned. Then these rights can be reduced through the use of the function `cap_rights_limit`.

```
local unix = require 'unix'

local in_, out = unix.seqpacket.socket.pair()
out:shutdown('receive')
out = out:release() --< get file descriptor

-- deny shutdown-send so one
-- worker cannot shutdown the
-- channel to everyone
out:cap_rights_limit({'send'})

for i = 1, 20 do
    local worker = spawn_vm{
        module = 'worker',
        subprocess = {
            init = 'C.cap_enter()'} }
    worker:send(out)
end
out:close()

local buf = byte_span.new(512)
while true do
    local nread = in_:receive(buf)
    print(buf:first(nread))
end
```

Although `open()` won't work in Capsicum mode, `openat()` will, and Capsicum will make sure the relative paths are only resolved to a hierarchy beneath the given directory-fd. I have a rough idea on how to make something similar work on Linux by abusing many layers of workarounds, but I don't have any code to show yet. Stay tuned til then.

I haven't actually faced many problems using Capsicum so the only complaint I've had was fixed long ago. There's not much to talk about Capsicum. The system is incredibly simple to use yet powerful. This model will be the inspiration for all sandboxes developed in the rest of this article no matter the OS.

Non-blocking IO on UNIX: it sucks!

Once we do receive a file descriptor from a sandbox, it's time to operate on it. However if we're sloppy about it, our thread will block. We need to avoid blocking operations to dodge some DoS attempts. The mess around non-blocking IO on UNIX has been long known. Believe me when I say I have my own share of comments to make here, but this article isn't about async IO and the text is already getting too long, so I'll just present a boring summary of what you need to know:

- `close()` may block according to POSIX.
- Supposedly `close()` always succeed on Linux. Don't bother checking for errors here.
- You may need to create a thread to run `close()` if "slow-to-close" files are a problem.
- Use `fstat` on the received file descriptor to check whether it's a socket.
- On sockets, use `MSG_DONTWAIT` on `recv()`. Boost.Asio still gets this wrong, but I'm short on time to send bugfixes anytime soon.
- On non-sockets, use a proactor (completion events opposed to readiness events) to perform IO operations (e.g. io_uring on Linux, POSIX AIO on FreeBSD, ...).
- On FreeBSD, you can now use `aio_read2()` with `AIO_OP2_FOFFSET` to read without an offset. Other approaches will likely fail with `ENOTCAPABLE`. Boost.Asio also gets this wrong, and I'm short on time to send bugfixes here too.
- On Linux, io_uring is widely distrusted and disabled. Therefore you might just as well reject non-socket IO on Linux if the file descriptor was received from a sandboxed process.

By now you should understand that there are actually two use cases:

- Trusted process creates a resource (file descriptor) and sends it to a distrusted sandboxed process.
- The distrusted sandboxed process creates a resource and sends it elsewhere.

If the file descriptor was created by a trusted process then the range of options we can work with widens. However state such as `O_NONBLOCK` is shared among all copies of a file descriptor and become a problem the moment the file descriptor reaches the first sandboxed process. On Capsicum we can at least forbid `F_SETFL` and alleviate the problem slightly, but this approach only works for FreeBSD.

Sandboxing existing code

By now you should have enough tools in your toolbox to sandbox all your future code. However there's a reason why we sandbox code. Projects grow high and large until it becomes impossible to ensure their code is... bug-free. A little bug in just one of the dozens of modules within a project

shouldn't equal to a fully compromised system when the bug is exploited by a hacker. Damage should be contained. Security policies implemented by OS tools external to your code can only work at a program (or user) level. The program will still be fully compromised and the hacker will have access to all data and credentials the program has access to.

When your program deals with data that can be processed independently, there's an opportunity to implement a safer approach. If you can run multiple instances of your program as different users on your OS, you can use existing security solutions in your project. However if the concept of allocating defined portions of the data to fixed users doesn't work for your project, you may need something more complex or custom-tailored. When the relationships between the users are blurry and your project demands policies that are more dynamic, you may need sandboxes.

As a rule of thumb, every shell should have sandboxes. Shell are programs that act as the membrane that sits between the human operator and some virtual world. Tablets, smartphones, and laptops display graphical shells to interact with programs, windows, and files. Servers employ textual shells. Likewise web browsers act as the shells to the www world.

I wouldn't be surprised if Firefox and Chrome were the only software employing discretionary privilege dropping that you know. They are shells after all so it matters to them. More than that, they're very well funded projects. Sandboxing used to be very expensive (especially outside FreeBSD). However these shouldn't be the only software out there with builtin sandboxing support. Take Telegram, for instance. The right media parsing bug could mean a hacker having access to all my chat history. What time does my son leave school? What people do I trust my credit card info with? When will I go in a trip and leave my house unattended? These are just a few examples of the damage that might be done due to the lack of sandboxes in Telegram. Not only Telegram, but every instant messenger should be employing sandboxes. Media parsing should always be performed in dedicated sandboxes.

