SEE TEXT ONLY

By Jonathan Anderson,
Stanley Godfrey, and Robert N. M. Watson

TOWARD OBLIVIOUS SANDBOXING WITH Capsicum

Capsicum [Wat+10] is a framework for principled, coherent compartmentalization of FreeBSD applications.

It is principled in that it draws from a rich history in computer security concepts such as *capabilities*, tokens that authorize their bearers to perform actions such as read from a file (using a file descriptor as a token very like a capability) or call a method (using an object reference as a capability). Capsicum is coherent in that it applies clear, simple security policies uniformly across applications. It is not possible—as can be the case in other schemes—to restrict an application's access to one set of operations while leaving equivalent operations available for use. When we describe Capsicum as providing principled, coherent *compartmentalization*, we mean that it allows applications to break themselves up into compartments that are isolated from each other and from other applications. Just as privacy-friendly companies put their users' data encryption keys out of their own reach, Capsicum allows applications and their compartments to give up certain abilities in order to protect other compartments, other applications, and—ultimately—their users.

However, a significant limitation of Capsicum today is that it only works when applications **voluntarily** give up the right to perform certain actions. It works with applications that understand Capsicum and that have been modified to take advantage of it; up to now, Capsicum has provided no mechanisms for confining applications without their cooperation. This is our long-term goal: to put applications into *sandboxes* without needing to modify the applications themselves, such that

any vulnerabilities in an application that are exploited by attackers can have their damage contained within an application's memory and outputs rather than granting full access to all of a user's data and activities. In this article, we describe recent and ongoing work to advance this agenda, pursuing the vision of protecting ourselves from vulnerable applications whether they like it or not.

A Taste of Capsicum

Today, Capsicum allows applications to protect themselves, other applications, and their users from themselves via two mechanisms: *capability mode* and *capabilities*.

Capability Mode

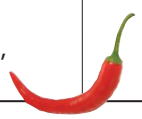
Capability mode is a way of *confining* a process to stop it from accessing any namespaces that are shared between processes such as the filesystem namespace, the process identifier (PID) namespace, socket-address namespaces, and interprocess communication (IPC) namespaces (System V and POSIX). Its only access to files and other system objects is mediated through *capabilities*, which are described below. Once a process enters capability mode, it loses all ability to access these namespaces and it cannot leave capability mode (nor can any processes forked from it from that point on). This first crucial Capsicum concept creates a strong isolation: if a process cannot open any resources *and holds no capabilities*, it cannot affect the operation of other processes or leave behind any side effects. For applications to do any useful work, however, some communications and/or side effects are necessary. The key, from a security perspective, is to ensure that these interactions occur in a **controlled** way.

Capabilities

The second crucial concept in Capsicum—the idea of *capabilities*—allows applications to be granted access to potentially-shared resources in a controlled way. Capabilities, as described by Dennis and Van Horn in 1966 [DV66], consist of an identifier or address for an object together with a description of the operations that may be performed on that object using the capability. In Dennis and Van Horn's model, computation occurred within a protection domain ("sphere of protection") and accessed resources using an index into a supervisor-maintained C-list. This model of userspace performing limited operations on system resources via indices within kernel-maintained arrays should sound familiar to current practitioners: capabilities were central to the PSOS design [FN79], which heavily influenced the design of Multics [SCS77], which directly inspired Unix and its *file descriptors* [RT78]. However, in the journey from capabilities to modern file descriptors, something was lost in translation: the rigorous, principled focus on capabilities as monotonic encodings of security policy (i.e., having a set of allowed operations that can be reduced, but never augmented). The Unix focus on user IDs within filesystems naturally led to an expansion of the role of file descriptors, such that the set of operations permitted via a file descriptor included operations that are not expressed in the descriptor itself, but are based on rights encoded in the filesystem. For example, most Unix-like systems will allow a user to `open(2)` a read-only file in a read-only mode and then `fchmod(2)` it to be a writable file (see Listing 1). This is an example of how file descriptors place more emphasis on the *identity* aspect of capabilities than on *operations*.

