Towards oblivious sandboxing

Jonathan Anderson, Stanley Godfrey
and Robert N M Watson

For those playing along at home: **https://github.com/trombonehero/sandbox-examples**, **https://github.com/freebsd/freebsd**

This work has been sponsored by the Research & Development Corporation of Newfoundland & Labrador (contract 5404.1822.101), the NSERC Discovery program (RGPIN-2015-06048) and the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Background: Capsicum

- framework for *principled*, *coherent* compartmentalization
- compartmentalization: application subdivision

Principled:

- draws on rich history of computer security concepts and literature
- monotonic reduction of authority

Coherent:

- clear, simple policies
- uniform application across applications

Background: capability mode

No access to global namespaces:

- PIDs
- file paths, filesystem IDs
- NFS file handles
- socket protocol addresses
- sysctl MIBs
- POSIX, SysV IPC names
- system clocks
- jails, CPU sets

Hotel California

Strong isolation

Alternative syscall filter: `seccomp(2)`

- filter system calls with BPF programs
- easy: check syscall number (on same arch)
- hard: check arguments (e.g., filenames)
- impossible: check arguments *meaningfully* (just like `systrace`)

```
#define Allow(syscall) \
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_##syscall, 0, 1), \
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW)

struct sock_filter filter[] = {
    // Check architecture: syscall numbers arch-dependent!
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, ArchField),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 1, 0),
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),

    // Check syscall:
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, SYSCALL_NUM_OFFSET),
    Allow(brk),             // allow stack extension
    Allow(close),           // allow closing files!
    /* ... */
    Allow(openat),          // to permit openat(config_dir), etc.
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRAP),    // or die
```

Thanks: https://eigenstate.org/notes/seccomp

Alternative syscall filter: `pledge(2)`

- *much* simpler filters
  - e.g., `stdio` limits to `clock_getres(2)`, `close(2)`, `dup(2)`, `fchdir(2)`, `read(2)`…
  - `rpath` allows read-only effects on the filesystem: `chdir(2)`, `getcwd(3)`, `openat(2)`, …
- optional path whitelisting

Still insufficient!

```
// Enter sandbox!
if (pledge("stdio rpath cpath flock", NULL) < 0)
{
    err(-1, "error in pledge()");
}
```

```
// Or we could've whitelisted a few specific paths
// (assuming we know them all in advance).
const char *paths[] =
{
    "foo.lock",
    "bar.lock!",
    NULL,
};
```

pledge(2) and seccomp(2): the good news

Can sandbox trivial applications trivially

If your application only needs `read(2)`, `write(2)`, `close(2)`, etc.:

```
+if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT) != 0)
+{
+     err(-1, "error inpledge()");
+}
+
 if (excite_file(STDIN_FILENO, // ...
```

```
+if (pledge("stdio", NULL) != 0)
+{
+     err(-1, "error in pledge()");
+}
+
 if (excite_file(STDIN_FILENO, // ...
```

Demo!

pledge(2) and paths

Slightly more interesting program:

```
[openbsd-vm jon]$ ./do_stuff ../conf `mktemp -d [...]`
hello!
package conf
locked.
[openbsd-vm jon]$ ls
Makefile        do_stuff.c      p0wnd!
do_stuff        do_stuff.o      scratch.MIwnTP
```

- escape from scratch directory

With path whitelist:

```
const char *known_paths[] =
{
    "foo.lock",
    "bar.lock!",
    NULL,
};
```

- enumerate all possible paths
- shallow filtering (e.g., symlinks)
- concurrency leads to TOCTTOU

It's the same story with `seccomp(2)`, just with more complex pattern matching. MacOS Sandbox is... interesting.

Authorizing security-sensitive operations

It's not enough to:

- filter on system call numbers/names
- filter on system call arguments

Authorization must be done:

- atomically with authorized operations
- deep within the kernel (not a wrapper)

Fundamental limitation for:

- systrace
- seccomp w/BPF
- pledge

Background: capabilities

Historic idea:

identifier for an object + *operations* that can be performed on it

Dennis and Van Horn (1966): index into supervisor-maintained *C-list*

Historic capabilities ⇨ PSOS ⇨ Multics ⇨ Unix

Background: file descriptors

Like capabilities:

- index into supervisor-maintained list of objects
- identifiers with operations: `read(2)`, `write(2)`, etc.