The first step into this direction is a realistic approach to real-world engineering: let's **not** rewrite all code from scratch. Deal? The tricks you learned earlier will still be useful, but from now on I'll share tricks to work on existing real-world code. Capsicum users refer to the ability to run unmodified code within sandboxes as oblivious sandboxing. Techniques for oblivious sandboxing most often than not have nothing to do with discretionary privilege dropping and can't solve the problems we were mentioning just a second ago. However it's possible to combine approaches from both worlds in the same project so it's important to study the techniques for oblivious sandboxing too.

The one place we need to look at to implement oblivious sandboxing is actually pretty obvious: the ambient authority functions. In fact, that's what projects such as Super Capsicumizer 9000 do. They inject a dynamic library into a process using `LD_PRELOAD` to interpose ambient authority calls. This technique is actually yesterday news and projects such as fakeroot have been using it for decades.

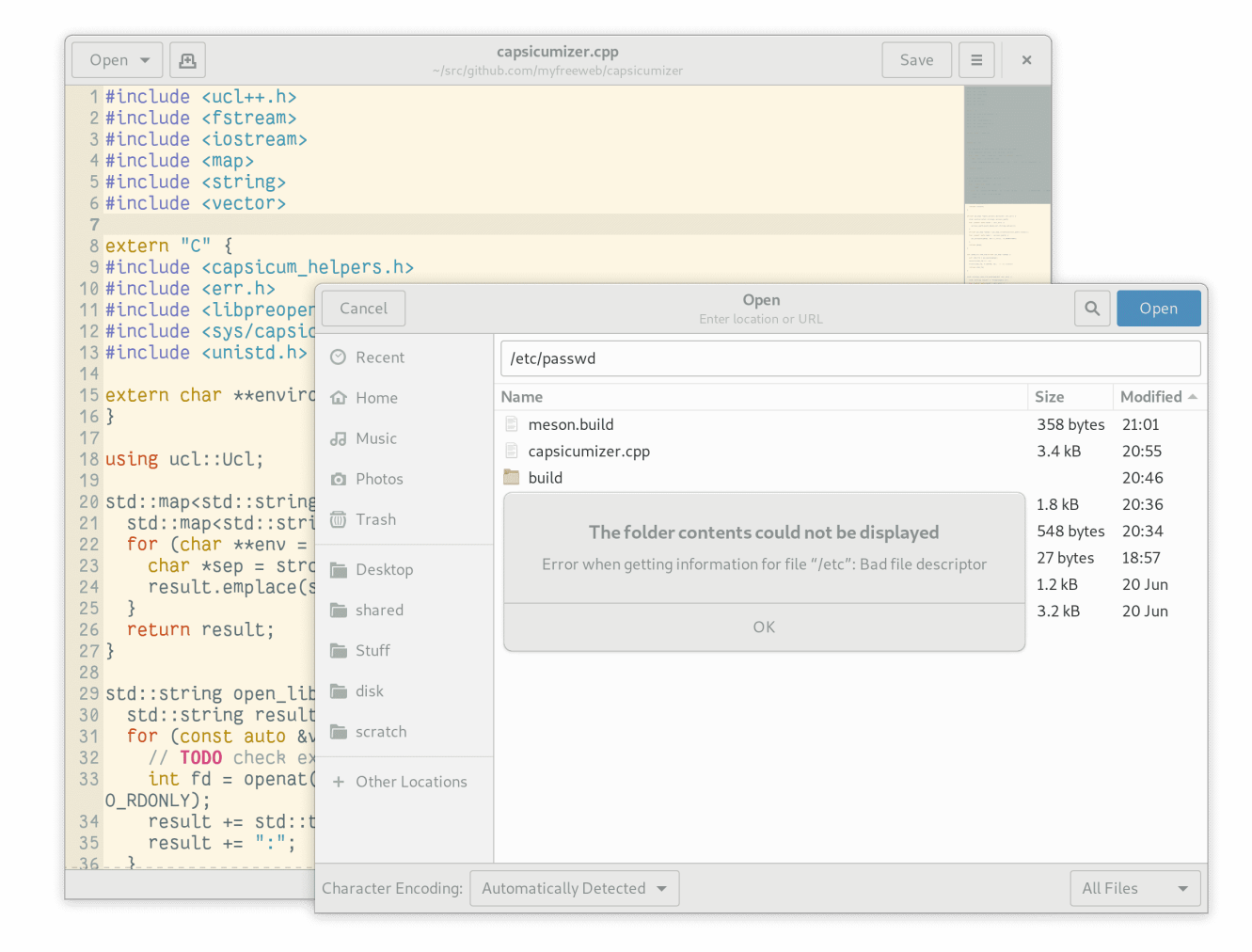

Figure 3. Super Capsicumizer 9000 demo: refusing to open `/etc` on gedit

Super Capsicumizer 9000 is actually a small experiment hacked together by a very very small team. The experiment succeeded into opening old software built on top of complex libraries with a long history of changes. This is very promising. It's a sign that maybe a single programmer working alone to interpose just a few functions for ambient authority access will have success in running legacy code.