Capsicum capabilities restore to file descriptors a rigorous focus on allowed operations. In FreeBSD 10 and later, every file descriptor is associated with a set of explicit rights that define which operations may be performed on that file descriptor. Outside of capability mode, file descriptors are opened with all rights to preserve traditional file descriptor semantics. When descriptors are explicitly limited or derived from other capabilities (e.g., via `openat(2)` relative to a directory capability), only those operations explicitly permitted by the capability may be performed using that capability. There are capability rights that correspond to existing `open(2)` flags such as `CAP_READ` and `CAP_WRITE`, but there are also rights that make formerly implicit privileges a matter of explicit policy, such as `CAP_SEEK`, `CAP_MMAP`, `CAP_FTRUNCATE`, and `CAP_FCHMOD`.

Capabilities are monotonic: the holder of a capability may always give up rights associated with that capability with `cap_rights_limit(2)`, but new rights can never be added to an existing capability. If a process in capability mode requires access that it does not already have, it must acquire it from another process that rightfully has the authority to delegate it. Like all file descriptors,



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

Listing 1: File descriptors allow operations beyond those directly expressed in a descriptor—in this case, a read-only file descriptor is used to modify properties of the file.

capabilities may be delegated from one process to another via inheritance or IPC, but because of their strong monotonic guarantees, capabilities can be delegated with confidence: a capability with `CAP_READ` can be shared with an untrusted process in capability mode with the certain knowledge that it cannot be used to `fchmod(2)` the file or perform any actions other than `read(2)`. (The experimental implementation of capabilities in FreeBSD 9 involved additional indirection: a `struct` capability that contained a set of rights and a pointer to an underlying `struct` file.)

Why Capsicum?

Capsicum is a *principled* and *coherent* way to construct compartments within applications. It is principled in that it relies on a conceptually rigorous mechanism to enforce clear security policies that can be composed naturally due to the monotonicity of capabilities. As Linden observed in his 1976 survey of OS security and reliability [Lin76], "a single general protection mechanism that is used without exception is better than a rigid one that has many exceptions." Capabilities map naturally to many program requirements, as today's software is already structured around reference-like access to files as objects with explicit methods.

Capsicum's coherence is due to its "deep in the kernel" implementation and its simple-yet-complete definition of capabilities and capability mode. Attempts to provide "shallow" system call wrapping, exemplified by Provos's `systrace` [Pro03], are unable to provide the atomicity guarantees that are critical for security policy evaluation: policy enforcement is weakened when the objects and operations seen by the kernel are subject to races with the security policy that authorizes the operations on those objects. Attempts to allow userspace processes to define their own sets of "safe" system calls, as in Linux's `seccomp-bpf` [Cor12] or OpenBSD's `pledge(2)` mechanism [Chi15], can lead to incoherent security policies that disallow one type of access to system resources while permitting an equivalent type of access via another path. This can lead to an exposure of not-quite-sandboxed processes to malicious data under a false sense of security. In contrast, Capsicum's kernel-defined capability mode is both sufficient and necessary to express isolation from global OS namespaces, a coherent and easy-to-understand security policy.

The simplicity and dependability of capability mode helps application developers use it effectively, so long as their applications fit the simple model of first acquiring resources and then computing on them. (The section called Toward Oblivious Sandboxing on page 21 contains a discussion of more complex models.) Capsicum also requires no privilege for an application to compartmentalize itself, in contrast to approaches that rely on mandatory access control (MAC) such as SELinux [LS01] or AppArmor [BM06] or approaches that rely on Linux namespaces [Bie06]. Such approaches are accessible to applications with system-administrator support and/or `setuid` helper binaries, but Capsicum can be applied to any program compiled and run by any developer.