Unlike capabilities:

- lots of implicit rights
- lack of monotonic reduction

```
int fd = open("my-data.dat", O_RDONLY);
if (fchmod(fd, 0777) < 0)
    err(-1, "unable to chmod"); // usually doesn't run
```

Background: capabilities

Rigorous focus on allowed operations

proc ⇒ filedesc ⇒ fdescenttbl ⇒ filedescent ⇒ filecaps ⇒ {cap_rights_t fc_rights, fc_ioctls, fc_fcntls}

```c
struct filedescent {
  struct file     *fde_file;
  struct filecaps  fde_caps;
  uint8_t          fde_flags;
  seq_t            fde_seq;
}
```

- allowed syscalls, ioctls, fcntls
- CAP_READ, CAP_FTRUNCATE, CAP_MMAP, CAP_FCHMOD...
- `fget(td, fd, cap_rights_init(&rights, CAP_FSTAT), &fp);`

`open(2)` gives all rights, `cap_rights_limit(2)` limits, `*at(2)`, `accept(2)` derive from others

Background: Capsicum in practice

```
--- a/true/true.c
+++ b/true/true.c
@@ -28,7 +28,11 @@
 #include <sys/cdefs.h>
 __FBSDID("$FreeBSD$");

+#include <sys/capsicum.h>
+
+#include <capsicum_helpers.h>
 #include <err.h>
+#include <errno.h>
 #include <stdbool.h>

 #include <true.h>
```

(see zxombie/libtrue:#5)

```
@@ -37,6 +41,12 @@ int
 main(int argc, char *argv[])
 {

+	if (caph_limit_stdio() != 0)
+		errx(1, "Failed to limit std{in,out,err}");
+
+	if (cap_enter() != 0 && errno != ENOSYS)
+		errx(1, "Failed to enter capability mode");
+
	if (!get_true())
		errx(1, "Bad true value");
```

But more seriously…

Background: Capsicum in practice

- limitation: requires **voluntary** self-compartmentalization

```
 if (lpc_bootrom())
     fwctl_init();

+#ifndef WITHOUT_CAPSICUM
+caph_cache_catpages();
+
+if (caph_limit_stdout() == -1 || caph_limit_stderr() == -1)
+    errx(EX_OSERR, "Unable to apply rights for sandbox");
+
+if (cap_enter() == -1 && errno != ENOSYS)
+    errx(EX_OSERR, "cap_enter() failed");
+#endif

 /*
  * Change the proc title to include the VM name.
  */
 setproctitle("%s", vmname);
```

Long-term goals:

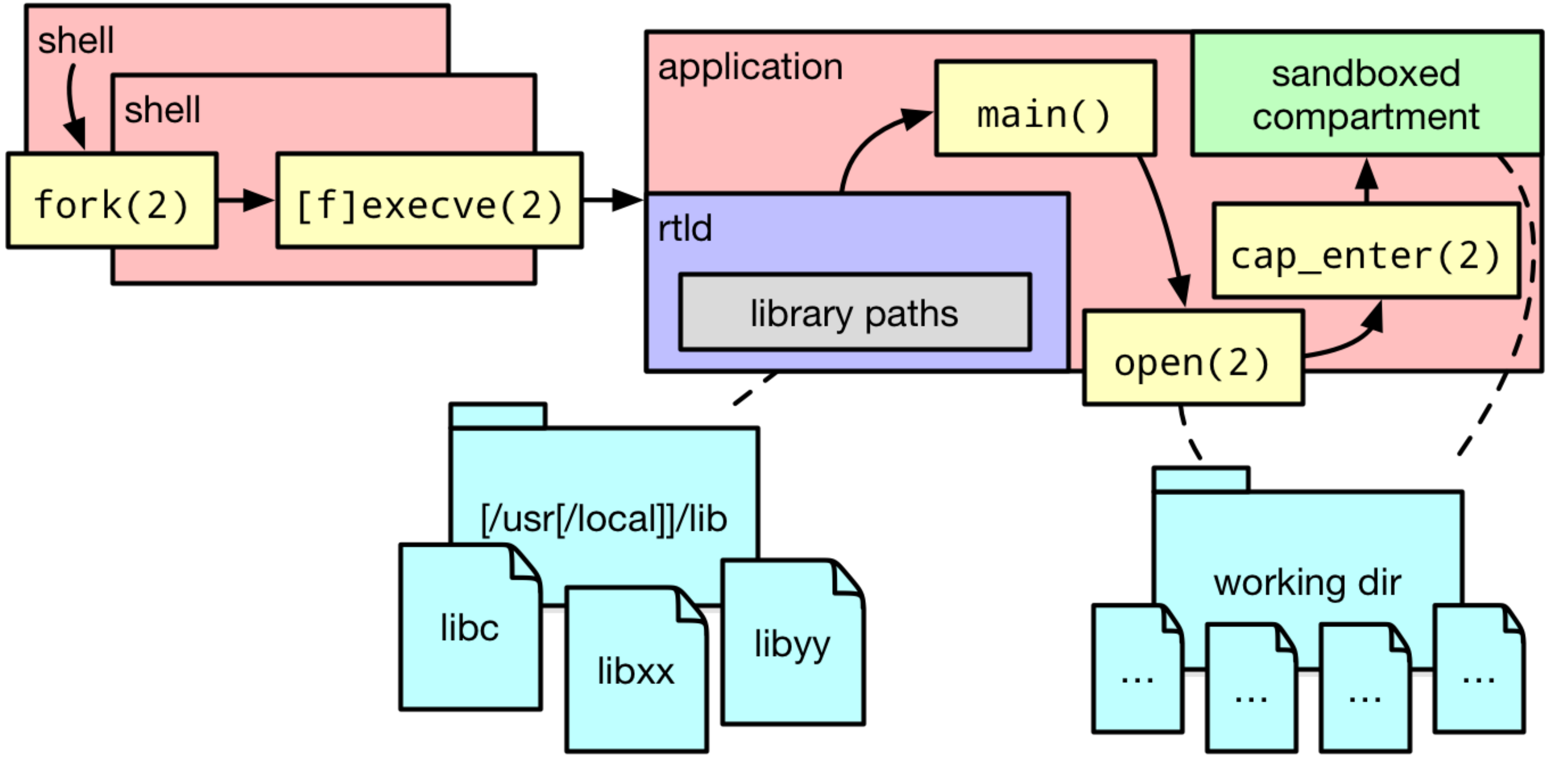
- compartmentalization without modification
- protecting ourselves from vulnerable applications **whether they like it or not**