Programmers almost never do syscalls directly, and instead rely on libc to do the syscalls on their behalf. That's why this approach works so well. All you have to do is to write a definition for the function from libc you want to interpose. If you're linking against the dynamic libc, your function will be loaded first and used instead. If you're linking against the static libc, chances are that the libc symbol is actually a weak symbol so it'll be dropped once the static linker see your definition. Emilua has been using this approach to support dynamic and static executables on Linux and FreeBSD and so far `getaddrinfo` was the only ambient authority function whose symbol lacked the attribute for weak symbols (please comment on the linked bug reports if you plan to build your own sandboxes using the same techniques or using Emilua):

- https://sourceware.org/bugzilla/show_bug.cgi?id=32509.
- https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=283528.

The next step is choosing which functions to interpose. Functions from FreeBSD's libcasper are good first candidates. However for some reason libcasper doesn't interpose the functions it intends to replace so you'll need to change names and parameters accordingly. libcasper functions (e.g.

`cap_getaddrinfo`) always take an extra parameter. Another good source of inspiration to decide which functions to interpose is the library used by Super Capsicumizer 9000: libpreopen. Most of the time, you'll only need to interpose a few functions even for complex projects.

> Chromium renderers need very little authority. They need access to fontconfig to find fonts on the system and to open those font files.
>
> — https://www.imperialviolet.org/2009/08/26/seccomp.html

Emilua 0.11 abstracts all these details into the module `libc_service`. The example below shows how we use this module to override the behavior of `open` to return a rogue file descriptor when the subprocess try to open `/dev/null`. Actual sandboxing setup (i.e. privilege dropping within the new subprocess) is omitted for brevity. The example also shows how to prefill the code cache for the new subprocess so it won't query the filesystem to fetch the Lua code to execute.

```
local libc_service = require 'libc_service'
local stream = require 'stream'
local pipe = require 'pipe'
local fs = require 'filesystem'

local master, slave = libc_service.new()

slave.open = [[
local real_open, path, flag, mode = ...
local res, errno, fd = real_open(path, flag, mode)
if fd then
    return fd
else
    return res, errno
end
]]

local source_tree_cache = {}
source_tree_cache['a.lua'] = [[
local stream = require 'stream'
local file = require 'file'
local fs = require 'filesystem'

local f = file.stream.new()
f:open(fs.path.new('/dev/null'), {'read_only'})
f = stream.scanner.new{ stream = f }
print(f:get_line())
]]

spawn_vm{
    module = fs.path.new('/a.lua'),
    subprocess = {
        source_tree_cache = source_tree_cache,
        libc_service = slave,
```

```
        stdout = 'share',
        stderr = 'share',
    }
}

spawn(function() pcall(function()
    while true do
        master:receive()
        if master.function_ ~= 'open' then
            master:use_slave_credentials()
            goto continue
        end
        local p, f, m = master:arguments()
        if p ~= fs.path.new('/dev/null') then
            master:use_slave_credentials()
            goto continue
        end

        local pi, po = pipe.pair()
        pi = pi:release()
        spawn(function()
            stream.write_all(po, '/dev/null contents\n')
            po:close()
        end):detach()
        master:send_with_fds(-2, {pi})
        ::continue::
    end
end) end):detach()
```

Emilua uses UNIX sockets behind the scenes for communication between both processes. This approach allows one to implement fully dynamic security policies. For instance, if you're trying use Telegram's tdlib to implement your own Telegram client, you could have the following rules for your secutiry policy:

- Only resolve name queries to `pluto.web.telegram.org`.
- Only allow connect requests to the IP addresses we resolved in previous steps.