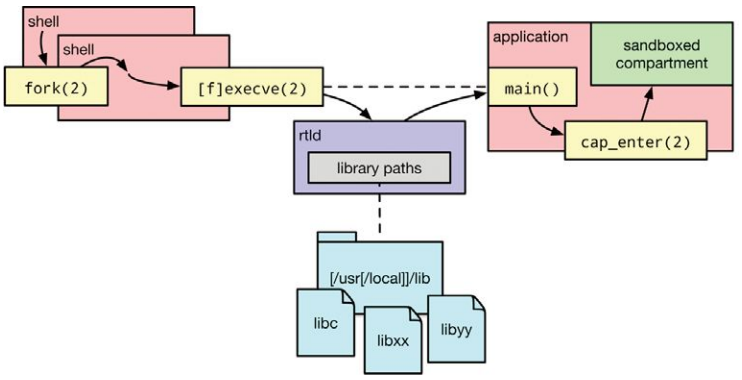


Fig. 1: Applications can compartmentalize themselves by acquiring static resources from global namespaces before calling `cap_enter(2)`.

Compartmentalizing with Capsicum

To take advantage of Capsicum, applications (including forked children of main application processes) call `cap_enter(2)` before they are exposed to any untrustworthy data, e.g., network requests. Once a process compartmentalizes itself, it can begin performing potentially dangerous operations such as parsing network traffic or user input in the confidence that any malicious exploitation will lead to, at worst, a corruption of the process's explicit outputs (files, network responses, etc.).

This self-sandboxing approach works well when a process is able to open all of the resources that it needs before entering capability mode. The most obvious resources to be opened before compartmentalization are files and sockets, but in a modern binary, even something as simple as `cat(1)` or `echo(1)`, dynamic linking means that a set of shared libraries must also be loaded before compartmentalization. As shown in Figure 1, the runtime linker runs within a process, sharing its address space and, on startup, its main thread. Most simple applications only rely on the

runtime linker to find all their libraries on startup, after which it's possible to enter capability mode and let the runtime linker fix up dynamic symbols as required from already-open library files.

Other static resources that a sandboxed compartment might need to access include explicit files, which can be pre-opened by an application before calling `cap_enter(2)` or accessed via a pre-opened *directory descriptor* and then accessed by the compartment with `openat(2)` and related system calls (`fstatat(2)`, `renameat(2)`, etc.). Implicit resources include locale files that are required by many libc functions, but these can also be pre-opened and their results cached. More dynamic resources require a connection with the outside world so that a sandboxed process can ask an unsandboxed process to access new resources. This mode of operation is commonly employed in compartmentalized applications on many platforms, including Web browsers and—perhaps surprisingly—all applications downloaded on MacOS through the Mac App Store, where all applications requesting access to user files must go through a trusted UI called a *powerbox* [App16; Yee04]. To help with these more dynamic applications' requirements, FreeBSD includes the `libcasper` (Capability Services Provider [Zab16]) mechanism to proxy access to named services, some of which (e.g., `system.dns`) are provided by the system itself.

With pre-opened capabilities, locale cacheing, directory descriptors, libcasper, and external proxies at their disposal, many applications are able to compartmentalize themselves with Capsicum. However, this only applies to applications whose authors are willing to spend the effort required to adopt Capsicum features and adapt their applications for compartmentalization. More impact could be attained if we were able to transparently sandbox applications without imposing any additional requirements on their authors, i.e., if we could employ *oblivious sandboxing*.

Toward Oblivious Sandboxing

With this goal of oblivious sandboxing in mind, work on Capsicum has been progressing across FreeBSD's runtime linker, a new library, and a new capability-aware (but not feature-complete) shell. Recently, some of these components have begun to bear fruit, leading to an exciting new development: the first transparent sandboxing of unmodified applications within Capsicum's capability mode. The applications that can be executed today in this matter are very simple, but they execute without access to global namespaces and without any modification: rather than sandboxing themselves, they begin life in a sandbox.

`exec(2)` Without a Name

The traditional approach by which one application executes another is to first `fork(2)` a child process and then, from within that new process, to call `exec(2)` and start running the new program. The `exec(2)` system call cleans up the memory mappings of the current process, closes any file descriptors that have an `O_CLOEXEC` flag set (preserving all other open files, together with current environment variables) and transfers control to the new application. In order to do this, `exec(2)` must first find the binary to be executed. Looking up a binary by name—as in the traditional `exec(2)` call—would require access to the global filesystem namespace; this is not permitted in capability mode. Instead, FreeBSD provides the `fexecve(2)` system call to execute a binary as specified by a file descriptor (which can be a capability) rather than by pathname. On Linux, `fexecve(3)` is a `glibc` function that uses a file descriptor to look up a symbolic link in `/proc/self/fd`, then calls `exec(2)` with that path name. The Linux implementation of Capsicum required the addition of an `execveat(2)` system call with true file descriptor semantics [Dry14].