Sandboxing as she is played today

1. open resources
2. `cap_enter(2)`
3. compute
4. (profit???)

Resources are:

statically-enumerable
or
externally-provided



Resource dependencies

Explicit resources

- files, directories, sockets...
- can pre-open files or directories (for `openat(2)`)
  - pre-opened file descriptors are preserved across `exec(2)` boundary
  - parent process can `fork(2)`, open directory descriptors, `setenv(3)`, `cap_enter(2)`...
- external services (e.g., libcasper, powerbox service*, ...)

---

* Ka-Ping **Yee**, "Aligning security and usability", *IEEE Security and Privacy 2(5)*, 2004 , "App Sandbox in Depth", *Apple Developer Guides*, 2016
.

Resource dependencies (2)

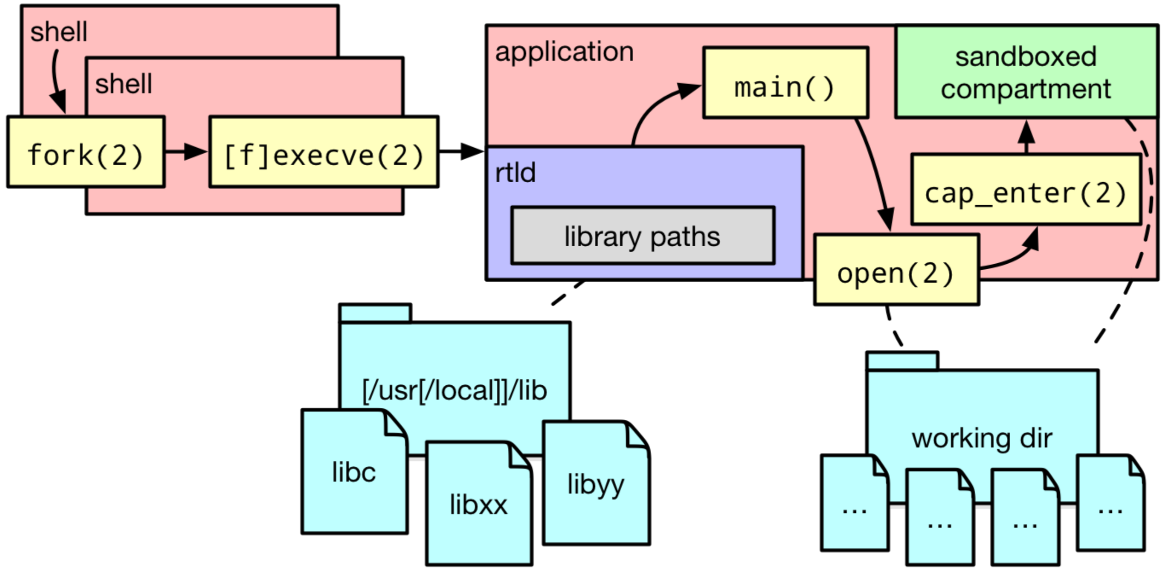
Implicit resources

- locale data (can be pre-cached; see Mariusz' r306657)
- shared libraries: even `cat(1)` and `echo(1)` need `libc`
- but neither `exec(2)` nor run-time linking work in capability mode!

`exec(2)` without a name

Traditional approach:

- `fork(2)` child process
- `exec(2)` binary
  - cleans up memory mappings, closes `O_CLOEXEC` files
  - preserves other open files, environment variables
  - finds binary **by name**, mmap's, transfers control to linker



`exec(2)` without a name (2)

Problematic line:

> finds binary **by name**, mmap's, transfers control to linker

- first problem: "finds binary **by name**"
  - solution: `fexecve(2)` takes already-open file descriptor for binary*
  - `fexecve(binary /* pre-opened? */, args, environ)`
- next problem: "mmap's"
  - but wait... isn't `mmap(2)` allowed in capability mode?
  - yes, but **what are we mapping**?

---

* ask me about `fexecve` on Linux, it's **kind of funny**.

How do we `exec(2)` binaries?

