SEE TEXT ONLY

By Mariusz Zaborski

Capsicum

JUST APPLY ME!

Applying sandboxing always seems challenging and there is a historic reason for this. For example, using linux `seccomp(2)`–one of the oldest sandboxing techniques–requires a lot of effort. The simplest mode of `seccomp(2)` — *SECCOMP_SET_MODE_STRIC*– restricts a program to an unusable state. On the other hand, the most popular mode–`SECCOMP_SET_MODE_FILTER`–is very hard to apply and maintain and it can also easily betray us. In this article, we discuss a sandboxing technique built in FreeBSD called Capsicum. If the reader is interested in a deeper comparison between different popular sandboxing techniques, the author recommends a few articles [1] [2] [3]. Here we will discuss what Capsicum is, the tooling we have and, most importantly, how we can use it in our applications.

Overview of Capsicum

The Capsicum infrastructure can be divided into two parts:

- Tight sandboxing
- Capability rights

Tight sandboxing means we don't have access to any global namespaces. With a global namespace, we are referring to a limited area within an operating system. These areas have a set of names that allows the unambiguous identification of an object [4]. To simplify it, we can't operate on objects using their identifier like a filepath. This tight sandbox still allows us to operate with objects by using their handlers. In UNIX-like operating systems, we have one universal handler—descriptor—and we can operate on the file using a file descriptor. The same with a process. In this mode, we can't operate on a process using PID, but we can operate on it using a process descriptor [5]. In Capisucm, this mode is called Capability mode. In FreeBSD, we have one simple syscall to enter this mode—`cap_enter(2)`.

With Capsicum we go one step further by permitting limitations on the descriptors. If we have a descriptor that we know will be used only to read, we can set a specific right (`CAP_READ`) that will ensure this particular descriptor will be read-only. If there is an attempt to write on this particular descriptor, it will fail. We refer to these limitations as capability rights. It is always possible to limit descriptors further, but we can't extend the capability rights of a given descriptor. That means if we have a descriptor with the `CAP_READ` and `CAP_WRITE` capability and at some point we decide we don't want to read any more of this descriptor, we can drop the capability. For obvious reasons, it doesn't allow you to extend them. In FreeBSD, we have a special function—`cap_rights_limit(2)`—which allows us to limit descriptors. Currently we have 79 rights that allow granularity limit handlers.

Thanks to capability mode and capability rights we can ensure that a process has access only to objects that it really needs. This eliminates the ambient authority problem, where any process has access to all user data. If an attacker would exploit a tool like `grep(1)`, `patch(1)` or even `cat(1)`, he would gain access to all user data. An attacker

could also create a new connection to an arbitrary server and send those files to it. In a Capsicum world, even if an attacker could exploit any of those tools, he would have read-only access to a few files on the disk. He wouldn't be able to overwrite any important data or send them over to the network.

Extending Capabilities

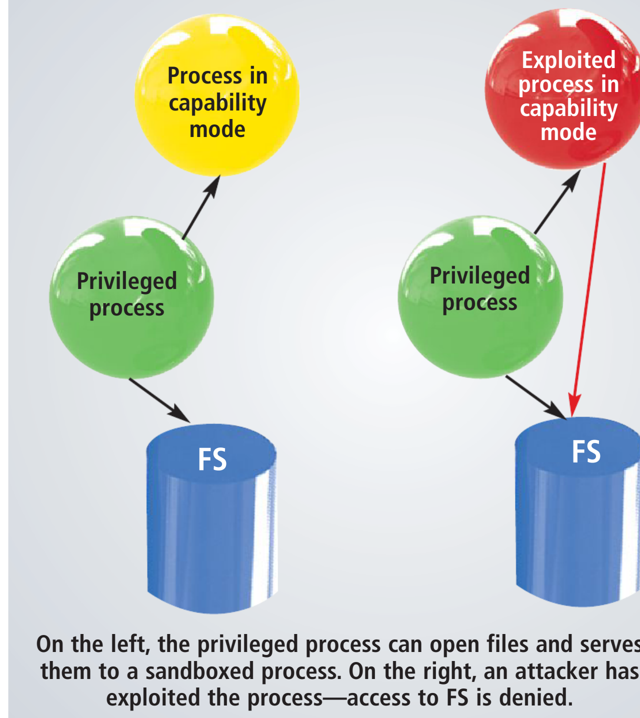
There are two methods by which we can obtain capabilities:

- initialization phase
- obtain them from another process

In the first case, we pre-open all connections, files etc. needed in our application before entering capability mode and we can only operate on those capabilities. This method is ideal for small applications that have limited functionality.