When `fexecve(2)` runs, it inspects the file passed to it to determine its type (ELF executable, script, `a.out` executable, etc.) and passes it to an *image activator* within the kernel (see Figure 2). Image activators parse various types of executable files and start running them; ELF image activators (32- and 64-bit) encode knowledge of runtime linkers and how to find them in the filesystem.

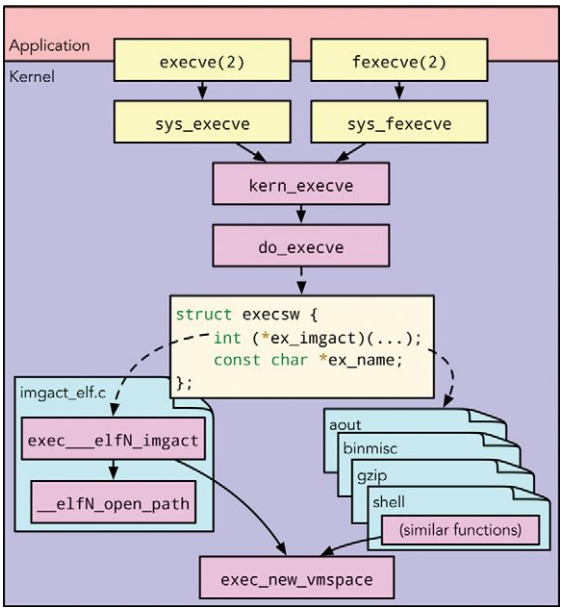


Fig. 2: The FreeBSD kernel contains a number of *image activators* to parse various types of executable files and start running them.

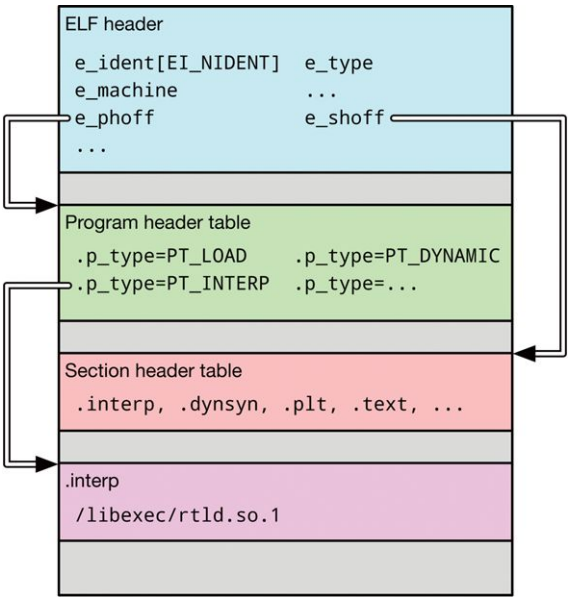


Fig. 3: The ELF file format includes an explicit path to the runtime linker that is expected to be used as an interpreter for the binary file.

Although various ABIs for various platforms have default runtime linker names, binaries can also explicitly encode a path to a preferred runtime linker, as shown in Figure 3. Whether discovered via a program's internal format header or the default of an image activator, runtime linkers are described using path names. In a conventional `exec(2)` or `fexecve(2)` invocation, the runtime linker would be looked up using this path and executed first, before the main function of the new application (illustrated in Figure 1). Inside a Capsicum compartment, however, access to global filesystem namespaces is not permitted, so another approach is required.