The little Lua script we send to be executed in the sandboxed side means we can apply some simple call fixups at the call site to further broaden the use cases we can tackle. For instance, when sandboxed code try to open a GUI connecting to `/tmp/.X11-unix/X0`, we can send a new file descriptor to an unrelated display server and replace the socket from the original request with the new one using `dup2` from the Lua script at the call site. In fact, we *can* do that:

```
local libc_service = require 'libc_service'
local stream = require 'stream'
local system = require 'system'
local pipe = require 'pipe'
local unix = require 'unix'
local fs = require 'filesystem'
```

```
local preload_libc_path
do
    local pi, po = pipe.pair()
    po = po:release()
    pi = stream.scanner.new{ stream = pi }

    system.spawn{
        program = 'pkg-config',
        arguments = {'pkg-config', '--variable=libpath', 'emilua_preload_libc'},
        environment = system.environment,
        stdout = po,
    }
    po:close()

    preload_libc_path = tostring(pi:get_line())
end

local xephyrconnep
for i = 1, 20 do
    local pi, po = pipe.pair()
    po = po:release()
    pi = stream.scanner.new{ stream = pi }

    local xephyr = system.spawn{
        program = 'Xephyr',
        arguments = { 'Xephyr', ':' .. i, '-displayfd', '3' },
        environment = system.environment,
        extra_fds = {
            [3] = po,
        },
    }
    po:close()

    if pcall(function()
        local nr = tostring(pi:get_line())
        xephyrconnep = '/tmp/.X11-unix/X' .. nr
        return true
    end) then
        break
    end
end
if not xephyrconnep then
    print('Failed to start Xephyr')
    system.exit(1)
end

local master, slave = libc_service.new()

slave.connect_unix = [[
local real_connect, fd, path = ...
```

```
local res, errno, fd2 = real_connect(fd, path)
if fd2 then
    C.dup2(fd2, fd)
    C.close(fd2)
end
return res, errno
]]

local guiappenv = system.environment
guiappenv.DISPLAY = ':0'
guiappenv.LD_PRELOAD = preload_libc_path
guiappenv.EMILUA_LIBC_SERVICE_FD = '3'
local guiapp = system.spawn{
    program = 'xterm',
    arguments = { 'xterm' },
    environment = guiappenv,
    stdout = 'share',
    stderr = 'share',
    extra_fds = {
        [3] = slave,
    },
}
scope_cleanup_push(function() guiapp:wait() end)

spawn(function() pcall(function()
    while true do
        master:receive()
        if
            master.function_ == 'connect_unix' and
            (
                master:arguments() == fs.path.new('\0/tmp/.X11-unix/X0') or
                master:arguments() == fs.path.new('/tmp/.X11-unix/X0')
            )
        then
            local xephyrconn = unix.stream.dial(xephyrconnep)
            master:send_with_fds(0, {xephyrconn:release()})
        else
            master:use_slave_credentials()
        end
    end
end) end):detach()
```

This example also shows that Emilua can make use of `LD_PRELOAD` to perform libc interposition on existing programs such as xterm.

Another interesting approach that might prove useful to your projects is to use `kcmp` in your security policies. This technique would allow you to increase your policies granularities even further by implementing different subpolicies for each file descriptor.

These techniques are already probably more than what you need, but I have few more tricks up in

my sleeve to share, so let's move on.

Sandboxing native plugins

A common theme in the threat model of sandboxes is to assume that initially trusted code becomes malicious once compromised. For instance, we might trust that ffmpeg developers are well-intentioned and didn't backdoored their project. However ffmpeg is a complex project and legitimate bugs lurk around just waiting to be found. Some of these bugs might be exploitable by hackers. In this case we can import ffmpeg as a library in our executable and only setup the sandbox right before we call ffmpeg functions on external data.

However how'd we approach the sandboxing steps if we assumed the code to be compromised from the start? Let's take Telegram's tdlib as an example. Suppose you don't trust tdlib at all. Under this threat model, even just loading the library would be a dangerous operation. For such scenario, first we need to build tdlib under a secure environment (e.g. jails on FreeBSD, namespaces on Linux). Once we do have the built plugin, we can move to the next challenge.

If we disable ambient authority to sandbox the code, `dlopen()` will fail to access the filesystem. To work around this issue, we can `dlopen` by file descriptors instead. On FreeBSD, we can use `fdlopen`. On Linux, we can pass a path to `/proc/self/fd/` as long as we never close the file descriptor to avoid path reuse by a different plugin (glibc will deduplicate plugins by path). That's not a perfect solution, but it's a start.

On some previous example, we saw code cache prefilling as an Emilua way to instruct a policy to avoid filesystem queries. Emilua just follows the same trend for plugins and exposes `native_modules_cache` to prefill the native plugin cache:

```
spawn_vm{
    module = 'some_module',
    subprocess = {
        native_modules_cache = {
            'some_plugin'
        } } }
```

The biggest problem with plugins is that they might depend on not yet loaded dynamic libraries and `dlopen` would fail to load them. We can try to use the workaround for libc-service that we saw in the previous section, but there are cleaner solutions for this problem. On Linux, we can simply use Landlock. Landlock would already be required for the `/proc/self/fd/` trick anyway. On FreeBSD, we can use `rtld_set_var` and `LIBRARY_PATH_FDS`.

```
spawn_vm{
    module = 'some_module',
    subprocess = {
        native_modules_cache = {
            'some_plugin'
        },
        ld_library_directories = library_path_fds,
```

```
    } }
```

Fortunately we won't need to worry about this for Tdlib so the code becomes slightly simpler. However it begs the question: if we don't trust tdlib, why are we trusting it with our data anyway? The question here is to evaluate if the threat model even makes sense. In the case of tdlib, we aren't giving tdlib's developers (Telegram developers) anything they don't already have (data on Telegram servers). Running tdlib within a plugin prevents the Telegram company to have unrestricted access to data on our systems. The answer for tdlib might be simple, but that's not what matters in this story. The lesson that you should take here is to evaluate whether your threat models even make sense.