`exec(2)`, `execve(2)`, `fexecve(2)` ⇒ `kern_execve()`
(supports lots of binary *image* formats)

- try process-specific image activator (`p_sysent`)
- try each `execsw[i]->ex_imgact` in turn (a.out, ELF, …)
- ELF: `exec_elfXX_imgact` @ `sys/kern/imgact_elf.c`:

```
static int
__CONCAT(exec_, __elfN(imgact))(struct image_params *)
{
    /* ... */
}
```

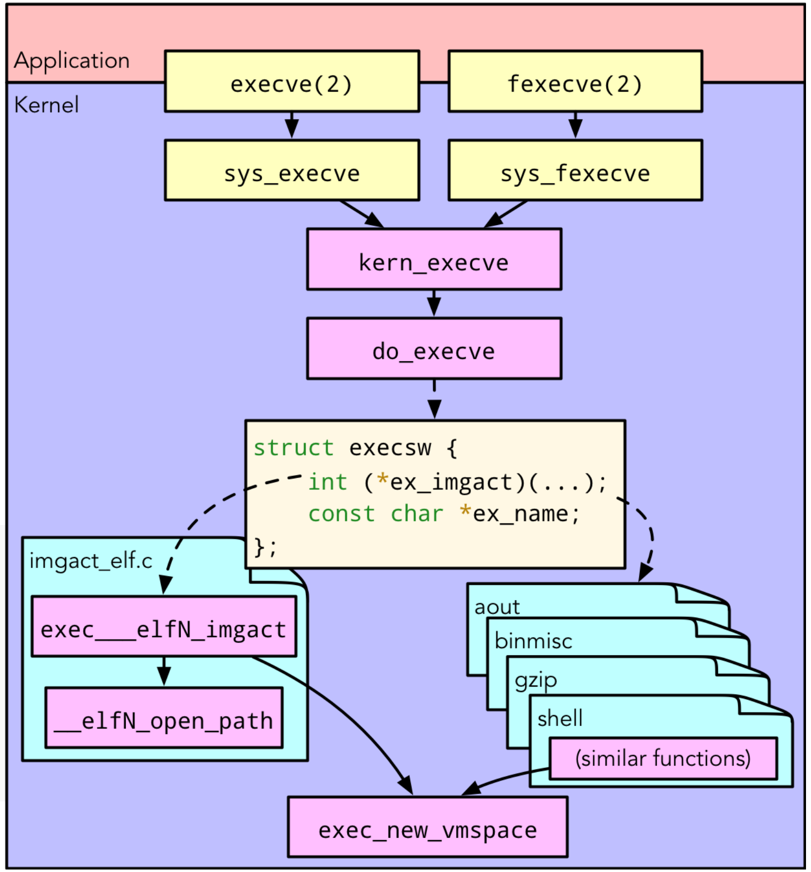
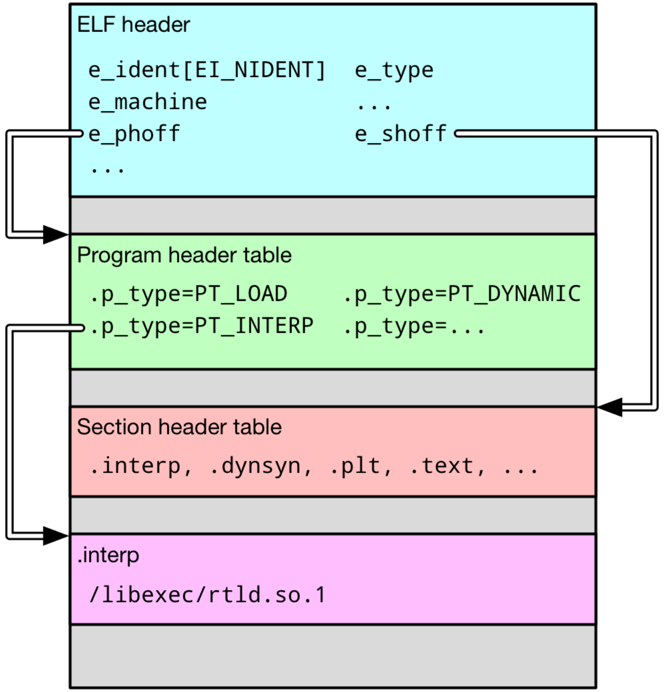


Image activation

- ELF image activator encodes knowledge of run-time linker (a.k.a., run-time "interpreter")
- binaries can also encode a run-time linker path: `PT_INTERP` field in ELF program header table
- ELF image activator **maps both interpreter and binary** into memory, starts running the interpreter
- what's the problem?



Linker always specified by path!

Finding the linker

- in capability mode, `open(2)` syscall disallowed
- more fundamentally, **all name lookups** in capability mode are restricted in `namei()`

see `sys/kern/vfs_lookup.c:350`:

```
if (error == 0 && IN_CAPABILITY_MODE(td) &&
    (cnp->cn_flags & NOCAPCHECK) == 0) {
     ndp->ni_lcf |= NI_LCF_STRICTRELATIVE;
```

```
if (cnp->cn_flags & ISDOTDOT) {
    if ((ndp->ni_lcf & (NI_LCF_STRICTRELATIVE |
        == NI_LCF_STRICTRELATIVE) {
            error = ENOTCAPABLE;
            goto bad;
    }
```

- `NI_LCF_STRICTRELATIVE`
  - don't allow `'/'`, `AT_FDCWD` or `".."`
  - explicit kernel override: `NOCAPCHECK` flag (only used for coredumps)
- **desirable property** of Capsicum's deep-in-the-kernel approach

Finding the linker (2)

The problem:

- can't look up the default RTLD path
- can't use the `PT_INTERP` path
- where can we get a run-time linker?