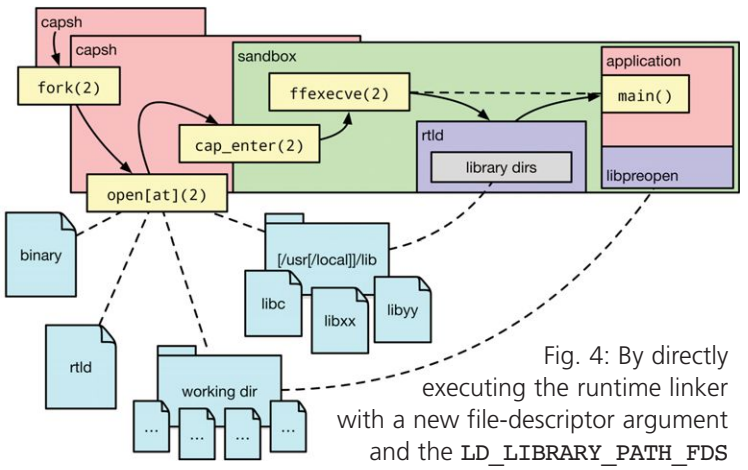


Fig. 4: By directly executing the runtime linker with a new file-descriptor argument and the `LD_LIBRARY_PATH_FDS` environment variable, `capsh` can execute an untrusted program from within a Capsicum sandbox. This application starts running without the ambient authority to access global namespaces.

FreeBSD's runtime ELF linker has recently been modified to support direct execution, i.e., on FreeBSD 12-CURRENT one can run `/libexec/ld-elf.so.1` as an executable—the usage string as of writing is shown in Listing 2. This ability has long been present in Linux's `ld-linux.so.2`, but it was not required on FreeBSD until motivated by the requirements of oblivious sandboxing. Now, direct execution has been implemented together with the ability for the runtime linker to accept as a command-line argument a file descriptor to link and run—these changes will be present in FreeBSD 12 and 11.1. Together, they allow a process in capability mode that has capabilities for a runtime linker and a binary to `fexecve(2)` the linker, preserving open files including the binary's capability, and to specify via command-line arguments which file the linker is to execute. The net result, shown in Figure 4, is that the specified binary is executed using the specified runtime linker. However, without access to shared libraries stored in the filesystem, the runtime linker is not able to satisfy the dynamic code-loading requirements of the application. That requires an additional mechanism: library path descriptors.

Shared Libraries in Capability Mode

As described previously, essentially all modern executable files are dynamically linked and therefore depend on access to shared libraries for their correct execution. In fact, the FreeBSD-derived MacOS does not support statically linked binaries: ABI guarantees are maintained only at the interfaces of core system libraries rather than the kernel [App11]. When applications compartmentalize themselves with `cap_enter(2)`, they can do so after the dynamic runtime linker has discovered library dependencies and `mmap(2)`'ed them in place for later linking. If an application starts running before these libraries have been opened, however, the linker is unable to satisfy the requirements of dynamic symbol resolution.

Traditionally, the dynamic runtime linker has supported a number of environment variables that control its behavior. One example, `LD_LIBRARY_PATH`, informs the linker of a set of directories in which additional libraries may be found. For example, a program may set `LD_LIBRARY_PATH` to an internal directory that contains application code or dynamically-loadable plugins. We have extended FreeBSD's ELF runtime linker to support an additional environment vari-

able, `LD_LIBRARY_PATH_FDS`. This variable allows the specification of directories containing shared libraries that will be searched in exactly the same way as `LD_LIBRARY_PATH_FDS`, but with one crucial difference: instead of a colon-separated list of pathnames, this variable contains a colon-separated list of directory descriptors. Since environment variables and open files are both preserved across the `fexecve(2)` boundary—unlike, e.g., memory mappings—this allows a parent process to open a set of library descriptors, set `LD_LIBRARY_PATH_FDS` and then enter capability mode and `fexecve(2)` the runtime linker itself with access to its shared library directories. Combined with the direct-execution support described in the previous section, this allows a dynamically linked application to be executed from within a sandbox.

```
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
```

Listing 2: Current usage options for FreeBSD's ELF runtime linker.

`libpreopen`: Transparent Filesystem Proxying

It is useful to be able to execute code from within a sandbox, including code that is dynamically linked, but that is insufficient for the goal of oblivious sandboxing. Most Unix applications are written using common system calls such as `access(2)`, `stat(2)`, and `open(2)` to test and gain access to files within the filesystem. These system calls inherently require access to the global filesystem namespace, so they are not permitted in capability mode. It is possible to write applications to use `fstatat(2)`, `openat(2)`, etc., relative to an explicit base directory, but many extant applications have not been written this way. To achieve oblivious sandboxing, applications must be confined and resources must be provided *without* application modification.