Let's explore another threat model story. The Linux kernel can load compressed initramfs images. Even if the Linux kernel uses a buggy unmaintained library full of well known exploits to decompress the initramfs image, it doesn't matter at all! We only use this library on data generated by a trusted user. If some adversarial actor had control over the initramfs images we load, the actor could already do any damage he wished for even if the decompression library had zero exploitable bugs. However the story would be completely different if the library were backdoored. Sometimes it's more important to have auditable code written by trusted individuals than supposedly better code written by individuals we aren't sure we can trust.

Dropping privileges with Seccomp

I've postponed this section for as long as I could because it's awful. Seccomp is not a good mechanism for discretionary privilege dropping. Seccomp is a good mechanism for OS hardening. Seccomp is a simple programmable syscall filtering mechanism based on BPF programs. The BPF program must choose an action for each syscall attempted by the process:

- `SECCOMP_RET_KILL_PROCESS`.
- `SECCOMP_RET_KILL_THREAD`.
- `SECCOMP_RET_TRAP`.
- `SECCOMP_RET_ERRNO`.
- `SECCOMP_RET_USER_NOTIF`.
- `SECCOMP_RET_TRACE`.
- `SECCOMP_RET_LOG`.
- `SECCOMP_RET_ALLOW`.

This mechanism can be used to deny access (e.g. `SECCOMP_RET_ERRNO`) to ambient authority by disallowing syscalls that work on names (e.g. `open`, `bind`). All you have to do is disable ambient authority, then your process will be properly sandboxed and you can use the lessons learned in previous sections for compartmentalised application development. Using this mechanism you may either implement whitelists or blacklists. These projects implement syscall blacklists:

- https://github.com/lxc/lxc/blob/v6.0.3/config/templates/common.seccomp
- https://github.com/flatpak/flatpak/issues/4187#issuecomment-1075512546

- https://github.com/flatpak/flatpak/blob/1.16.0/common/flatpak-run.c#L1839

The problem with blacklists is well known. New kernel versions may add new syscalls. You don't know today what syscalls will be added tomorrow. You don't know today if tomorrow's syscall breaks your today's policy. However the problem is even worse for seccomp. Linux allows multiarch systems and syscall numbering varies wildly among arches. Even if you block syscalls such as `acct` on your x86-64 program, a compromised sandbox could bypass the syscall filter by running a x86 executable. To our delight, Linux somehow manages to make the problem even worse:

> The `arch` field is not unique for all calling conventions. The x86-64 ABI and the x32 ABI both use `AUDIT_ARCH_X86_64` as arch, and they run on the same processors. Instead, the mask `__X32_SYSCALL_BIT` is used on the system call number to tell the two ABIs apart.
>
> This means that a policy must either deny all syscalls with `X32_SYSCALL_BIT or it must recognize syscalls with and withoutX32_SYSCALL_BIT` set. A list of system calls to be denied based on `nr` that does not also contain `nr` values with `X32_SYSCALL_BIT set can be bypassed by a malicious program that setsX32_SYSCALL_BIT`.
>
> — seccomp(2)

Even when we do migrate to whitelists, we'll still be haunted by these implementation details. Here are a few notable projects based on whitelists:

- https://github.com/moby/moby/blob/v27.4.1/profiles/seccomp/default.json
- https://github.com/containers/podman/blob/v5.3.1/vendor/github.com/containers/common/pkg/seccomp/seccomp.json
- https://gitlab.gnome.org/GNOME/localsearch/-/blob/3.8.2/src/libtracker-miners-common/tracker-seccomp.c#L141
- https://android.googlesource.com/platform/bionic/+/704772bda034448165d071f68b6aeca716f4220e/libc/seccomp/seccomp_policy.cpp

Once you do go through these lists to implement your own seccomp policies, you'll face another problem: Linux syscall parameter ordering changes among arches. That's why Docker uses multiple rules for the syscall `clone`. Why do we have to become experts in Linux syscall conventions to do what FreeBSD users get done in 10 seconds by just calling `cap_enter()`? Welcome to Linux.

The good news is that in theory we could implement some library with a single function to disable ambient authority and everyone would then just use this library. I have a few ideas on how to implement this library, but so far no customer of mine was interested in this problem (at the same time I'm busy with different projects).