The solution:

- **punt!***

"Dear user, **you tell me** what linker to use! kthxbye."

Finding the linker (3)

- applications that launch binaries from sandboxes need **some knowledge of ABIs**
  - library? ("dear binutils, what sort of binary is this?")
  - system service? ("what linker should this binary use?")

Future work☺

- initial approach: `ffexecve(2)` (specify linker, binary by FD)
- final approach: **directly-executable linker**

Directly-executable linker

```
Usage: /libexec/ld-elf.so.1 [-h] [-f <FD>] [--] <binary> [<args>]

Options:
  -h        Display this help message
  -f <FD>   Execute <FD> instead of searching for <binary>
  --        End of RTLD options
  <binary>  Name of process to execute
  <args>    Arguments to the executed process
```

as before: `fork(2)`, open directory descriptors, `setenv(3)`, `cap_enter(2)`

the new bit: `fexecve(the_linker, args + [ "-f", the_binary ], environ)`

```
r319135 | kib      | 2017-05-29 MFC direct execution mode for rtld.
r318431 | jonathan | 2017-05-17 Allow rtld direct-exec to take a file descriptor.
r318380 | kib      | 2017-05-16 Pretend that there is some security when executing in direct mode.
r318313 | kib      | 2017-05-15 Make ld-elf.so.1 directly executable.
```

Demo: run(1)

The opposite of sudo(8)

- find the ELF interpreter
- find a binary
- execute it in a sandbox

This solves all our problems... right?

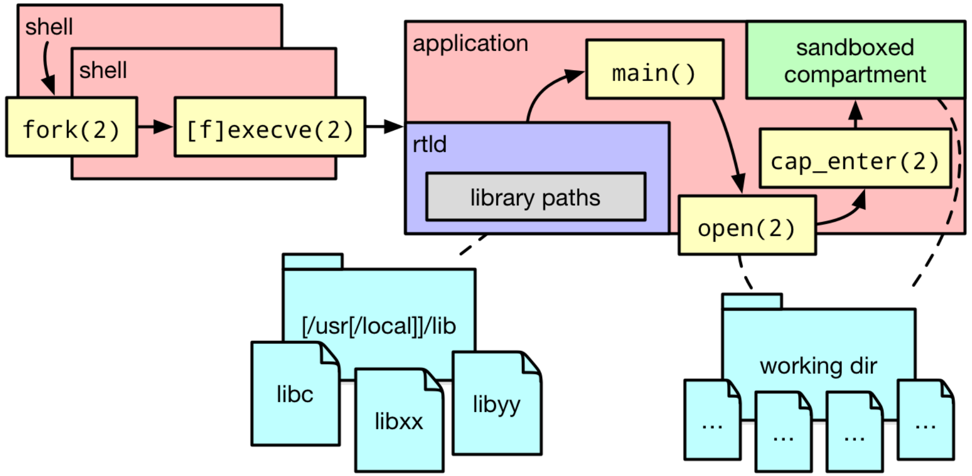
Er, not quite.

```
int rtld = open("/libexec/ld-elf.so.1", O_RDONLY);
int binary = open(name, O_RDONLY);

char *args[argc + 4];
args[0] = strdup(name);
args[1] = "-f";
asprintf(args + 2, "%d", binary);
args[3] = "--";
args[argc + 3] = NULL;

for (int i = 0; i < argc - 1; i++)
    args[i + 4] = argv[i + 1];

fexecve(rtld, args, environ);
```

note: error handling removed for space reasons

Linking within compartments

The story so far:



- most applications need dynamic libraries*
- run-time linker is "just" code in a process
  - same address space / security domain
  - runs before `main`, opens needed libraries

Actual linking can happen at run-time, even in capability mode...
but libraries cannot be `open(2)`'ed from capability mode!

---

* Other than the FreeBSD-derived MacOS, which doesn't support statically-linked binaries...

Finding libraries

How it normally works:

(see `find_library` at `libexec/rtld-elf/rtld.c:1586`)

- `DT_RPATH` (with rules about DSO, `DT_RUNPATH`...)
- `LD_LIBRARY_PATH`
- `DT_RUNPATH`
- ldconfig hints (with rules aroudn `-z nodefaultlib`)
- `STANDARD_LIBRARY_PATH` (`/lib32:/usr/lib32`, `/lib/casper:/lib:/usr/lib`...)

... followed by `open(2)` ... which isn't allowed in capability mode!

Shared libraries in capability mode

- r267678: `LD_LIBRARY_PATH_FDS`
  - like `LD_LIBRARY_PATH`, but with file descriptors
  - directory descriptors for `/lib`, `/usr/lib`, `/usr/local/share/myapp/plugins`…
  - then `openat(2)`, then `fstat(2)`…

The story so far

We can run RTLD

RTLD can find libraries

RTLD can run binaries

Profit???