The second method is to use another process that already has the capability rights to the object we want. Because all objects are represented by descriptors, we can easily pass them from one process to another using UNIX domain sockets. If one process has a descriptor to talk with a server, it can pass it to another process, in which case both processes can operate on the single descriptor.

A very commonly used pattern with Capsicum is having two processes. The first has ambient authority so it can operate on behalf of the user, and the second is sandboxed. The ambient authority process is used only for simple things like opening a list of files or connecting to a server. The sandboxed process does all the complicated logic of the application ex. parsing. The privileged process is connected with a sandboxed process over a UNIX domain socket and simply serves the next file descriptors to be parsed. If an attacker exploited our parser (which is very common),



On the left, the privileged process can open files and serves them to a sandboxed process. On the right, an attacker has exploited the process—access to FS is denied.

Thank you!

The FreesBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.

Are you a fan of FreeBSD? Help us give back to the Project and donate today! **freebsdfoundation.org/donate/**

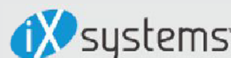
Please check out the full list of generous community investors at freebsdfoundation.org/donate/sponsors

Uranium

Iridium

Platinum

Gold



Silver

Microsoft

STORMSHIELD

he wouldn't have any access to any other part of the system—but only to another read-only file.

Capsicumizing

Capsicumizing is a funny name that we use to describe the process of sandboxing existing applications. Currently in FreeBSD we have around 60 sandboxed applications. From very simple ones such as `yes(1)`, to some more complicated like `jot(1)`, to much more complex ones like `tcpdump(1)` and `ping(8)`. There is a decent number of sandboxed applications, but we still work all the time to sandbox more of them. The full list of applications and current progress can be followed on the Capsicum wiki [6]. In Example 1, we have a patch for capsicumizing `cmp(1)`, a program that compares two files. In this patch, we use a method to pre-open all files that we need before entering capability mode. `cmp(1)` is working on two descriptors `fd1` and `fd2` and both have capability `CAP_FSTAT` and `CAP_FCNTL`, which respectively allow us to get a file status using the `fstat(2)` function and `fcntl(2)` for file control. (The `cmp(1)` uses the `fdopen(3)` function that requires `fcntl(F_GETFL)`. In the original patch, we also limit the amount of `fcntls(2)` using `cap_fctnls_limit(2)`, but this is beyond the scope of this article.) We also give descriptors the `CAP_MMAP_R` capability which allows us to map file into the memory. At the end, we limit the `stdout` descriptor and pre-cache the native language support data (NLS). Finally, we enter the capability mode.

It is very straightforward. For this particular application we didn't even need to reorganize the code because all the files were already opened in one place before parsing, and so the initialization phase was really easy to find. Then we limited some of the descriptors and entered capability mode.

Capsicum Helpers

In Example 1, we have one change that is not obvious when you see it for the first time—calling `catopen(3)` to cache NLS data. Normally when we call the `err(3)` function for the first time, it goes to file system and opens a file for NLS. Unfortunately, in compatibility mode this can't be done because it can't open the file.

There is one more known issue with pre-caching things in `libc` which deals with manipulating time. The functions like `localtime(3)`--when we call them for the first time—pre-cache the current time zone of the machine and they do

```
  --- usr.bin/cmp/cmp.c
+++ usr.bin/cmp/cmp.c
@@ -68,2 +71,5 @@ main(int argc, char *argv[])
        const char *file1, *file2;
+       cap_rights_t rights;
+       unsigned long cmd;
+       uint32_t fcntls;

@@ -148,2 +154,19 @@ main(int argc, char *argv[])

+       cap_rights_init(&rights, CAP_FCNTL, CAP_FSTAT, CAP_MMAP_R);
+       if (cap_rights_limit(fd1, &rights) < 0 && errno != ENOSYS)
+               err(ERR_EXIT, "unable to limit rights for %s", file1);
+       if (cap_rights_limit(fd2, &rights) < 0 && errno != ENOSYS)
+               err(ERR_EXIT, "unable to limit rights for %s", file2);
+
+       cap_rights_init(&rights, CAP_FSTAT, CAP_WRITE, CAP_IOCTL);
+      if (cap_rights_limit(STDOUT_FILENO, &rights) < 0 && errno != ENOSYS)
+               err(ERR_EXIT, "unable to limit rights for stdout");
+
+       /*
+       * Cache NLS data, for strerror, for err(3), before entering capability
+        * mode.
+        */
+       (void)catopen("libc", NL_CAT_LOCALE);
+
+       if (cap_enter() < 0 && errno != ENOSYS)
+               err(ERR_EXIT, "unable to enter capability mode");
+
        if (!special) {
```

Example 1. Capsicumizing `cmp(1)`, simplified patch proposed by Conrad E. Meyer.

that only the first time.