However there's just another caveat you must keep in mind. Linux userspace relies on the filesystem too much. Even if you interpose `open` for compatibility with legacy code, legacy code will

fail trying to access `/proc/self` when nested sandboxes are at play. It's better to just allow `open` and filter filesystem access with Landlock instead. The problem now is that file descriptors can't be modeled as capabilities if a process can just reopen `/proc/self/fd/` with a different `mode`. Landlock developers shared a long-term goal to expose capabilities compatible with Capsicum, so maybe we'll have a solution to this problem in the future.

Seccomp's complexity might be disheartening for the compartmentalised application developer, but the situation is even worse if you look into Linux namespaces. For Linux, Seccomp + Landlock is what we have.

Easier Seccomp with Kafel

> Kafel is a language and library for specifying syscall filtering policies. The policies are compiled into BPF code that can be used with seccomp-filter.
>
> — https://google.github.io/kafel/

Kafel is the most promising project to enable Seccomp policy reusing that I've seen during my quests. Using Kafel I've managed to define a few policy groups that might be of interest to you:

Kafel policies

```
// These policies are heavily influenced by Docker's default profile. Further
// customization done on top:
//
// - Avoid syscalls that need root anyway. The policies here are mostly meant to
//   be used by unprivileged users (not containers with root inside). The
//   syscalls wouldn't be harmful, but would result in larger BPF programs that
//   in turn incur more overhead.
// - Avoid rarely used syscalls that can be abused for yet more fingerprinting
//   on desktop applications. This category mostly contains syscalls useful for
//   profiling (e.g. mincore, cachestat).
// - Split them into categories inspired by systemD's seccomp filter sets and
//   OpenBSD's pledge promises.

POLICY Aio {
    ALLOW {
        io_cancel, io_destroy, io_getevents, io_pgetevents, io_setup, io_submit
    }
}

POLICY BasicIo {
    ALLOW {
        read, readv, tee, vmsplice, write, writev,

        // ioctl() is definitively not about generic/stream/basic I/O. ioctl()
        // is really a syscall in disguise that device drivers can use for
        // anything. However it's expected that any program doing file I/O or
        // socket I/O or TTY IO will eventually stumble on glibc using ioctl()
```

```
        // for some operations so let's go ahead and just include it in the
        // basic IO set to force other IO categories to include it too.
        ioctl
    }
}

POLICY Clock {
    ALLOW {
        clock_getres, clock_gettime, gettimeofday, time, times
    }
}

// Compat quirks. This family of policies is a good candidate to be maintained
// in a different repo.
POLICY CompatX86 {
    ALLOW {
        // important for old ABI emulation
        personality(persona) {
            persona == /*PER_LINUX=*/0 || persona == /*PER_LINUX32=*/8 ||
            persona == /*UNAME26=*/0x0020000 ||
            persona == /*PER_LINUX32|UNAME26=*/0x20008 ||
            persona == 0xffffffff
        },

        // Important for x86 family's ABI. We put it in here instead of
        // c-runtime because other archs don't need it. Ideally Kafel would
        // allow us to write arch_prctl@amd64 in c-runtime and the rule would
        // only be included when we're building for the amd64 arch.
        arch_prctl
    }
}

POLICY CompatDB32 {
    ALLOW {
        remap_file_pages
    }
}

POLICY CompatSystemd {
    ALLOW {
        // SystemD uses this to get mount-id
        name_to_handle_at
    }
}

POLICY CompatWine {
    ALLOW {
        modify_ldt
    }
}
```

```
POLICY Credentials {
    ALLOW {
        getegid, geteuid, getgid, getgroups, getresgid, getresuid, getuid
    }
}

POLICY CredentialsExtra {
    ALLOW {
        // SystemD lists this syscall in the policy 'process' with the reasoning
        // that it's able to query arbitrary processes so it's a process
        // relationship related syscall. Following the same reasoning, we opt to
        // not include this syscall in the policy 'credentials' as other
        // syscalls in that category don't allow querying arbitrary
        // processes. However we also opt to not include capget in the category
        // 'process' given most usages of that policy won't need capget at all
        // and would just make the resulting BPF bigger.
        capget
    }
}

POLICY CredentialsMutation {
    ALLOW {
        capset, setfsgid, setfsuid, setgid, setgroups, setregid, setresgid,
        setresuid, setreuid, setuid
    }
}

// Memory allocation, threading, syscall interaction (or libc support) and
// functions that should always be available (e.g. exit_group to bail out as a
// program's last resort).
//
// Do notice that actually opening a libc-based program requires access to much
// more syscalls as the loader is going to scrape the filesystem for the
// required libraries and do many operations to stich the program image
// together. The idea here is to apply a filter that will allow the C runtime to
// keep running after we already have the program image in RAM.
POLICY CRuntime {
    ALLOW {
        brk, exit, exit_group, futex, futex_requeue, futex_wait, futex_waitv,
        futex_wake, get_robust_list, get_thread_area, gettid, madvise,
        map_shadow_stack, membarrier, mmap, mprotect, mremap, munmap,
        restart_syscall, rseq, sched_yield, set_robust_list, set_thread_area,
        set_tid_address,

        // glibc's malloc() has references to getrandom(), so it's included here
        getrandom
    }
}

// These syscalls are already gated by YAMA's ptrace_scope or capabilities
// (e.g. CAP_PERFMON). The usual reasoning would be that it's safe to permit
```

```
// them, but:
//
// - They are really only useful for process inspection/debugging.
// - For IPC usage, better mechanisms exist (e.g. one can memfd+seal+mmap to
//   have zero copy I/O between cooperating processes).
// - They appeared in a few CVEs in the past.
POLICY Debug {
    ALLOW {