Not quite!

```
int rtld = open("/libexec/ld-elf.so.1", O_RDONLY);
int binary = open(name, O_RDONLY);

char *args[argc + 4];
args[0] = strdup(name);
args[1] = "-f";
asprintf(args + 2, "%d", binary);
args[3] = "--";
args[argc + 3] = NULL;

for (int i = 0; i < argc - 1; i++)
    args[i + 4] = argv[i + 1];

fexecve(rtld, args, environ);
```

note: error handling removed for space reasons

The story so far (2)

Libraries are not enough:

```
$ cc run.c -o run && ./run /bin/cat /etc/passwd
cat: /etc/passwd: Not permitted in capability mode
```

Also need support for traditional resource access

Accessing file resources

Existing applications like to use:

- `access(2)`
- `stat(2)`
- `open(2)`

… none of which are allowed!

We could rewrite the application to assume it will be given a directory descriptor and use `openat(2)`, etc. … but that wouldn't be very oblivious!

libpreopen

Transparent filesystem proxying

- libpreopen's `struct po_map` maps virtual paths to capabilities
- `libc` wrappers provide Capsicum-aware versions of, e.g., `open(2)`:
  - `LD_PRELOAD` libpreopen* to take precedence over system calls†
  - take given (absolute) path, search through `struct po_map`:
    - on success: translate `"/usr/local/share/my_app/foo.conf"` ⇒ `(FD 3,"foo.conf")`; these can be passed to `accessat(2)`, `openat(2)`, `statat(2)`…
    - no suitable pre-opened path: translate to `(FD -1,NULL)`, return error

---

* This works in capability mode iff `libpreopen.so` is reachable via `LD_LIBRARY_PATH_FDS` — if not, we **always fail closed**.
† System calls are defined as *weak symbols* in libc to allow overriding.

libpreopen (2)

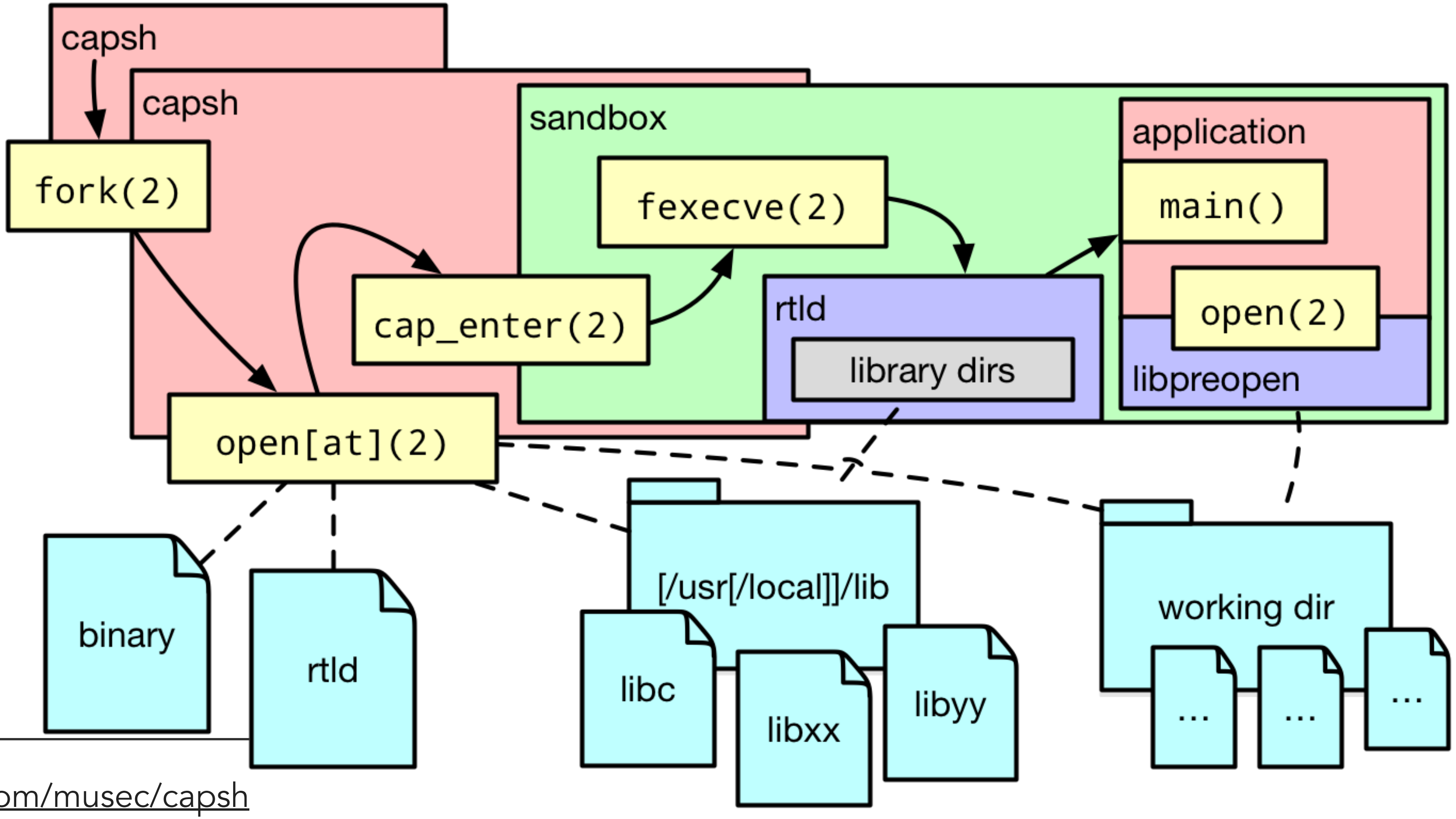
But where does a `po_map` come from?

