


Capsicum

Update 2019 by Mariusz Zaborski



FreeBSD is a general-purpose operating system. One of its goals is to provide a secure environment for its users. To meet that goal, among others, FreeBSD has introduced a Capsicum framework. Every day, the FreeBSD community works to make improvements, and in this article, we will take a look at how Capsicum has changed over the past year.

The Capsicum framework provides tight isolation between processes. When a process enters the capability mode (sandbox), it doesn't have access to any global namespace. The process can enter this state by using `cap_enter(1)` syscall. The difference between Capsicum and other popular sandbox frameworks is that Capsicum focuses on the capabilities of processes instead of filtering syscalls. In Capsicum, descriptors represent capabilities. The descriptors are perfect for capabilities since we can duplicate (`dup(2)`), send them to other processes (through Unix Domain Socket), or revoke them (`close(2)`).

Another component of the framework is capability rights. These allow us to limit capabilities (file descriptors) even further. The descriptor can be limited to be readable (`CAP_READ`) or writable (`CAP_WRITE`) using a `cap_rights_limit(2)` syscall. The capabilities can be limited even if they can reposition their offset in the file (`CAP_SEEK`). Capsicum allows the purpose of the descriptor to be defined very precisely. There are over 50 capabilities right now.

For a more detailed description of the framework, we recommend you look at other issues of the *FreeBSD Journal* where you will find more detailed descriptions of Capsicum [1] [2].

Casper

Other primitives necessary to understand the Capsicum environment are Casper and Casper services. Casper provides functionality not available in

capability mode through convenient APIs, making Capsicum more practical. Casper accomplishes this by utilizing the privileged separation of the process. After creating a new Casper instance, the initial process forks. Next, the unprivileged process enters capability mode. If the sandboxed application wants to perform some forbidden operation in Capsicum, it has to delegate this work to Casper. Casper communicates with a sandboxed application using a straightforward API called *libnv* (or *nvlist*).

Let's assume that an application wants to resolve the DNS name. Before entering capability mode, it can create Casper services—*system.dns*. Each time it needs to resolve a DNS, instead of calling the *getaddrinfo* function, it calls the *cap_getaddrinfo*. The Casper function has precisely the same API as standard *libc*, except it has one more argument that takes Casper connection.

The Casper services API is straightforward. Thanks to this, the sandboxing application is much more comfortable. Without Casper, all applications developers would need to reimplement the privileged separation routines to sandbox new application.

Introducing fileargs

The *fileargs* service allows programs to access a filesystem. This service makes it relatively easy to sandbox applications. The *wc(1)* and *head(1)* are examples of applications which depend on it. The *fileargs* service doesn't provide a full filesystems service. Its primary focus is to address the applications which, as an argument, takes a vector of files to open. Nevertheless, this service may also be used for different scenarios.

The service is essential for two reasons. First, as described above, it allows a new applications to be sandboxed. Secondly, because this is the first service in which the API doesn't reflect the *libc* functions.

Currently, the *fileargs* services provides two main functions—it allows the opening of files and to get their status. The *fileargs_open/fileargs_fopen* functions allow the opening of a file from a given path and the *fileargs_fstat* function provides the capability to gather the status. The primary function is *fileargs_init*. This function initializes the Casper service.

```
fileargs_t *fileargs_init(int argc, char *argv[], int flags, mode_t mode, cap_rights_t *rightsp, int operations);
```



The *argc* and *argv* arguments are just vectors within the files that applications should be able to open. The *flags* and *mode* argument aren't any different from an argument to open. Those arguments describe how the file should be opened and in which mode it should be created. Next is the list of capabilities that the newly opened file should keep. The last argument is which operations in service are permitted. For now, services define two operations, open (*FA_OPEN*) and *lstat* (*FA_LSTAT*).