        kcmp, pidfd_getfd, perf_event_open, process_madvise, process_mrelease,
        process_vm_readv, process_vm_writev, ptrace
    }
}

POLICY FileDescriptors {
    ALLOW {
        close, close_range, dup, dup2, dup3, fcntl
    }
}

// This policy is split off from filesystem so a process could still perform
// file IO on:
//
// - Already open files.
// - Files received from UNIX sockets.
// - Memfds.
POLICY FileIo {
    ALLOW {
        copy_file_range, fadvise64, fallocate, flock, ftruncate, lseek, pread64,
        preadv, preadv2, pwrite64, pwritev, pwritev2, readahead, sendfile,
        splice
    }
}

// OpenBSD's pledge further breaks down this promise into rpath, wpath, cpath
// and dpath, but Landlock would be more appropriate to mirror the intention of
// such granular designs
POLICY Filesystem {
    ALLOW {
        access, chdir, creat, faccessat, faccessat2, fchdir, fgetxattr,
        flistxattr, fstat, fstatfs, getcwd, getdents, getdents64, getxattr,
        inotify_add_watch, inotify_init, inotify_init1, inotify_rm_watch,
        lgetxattr, link, linkat, listxattr, llistxattr, lstat, mkdir, mkdirat,
        mknod, mknodat, newfstatat, open, openat, openat2, readlink, readlinkat,
        rename, renameat, renameat2, rmdir, stat, statfs, statx, symlink,
        symlinkat, truncate, umask, unlink, unlinkat
    }
}

// Allowed to make explicit changes to fields in struct stat relating to a file.
POLICY FilesystemAttr {
    ALLOW {
```

```
        chmod, chown, fchmod, fchmodat, fchmodat2, fchown, fchownat,
        fremovexattr, fsetxattr, futimesat, lchown, lremovexattr, lsetxattr,
        removexattr, setxattr, utime, utimensat, utimes
    }
}

// Event loop system calls.
POLICY IoEvent {
    ALLOW {
        epoll_create, epoll_create1, epoll_ctl, epoll_ctl_old, epoll_pwait,
        epoll_pwait2, epoll_wait, epoll_wait_old, eventfd, eventfd2, poll,
        ppoll, pselect6, select
    }
}

// io_uring nowadays is considered unsafe for general usage:
// http://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html
POLICY IoUring {
    ALLOW {
        io_uring_enter, io_uring_register, io_uring_setup
    }
}

// SysV IPC, POSIX Message Queues or other IPC.
POLICY Ipc {
    ALLOW {
        memfd_create, mq_getsetattr, mq_notify, mq_open, mq_timedreceive,
        mq_timedsend, mq_unlink, msgctl, msgget, msgrcv, msgsnd, pipe, pipe2,
        semctl, semget, semop, semtimedop, shmat, shmctl, shmdt, shmget
    }
}

// Memory locking control.
POLICY Memlock {
    ALLOW {
        memfd_secret, mlock, mlock2, mlockall, munlock, munlockall
    }
}

POLICY NetworkIo {
    ALLOW {
        connect, getpeername, getsockname, getsockopt, recvfrom, recvmmsg,
        recvmsg, sendmmsg, sendmsg, sendto, setsockopt, shutdown
    }
}

POLICY NetworkServer {
    ALLOW {
        accept, accept4, bind, listen
    }
}
```

```
POLICY NetworkSocketTcp {
    ALLOW {
        socket(domain, type, protocol) {
            (type & 0x7ff) == /*SOCK_STREAM=*/1 && protocol == 0 &&
            (domain == /*AF_INET=*/2 || domain == /*AF_INET6=*/10)
        }
    }
}

POLICY NetworkSocketUdp {
    ALLOW {
        socket(domain, type, protocol) {
            (type & 0x7ff) == /*SOCK_DGRAM=*/2 && protocol == 0 &&
            (domain == /*AF_INET=*/2 || domain == /*AF_INET6=*/10)
        }
    }
}

POLICY NetworkSocketUnix {
    ALLOW {
        socket(domain, type, protocol) {
            domain == /*AF_UNIX=*/1 && protocol == 0
        },
        socketpair(domain, type, protocol) {
            domain == /*AF_UNIX=*/1 && protocol == 0
        }
    }
}

// System calls used for memory protection keys.
POLICY Pkey {
    ALLOW {
        pkey_alloc, pkey_free, pkey_mprotect
    }
}

// Process control, execution, namespacing, relationship operations.
//
// Most likely you'll ALWAYS need access to this set to sandbox other binaries:
// <https://lore.kernel.org/all/202010281500.855B950FE@keescook/T/>. It's only
// really practical to exclude this set from the seccomp filter if you're
// sandboxing yourself (i.e. cooperatively dropping further privileges before
// doing dangerous stuff). It's a shame that Linux doesn't offer this type of
// transition-on-exec mechanism for seccomp nor cgroups. Folks from SELinux
// already know just how important it is to support this kind of mechanism for
// properly dropping privileges, and it'd be good for more kernel hackers to
// learn this lesson as well.
POLICY Process {
    ALLOW {
        // Where's clone2? ia64 is the only architecture that has clone2, but
```

```
        // ia64 doesn't implement seccomp. c.f.
        // acce2f71779c54086962fefce3833d886c655f62 in the kernel.
        clone, clone3, execve, execveat, fork, getpgid, getpgrp, getpid,
        getppid, getrusage, getsid, kill, pidfd_open, pidfd_send_signal, prctl,
        rt_sigqueueinfo, rt_tgsigqueueinfo, setpgid, setsid, tgkill, tkill,
        vfork, wait4, waitid
    }
}

POLICY Resources {
    ALLOW {
        getcpu, getpriority, getrlimit, ioprio_get, sched_getaffinity,
        sched_getattr, sched_getparam, sched_get_priority_max,
        sched_get_priority_min, sched_getscheduler, sched_rr_get_interval
    }
}

// Alter resource settings.
POLICY ResourcesMutation {
    ALLOW {
        ioprio_set, prlimit64, sched_setaffinity, sched_setattr, sched_setparam,
        sched_setscheduler, setpriority, setrlimit
    }
}

POLICY Sandbox {
    ALLOW {
        landlock_add_rule, landlock_create_ruleset, landlock_restrict_self,
        seccomp
    }
}

// Process signal handling.
POLICY Signal {
    ALLOW {
        pause, rt_sigaction, rt_sigpending, rt_sigprocmask, rt_sigreturn,
        rt_sigsuspend, rt_sigtimedwait, sigaltstack, signalfd, signalfd4
    }
}

// Synchronize files and memory to storage.
POLICY Sync {
    ALLOW {
        fdatasync, fsync, msync, sync, sync_file_range, syncfs
    }
}

// Schedule operations by time.
POLICY Timer {
    ALLOW {
        alarm, getitimer, clock_nanosleep, nanosleep, setitimer, timer_create,
```

```
        timer_delete, timer_getoverrun, timer_gettime, timer_settime,
        timerfd_create, timerfd_gettime, timerfd_settime
    }
}
```