Our overall objective: **launch an unmodified application from a sandbox**

Who pre-opens files and directories?
It's the responsibility of the thing (process) doing the launching.

- (in most cases) `fork(2)`
- pre-open any required resources
- populate a `struct po_map`
- pack the `po_map` into POSIX SHM
- set `LD_LIBRARY_PATH_FDS`, `LD_PRELOAD`
- set `LIB_PO_MAP`
- `fexecve(2)` the linker
- let `libc` wrappers unwrap/use `LIB_PO_MAP`

capsh: a capability-enhanced shell



Available from github.com/musec/capsh

capsh **status**

Where are we today?

Not a real shell: only usable as `capsh <args>` for direct execution.

Not very sophisticated: we can do a little more than `echo`, but not much*!

But: it does execute execute **unmodified software**.

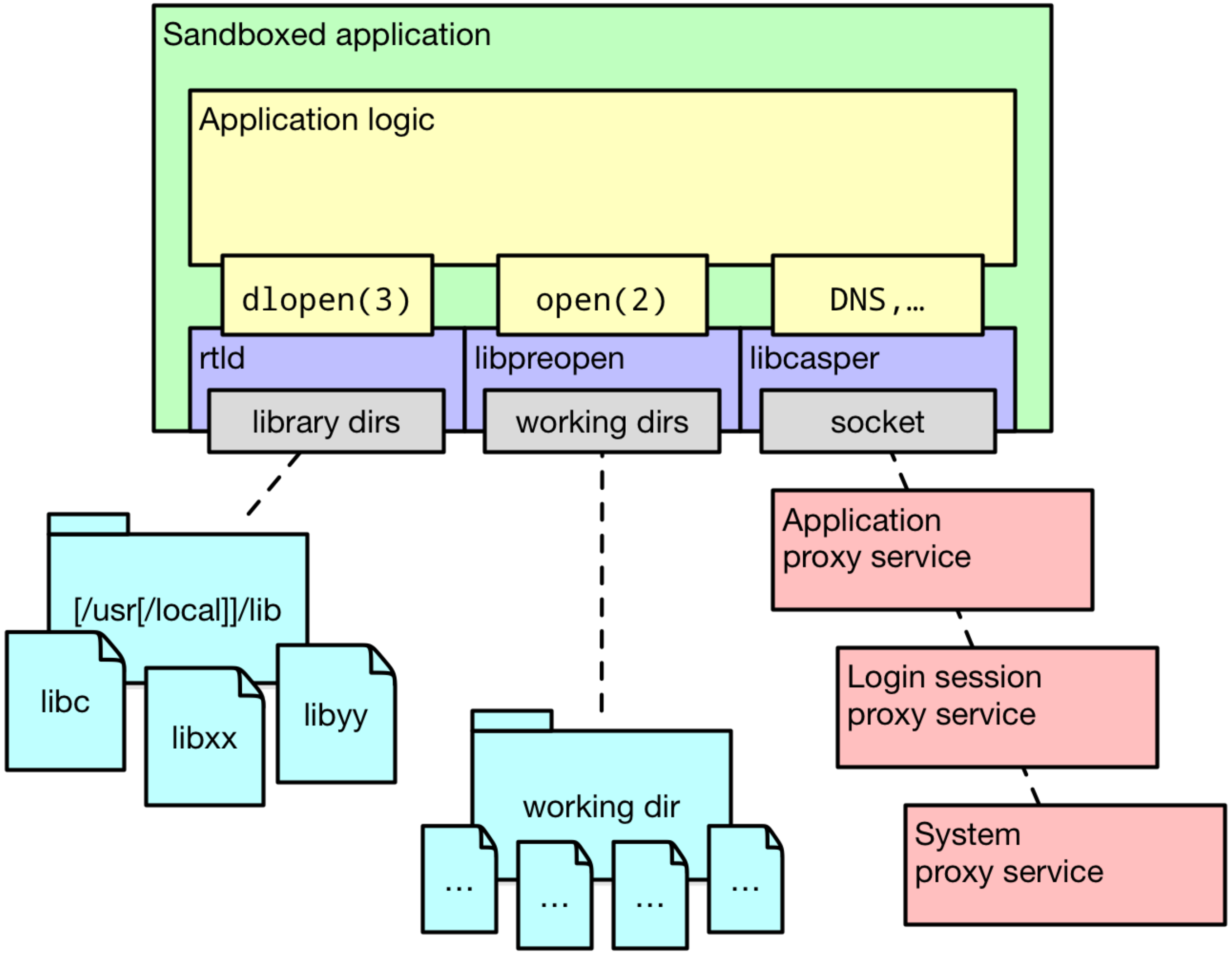
We can run `/usr/bin/true`, `banner`, `pom`, `primes`...