The *fileargs_cinit* function is very similar to the *fileargs_init* function. The only difference is that *fileargs_cinit* reuses already existing Casper instances. In the case of a *fileargs_init* function, the Casper service creates new instances.

```

@@ -79,6 +85,8 @@ main(int argc, char *argv[])
+ ..... char *ep;
+ ..... off_t bytecnt;
+ ..... int ch, first, linecnt, eval;
+ ..... fileargs_t *fa;
+ ..... cap_rights_t rights;

+ ..... linecnt = -1;
+ ..... eval = 0;
@@ -106,13 +114,22 @@ main(int argc, char *argv[])
+ ..... argc -= optind;
+ ..... argv += optind;

+ ..... fa = fileargs_init(argc, argv, O_RDONLY, 0,
+ ..... cap_rights_init(&rights, CAP_READ, CAP_FSTAT,
CAP_FCNTL));
+ ..... if (fa == NULL)
+ .....     errx(1, "unable to init casper");
+
+ ..... caph_cache_catpages();
+ ..... if (caph_limit_stdio() < 0 || caph_enter_casper() < 0)
+ .....     err(1, "unable to enter capability mode");
+
+ ..... if (linecnt != -1 && bytecnt != -1) ¶
+ .....     errx(1, "can't combine line and byte counts");
+ ..... if (linecnt == -1)
+ .....     linecnt = 10;
+ ..... if (*argv != NULL) {
+ .....     for (first = 1; *argv != NULL; ++argv) {
- .....         if ((fp = fopen(*argv, "r")) == NULL) {
+ .....             if ((fp = fileargs_fopen(fa, *argv, "r")) ==
NULL) {
+ .....                 warn("%s", *argv);
+ .....                 eval = 1;
+ .....                 continue;

```

Listing 1. The patch for sandboxing `head(1)`.



Listing 1 presents a patch for sandboxing the `head(1)`. The patch is straightforward. All we needed to do was initialize the Casper service with the right capabilities, pass `argv` and `argc` to it, and change the open function to the Casper version. Finally, we entered the capability mode.

It is worth noting that the fileargs service API is still considered to be experimental and may change.

Improving `cap_sysctl`

The `cap_sysctl` services allow us to interact with the kernel state. In the original implementation, we introduced the `cap_sysctlbyname` function. However, when the sandboxing process of `rsol(8)` and `rsold(8)` began, it became clear it was not enough. The `sysctl` can be referred to in two ways—by its text representation and by its numeric representation.

The process of sandboxing those applications motivates developers to extend `cap_sysctl`. The `cap_sysctl` and `cap_sysctlnametomib` functions were introduced. The first allows the manipulation of values through numeric representation. The

second makes it possible to fetch a numeric representation of Management Information Base (MIB) of sysctl from a given string representation. The interface of those two functions is very similar to their predecessor. The only change is that the Casper functions expect additional argument with the connection to the Casper service.

The interface extension also meant that the limitation functions should be reworked. We introduced a new interface for the cap_sysctl service:



- `cap_sysctl_limit_init` - initializes the limitation structures
- `cap_sysctl_limit_name` - allows the limiting of single MIBs through name representation
- `cap_sysctl_limit_mib` - allows the limiting of single MIBs through numeric representation
- `cap_sysctl_limit` - sets the limits on given Casper service instances and frees all underlying structures.

In Listing 2, we have an example of using it. First, we create a Casper instance with `cap_init`, and Casper service with `cap_service_open`, which is the standard

```
cap_channel_t *capcas, *capsysctl;
const char *name = "kern.trap_enotcap";
cap_sysctl_limit_t *limit;
int value;
size_t size;

/* Open capability to Casper. */
capcas = cap_init();
if (capcas == NULL)
    err(1, "Unable to contact Casper");

/* Use Casper capability to create capability to the system.sysctl
service. */
capsysctl = cap_service_open(capcas, "system.sysctl");
if (capsysctl == NULL)
    err(1, "Unable to open system.sysctl service");

/* Close Casper capability, we don't need it anymore. */
cap_close(capcas);

/* Create limit for one MIB with read access only. */
limit = cap_sysctl_limit_init(capsysctl);
(void)cap_sysctl_limit_name(limit, name, CAP_SYSCTL_READ);

/* Limit system.sysctl. */
if (cap_sysctl_limit(limit) < 0)
    err(1, "Unable to set limits");

/* Fetch value. */
if (cap_sysctlbyname(capsysctl, name, &value, &size, NULL, 0) < 0)
    err(1, "Unable to get value of sysctl");

printf("The value of %s is %d.\n", name, value);
```

Listing 2. The main page example of usage cap_sysctl limits.

method. Next, we initialize *sysctl limits*. We limit our service only to one *sysctl--kern.trap_enotcap*. We can refer to it only with a text representation. The *CAP_SYSCTL_READ* also means that an application can only fetch the value of this *sysctl*. At the end of Listing 2, the program fetches that value.