These are the policies I've been using for almost a year. I'd make a few changes nowadays, but I haven't gotten the time to it yet. Also do keep in mind that Kafel's syscall database is inexcusably poor so you'll need a few changes (some `sed`-like preprocessing to replace the syscall names by their numbers) to make the above work. Even when it does work, it'll be omitting many syscalls that projects using better syscall databases such as Docker handle. Did you notice that we don't mention `clock_gettime64`? That's because Kafel's syscall database is just too poor. Kafel won't ever be adopted by projects such as Docker for as long as it retains such poor syscall databases and poor multiarch support.

However a lack of better syscall databases isn't the only thing we could improve in Kafel. I'd like to see policy versioning and better policy composition operators, for instance. I'd be willing to develop these features, but again, no current customer of mine was interested in this project, so my time will be spent in different projects.

The end?

In this article, I've gone just over the basics for sandboxing in Linux and FreeBSD. However there are more lessons that I'd like to share. I'll save these for a later article in the future. Topics that I'll likely cover if I ever get into the mood to write another blog post again:

- More demos (for years I've been running graphical apps in some of my machines solely in containers so I got quite a few demos to show).
- More about the actor model, capability-based security, and access control policies.
- More about Linux kludges.
- Maybe a comment or two on Windows and macOS.
- Containers (Emilua also works as a container runtime).
- Sandboxing GUI applications.
- Attack surfaces & safe parsers.
- FreeBSD's libnv.
- Rights revocation & proxies.
- Sandboxing patterns.
- UNIX tricks for the C programmer that Emilua makes use of.