The goal

Sandboxing programs by default

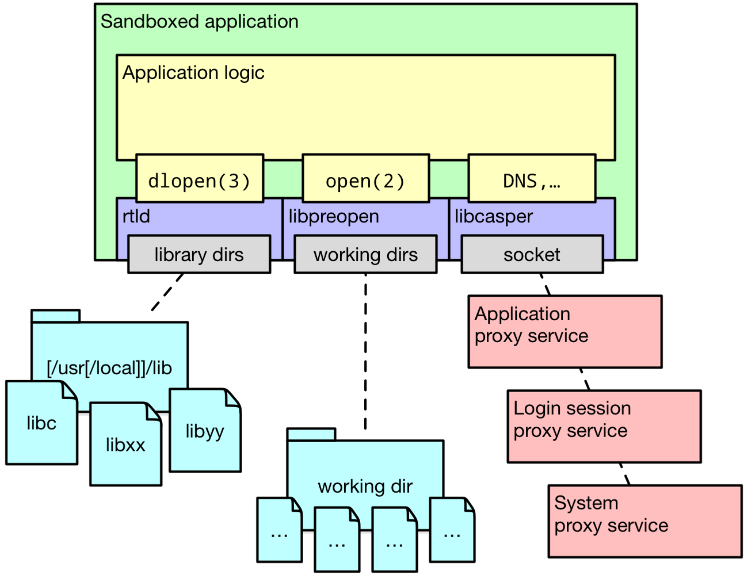
Services required:

- static pre-opened files:
  - CLI arguments
  - package policies
- dynamic provisioning:
  - UI interaction
  - user/system policies



Static policies*

```
services:
  exec:
    paths: [ /usr/local/llvm39, /bin, /usr/bin ]

  filesystem:
    - root: /usr/local/llvm39
      preopen: true        # we always need the llvm39 dir
      rights: [ read, seek, lookup ]
    - root: /usr/local/bin
      globs: [ "clang*", "ll*" ]
      rights: [ read, seek, exec ]

  network:
    https:
      hostname: llvm-crashreporter.freebsd.org
      certificate-policy: # ...
```

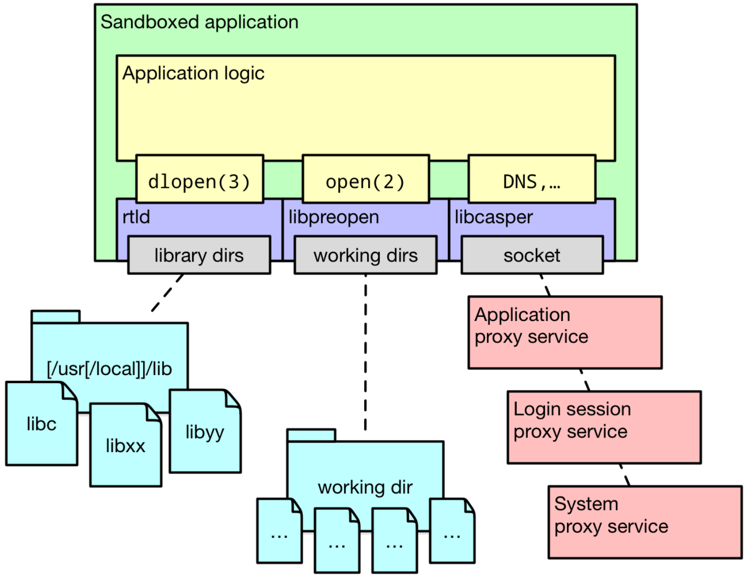


---

* Policy syntax is **suggestive of future directions**: this stuff **doesn't exist yet**.

Dynamic services

- application-level services
  - TLS handling
  - worker processes
- session-level services
  - D-Bus
  - UI powerboxes
  - user data provisioning
- system-level services
  - names, syslog..
  - shared data and configuration



Towards oblivious sandboxing

From a Capsicum sandbox, we can:

- pre-open libraries and resources
- run RTLD directly
  - use library directory FDs
  - map, run binary
- wrap ambient-authority system calls
  - retrieve FDs from anonymous shared memory
  - convert `access(2)` to `accessat(2)`, `open(2)` to `openat(2)`…
- provide access to **named system services**

Conclusion

- Capsicum provided kernel-level foundation for **principled, coherent compartmentalization**
- new work provides application-level foundation for:
  - running **unmodified** applications
  - providing application services
- stage set for deeper exploration of **oblivious sandboxing**
- movement towards applications that **just work** and are **secure by default**