We can interpose a runtime translation of system calls from capability-unsafe to capability-safe variants using the runtime linker: the `LD_PRELOAD` environment variable allows us to name libraries that should be loaded before any others. When libraries are named in this environment variable without absolute paths, the runtime linker searches through its default search paths for libraries of the given names, but not before consulting `LD_LIBRARY_PATH_FDS`, making `LD_PRELOAD` a capability-mode-compatible directive. If we provide an implementation of a `libc` function such as `open(2)`, our implementation will take precedence over that of libc, where system calls are defined as "weak" symbols. Our implementation of `open(2)` can translate the provided path argument into an `openat(2)` call, but by itself, this adaptation accomplishes nothing: the application will still attempt to look up a path name that is not relative to a directory, only this time it will do so using the `openat(2)` system call instead of `open(2)`.

The final component that is required to adapt filesystem namespace operations is a set of pre-opened directory descriptors that other operations can be performed relative to. This is the core abstraction provided by `libpreopen`, a library that is—for the moment—maintained independently of FreeBSD. (`libpreopen` can be downloaded and built from https://github.com/musec/libpreopen). `libpreopen` provides a `struct po_map` type that is used to map directory names to directory descriptors, flags, and capability rights, as well as `libc` wrappers that can look up and query a default `po_map`. For example, when `libpreopen`'s implementation of `open(2)` is passed an absolute path, it looks up the default `po_map`, which can be specified as data packed into an anonymous shared memory segment. FreeBSD's implementation of POSIX shared memory allows a constant "path" of `SHM_ANON` to be passed to `shm_open(2)`, creating a shared memory segment that can be manipulated by file descriptor but does not appear in the regular POSIX shared memory namespace, making it safe for use in capability mode. `libpreopen` can open such a shared memory segment, specified by file descriptor in an environment variable, and unpack its data into an in-memory `struct po_map` object. From that `po_map`, the wrapper queries, "do you have a directory descriptor whose name is a prefix of this absolute path?" If such a descriptor exists in the map, the absolute path is decomposed into a directory descriptor and a relative path from that descriptor. These two elements can then be passed to `openat(2)`.

`libpreopen` provides a mechanism for a process in capability mode to access filesystem resources, as long as some directory descriptors have been pre-opened and stored in a way that is accessible to the library. Opening such descriptors, building a `struct po_map` representation, packing it into anonymous shared memory, and storing the file descriptor of the shared memory segment in an environment variable are all the

responsibility of the process spawning the sandboxed child. One example of such a process is `capsh`, the capability shell.

`capsh`: A Capability-enhanced Shell

The final major component of Capsicum-based oblivious sandboxing is a program to transparently sandbox unmodified—and unsuspecting—applications. A proof-of-concept implementation of such a program is `capsh`, a shell that uses capabilities and capability mode to sandbox applications. Hosted independently of FreeBSD for the time being, capsh allows users to execute simple unmodified applications from within a Capsicum sandbox. (The `capsh` source code is available at https://github.com/musec/capsh.) In its current implementation, the program is hardly a shell at all: it has no interactive mode, only executing a single program per invocation. It also only supports simple applications with statically-enumerable resource requirements. Nonetheless, programs that fit into this model can be executed with sandboxing from inception without program modification.

`capsh` works by tying together the pieces of the oblivious sandboxing puzzle described above. It finds and opens a user-specified executable file, together with a runtime linker to interpret it. It opens library directories and stores them in the `LD_LIBRARY_PATH_FDS` environment variable. It manipulates pre-opened directory descriptors, storing them in `struct po_map` types provided by `libpreopen` and making them available to child processes via shared memory and environment variables. It then enters capability mode via `cap_enter(2)` and uses `fexecve(2)` to execute the runtime linker. The net result is that an unmodified application starts running from within a Capsicum sandbox, as shown in Figure 4.