Private Services

Mark Johnston did some more exciting work. When he was sandboxing *rtsock(8)* and *rtsockd(8)*, he implemented a private Casper service dedicated only to those two applications. The *rtsockd(8)* is a daemon program to send ICMPv6 Router Solicitation messages on the specified interfaces. The service is application specific, so there was no reason to make it publicly available. This approach may get us to the point where some services will be installed from the ports/packaging systems. His work allows us to see that the Casper service may also be used in different environments for process separation.

The *rtsock(8)* and *rtsockd(8)* used Casper to create a service for sending Router Solicitation messages on a raw ICMPv6 socket. This is accomplished by the *cap_sendmsg* service. Another private service, *cap_script*, is used to spawn and collect the status of scripts required by the *rtsockd* daemon. The third and last service implemented for this program is *cap_llflags*. This service is responsible for fetching the flags for the link-local IPv6 address on the specified interface.

The *rtsockd(8)* is an example of a sandboxed program within the Casper service that didn't require the implementing of general wild services.

The Super Capsicumizer 9000

Concealed behind this funny name is a small diamond. The Super Capsicumizer 9000, or just Capsicumizer, is an open-source project that has tried, with success, to implement the sandbox launcher that uses Capsicum. [3] *AppArmor* inspires the Capsicumizer. *AppArmor* is a mandatory access control system that allows process access to be limited. Its confinement is provided via profiles loaded into the kernel, typically on boot. The profiles can be managed by the administrator of the system and describe which resources the application should have access to.

Capsicumizer is based on the profiles as well. The difference is that instead of loading profiles to the kernel, we run Capsicumizer in userland and allow Capsicum to handle limitations of the process. The patterns are defined using UCL syntax.

The Capsicumizer uses a *libpreopen* library to all the resource pre-opened directory descriptors and using them in capability-safe libc wrappers. Thanks to that library, our application will have all the capabilities it requires.

Currently, the limitation of Capsicumizer is that it allows only limited resources to the filesystems one. Unfortunately, defining or preconfiguring network access is not supported.

The Capsicumizer is an exciting project that already allows us to sandbox applications without modifying them. For now, it is limited only to the filesystem. It would be interesting to see it combined with the Casper. With such a



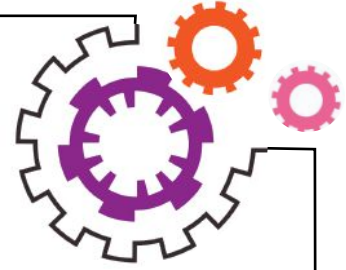
```
#!/usr/bin/env capsicumizer

run = "/usr/local/bin/gedit";

access_path = [
    "$HOME",
    "/usr/local",
    "/var/db/fontconfig",
    "/tmp",
];

library_path = [
    "/lib",
    "/usr/lib",
    "/usr/local/lib",
    "/usr/local/lib/gvfs",
    "/usr/local/lib/gio/modules",
    "/usr/local/lib/gedit",
];

# gedit does not need any extra preloads
# this is just an example
ld_preload = [
    "libgobject-2.0.so"
];
```



Listing 3. An example of configuration from Capsicumizer 3000 which allows to sandbox gedit.

combination, we would be able to sandbox a lot of applications without changing their code.

Summary

The FreeBSD Capsicum framework is still under development but is already widely used. The improvements to the Casper services, especially *cap_fileargs*, opens a whole new set of applications that can be easily sandboxed. Projects like Capsicumizer can get us to the point where administrators wanting to separate single process will not need to touch the code to achieve their aim. ●

BIBLIOGRAPHY

- [1] Jonathan Anderson, Stanley Godfrey, Robert N. M. Watson, *Toward Oblivious Sandboxing with Capsicum*
- [2] Mariusz Zaborski, *FreeBSD Journal* issue May/June 2018, "Capsicum—Just apply me!"
- [3] <https://github.com/myfreeweb/capsicumizer>

Mariusz Zaborski is a QA & Dev manager at Fudo Security. He has been the proud owner of the FreeBSD commit bit since 2015. Mariusz's main areas of interest are OS security and low-level programming. At Fudo Security, Mariusz leads a team that is developing the most advanced solutions to monitor, record, and control traffic in IT infrastructure. In 2018, Mariusz organized the Polish BSD User Group. In his free time, he enjoys blogging—<https://oshogbo.vexillum.org>.