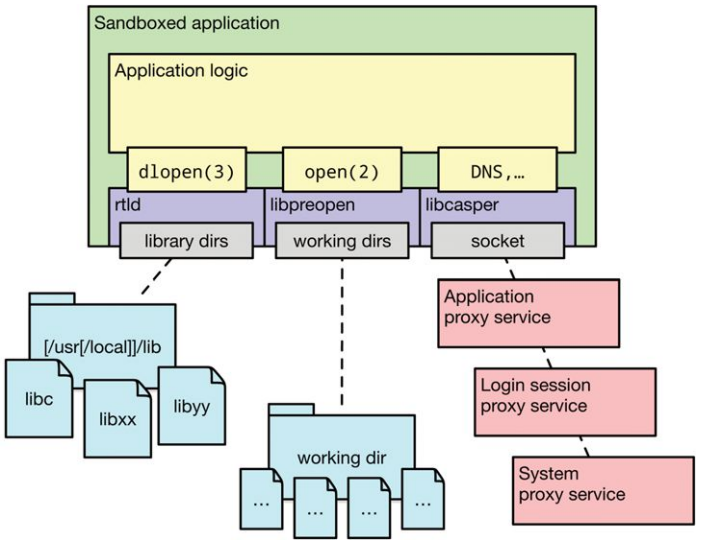


Fig. 5: A fully-sandboxed application can only access files and services that have been delegated to it: the runtime linker can find libraries in pre-opened directories using `LD_LIBRARY_PATH_FDS`, `libpreopen` can operate on files in pre-opened working directories—translating global-namespace–dependent system calls such as `open(2)` to relative variants such as `openat(2)`—and `libcasper` can proxy access to namespaces of external servers.

Oblivious Sandboxing

Applications running under capsh can access only those resources that are explicitly delegated to them; there can be many sources of policy as to which resources ought to be delegated. Users running `capsh` implicitly specify policy when they type command-line arguments to sandboxed applications: the presence of a filename as an argument may be an indication that the file should be pre-opened before the application is executed or that permission to open the file via a proxied mechanism such as `libcasper` should be granted. Users may also drive policy decisions implicitly through interactions with a graphical user interface, as in the powerbox model described in the earlier section on Compartmentalizing with Capsicum, and future work on `capsh` may connect to existing models of graphical login sessions to provide this mode of policy elicitation. Policy may also be derived from files packaged together with applications: a compiler's package metadata may specify where its standard library is located, and `capsh` could pre-open that directory with a read-only capability. More sophisticated policy files could describe limited interactions that are permitted with named libcasper services, leading to a more general model of a Capsicum application as shown in Figure 5. Additional exploration of mechanism is also possible: of particular interest to the authors of this article is the possibility of applying LLVM-based transformation to libraries and applications that embed LLVM bitcode, allowing a transparent rewriting of function calls to capability-mode–friendly APIs without needing `LD_PRELOAD` for interposition.

Conclusion

Capsicum, a principled and coherent design for software compartmentalization, has taken strides in recent days toward a new security model. Changes in the FreeBSD ELF runtime linker, together with developments in `libpreopen` and `capsh`, allow simple applications to be sandboxed *transparently*, without any participation on the part of the application. These foundational ele-

ments have now set the stage for a deeper exploration of how programming models interact with the need for compartmentalization and to what extent software can be sandboxed *obliviously*, operating as normal with no requirement to know whether it is operating inside of a sandbox or not. A broader availability of oblivious sandboxing will allow us to move to a FreeBSD in which applications "just work" *and* are secure by default. •

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.

Jonathan Anderson is an Assistant Professor in Memorial University of Newfoundland's Department of Electrical and Computer Engineering, where he works at the intersection of operating systems, security, and software tools such as compilers. He is a FreeBSD committer and is always looking for new graduate students with similar interests.

Stanley Godfrey is a graduate student at Memorial University of Newfoundland. His research interest is in Capsicum, FreeBSD, and operating system security. He did his undergraduate studies at Helsinki Metropolia University of Applied Sciences, majoring in software development and graduating with a Bachelor of Engineering in Information Technology.

Dr. Robert N. M. Watson is a Senior Lecturer (Associate Professor) at the University of Cambridge Computer Laboratory, where he leads research spanning operating systems, security, and computer architecture. He is a FreeBSD developer, member of the FreeBSD Foundation Board of Directors, and coauthor of *The Design and Implementation of the FreeBSD Operating System (second edition)*.

[App11] Apple Inc. "Technical Q&A QA1118: Statically linked binaries on Mac OS X," *Apple Developer Guides*. https://developer.apple.com/library/content/qa/qa1118. (2011)

[App16] Apple Inc. "App Sandbox in Depth," *Apple Developer Guides*. https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AppSandboxInDepth/AppSandboxInDepth.html. (2016)

[Bie06] Biederman, Eric W. "Multiple Instances of the Global Linux Namespaces," *Linux Symposium Volume One*, pp. 101–111. https://www.landley.net/kdocs/ols/2006/ols2006v1-pages-101-112.pdf. (2006)

[BM06] Bauer, Mick. "Paranoid Penguin: An Introduction to Novell AppArmor," *Linux Journal* 2006.148, p. 13. ISSN: 1075-3583. URL: https://dl.acm.org/citation.cfm?id=1149839. (2006)

[Chi15] Chirgwin, Richard. "Untamed pledge () aims to improve OpenBSD security: Monkey with the wrong permissions, your program dies," *The Register*. https://www.theregister.co.uk/2015/11/10/untamed_pledge_hopes_to_improve_openbsd_security. (2015)

[Cor12] Corbet, Jonathan. "Yet another new approach to seccomp." https://lwn.net/Articles/475043/. (2012)

[Dry14] Drysdale, David. "syscalls,x86: Add execveat() system call," *Linux Kernel Mailing List*. https://lkml.org/lkml/2014/5/27/147. (2014)

[DV66] Dennis, Jack B. and Van Horn, Earl C. "Programming semantics for multiprogrammed computations," *Communications of the ACM* 9.3, pp. 143–155. DOI: 10.1145/365230.365252. (1966)

[FN79] Feiertag, R. J. and Neumann, Peter G. "The foundations of a provably secure operating system (PSOS)," NCC '79: *Proceedings of the 1979 AFIPS National Computer Conference*. DOI: 10.1109/AFIPS.1979.116. (1979)

[Lin76] Linden, Theodore. "Operating System Structures to Support Security and Reliable Software." ACM Computing Surveys (CSUR) 8.4, pp. 409–445. DOI: 10.1145/356678.356682. (1976)

[LS01] Loscocco, Peter A. and Smalley, Stephen D. "Meeting Critical Security Objectives with Security-Enhanced Linux," *Proceedings of the 2001 Ottawa Linux Symposium*. https://lwn.net/2001/features/OLS/pdf/pdf/selinux.pdf. (2001)

[Pro03] Provos, Niels. "Improving Host Security with System Call Policies," *Proceedings of the 12th USENIX Security Symposium*. http://niels.xtdnet.nl/papers/systrace.pdf. (2003)

[RT78] Ritchie, O. M. and Thompson, K. "The UNIX time-sharing system," *Bell System Technical Journal* 57.6, pp. 1905–1929. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1978.tb02136.x. (July 1978)

[SCS77] Schroeder, Michael D.; Clark, David D.; and Saltzer, Jerome H. "The Multics kernel design project," SOSP '77: *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*. ACM. DOI: 10.1145/800214.806546. (1977)

[Wat+10] Watson, Robert N. M.; Anderson, Jonathan; Laurie, Ben; and Kennaway, Kris. "Capsicum: practical capabilities for UNIX," *Proceedings of the 19th USENIX Security Symposium*. https://www.usenix.org/legacy/events/sec10/tech/full_papers/Watson.pdf. (2010)

[Yee04] Yee, Ka-Ping. "Aligning security and usability," *IEEE Security and Privacy Magazine* 2.5, pp. 48–55. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.64. (2004)

[Zab16] Zaborski, Mariusz. "libcasper(3)," *FreeBSD Library Functions Manual*. https://www.freebsd.org/cgi/man.cgi?query=libcasper. (2016)