Due to these unclear behaviors, Capsicum helpers were created. Capsicum helpers are a set of small inline functions that allow us to pre-cache things i.e. time zones and NLS data.

The man page for them is also a good place to document this unclear behavior. So for caching time zones and NLS, we will find two functions: `caph_cache_tzdata(3)` and `caph_cache_catpages(3)`.

Another use of Capsicum helpers is to reduce the amount of code needed in our application. If we go back to Example 1, we can see we are limiting `stdout`. The question is how many applications need to limit `stdio`—probably most of them—which is why the `caph_limit_stdin(3)`, `caph_limit_stdout(3)`, `caph_limit_stderr(3)` and `caph_limit_stdio(3)` functions were introduced. Those functions allow us to limit single descriptors with the most common rights, and to limit others with a specific one for the application. If the default limitations are enough for our program, we can simply call the `caph_limit_stdio(3)` function which will limit all of them in a single command.

Another very common pattern is calling the `cap_enter(2)` function, and if the function fails, then checking `ERRNO`. The purpose of this is to check if entering capability mode failed because something happened or if our operating systems just don't support it. In the first case, an application should stop working at this point. In the second case, it should still run because our whole system doesn't support it. At first, this check can be counterintuitive, and it's very uncommon to not check `ERRNO`. This is why we are working on presenting another function `caph_enter(3)` which will hide this check [7].

Debugging Tools

When sandbox is added to an application, a developer cannot notice certain conditions of a program. An application could use a library which a developer doesn't know very well. For example, if an application is using a library and this library uses a random number generator by opening `/dev/random` if possible, otherwise it can use an insecure random generator. If a developer does not notice this behavior by analyzing the code, this can lead to introducing new bugs while sandboxing an application. This is why providing useful debugging tools is one of the biggest challenges in building a sandboxing mechanism. For Capsicum in FreeBSD, we have two tools:

- `ktrace(1)/kdump(1)`,
- gdb with *TRAPCAP*.

During sandboxing, a developer can run a program with `ktrace(1)` and check if no `ECAPMODE` or `ENOTCAPABLE` has been returned. This might cause some issues for a developer, as sometimes it can be hard to know which called function failed. It's also very hard to cover all possible run paths.

In our previous example, this library function which opens `/dev/random` is used with only one particular option provided in the program. However, a program has many options and it could be very easy to miss. This is also why regression tests are so important. Unfortunately, in this case we would need to run a whole test suite under `ktrace(1)` and analyze its output. In Example 2 below, we have a sample output of the `ktrace(1)`.

```
802 random  CALL  cap_enter
802 random  RET   cap_enter 0
802 random  CALL  openat(AT_FDCWD,0x400877,0<O_RDONLY>)
802 random  CAP   restricted VFS lookup
802 random  RET   openat -1 errno 94 Not permitted in capability mode
802 random  CALL  exit(0)
```

Example 2. Result of a `kdump(1)` on a program that enters capability mode and tries to open `/dev/random`.

Another way of analyzing our program is to use a new debugging feature for Capsicum which was implemented by Konstantin Belousov under FreeBSD Foundation sponsorship. Thanks to this, when `ECAPMODE` or `ENOTCAPABLE` is returned, a kernel will issue `SIGTRAP`. As a result, we will get a core dump exactly at the moment the error occurred. This makes it harder to overlook some errors as our program will abort and we will notice it while we run it. A core dump also provides more information about the state of the process when the error occurred. We can enable this feature using `sysctl kern.trap_enotcap` (globally in the system). If there is a need to enable it per-process, we can use `proctl(2)`. For a system to be able to generate a core dump in capability mode, we also need to set `sysctl capmode_coredump`, otherwise programs in sandbox are unable to create them.

Nvlist Library

We've already discussed some methods of obtaining more capabilities in our process. One method is to receive them from another process. To make it

easier to split programs between privileged and unprivileged processes, Caspsicum developers also introduced a very easy IPC library—`nvlist`. This library is based on the list containing pairs (name, value), and it allows us to keep many primitives like: numbers, strings, binary and bools.

However, one of the most special things is that it also allows us to keep descriptors on the list. Furthermore, it provides functions to send and receive `nvlist` over the socket. All of this has been designed to allow for separation of processes and for capiscumizing them more easily. `nvlist` as a container also exists in the kernel and is used by some drivers (ex. `ixl`). The implementation used exactly the same code as user land—it doesn't contain primitives that don't exist in the kernel (like descriptor or socket). In Example 3, we can see a simple use of `nvlist`. The author recommends considering `nvlist` as a candidate for a serialization library because the implementation and use of it is very straightforward. If you are interested in the use-cases of `nvlist`, see man page (`nv(9)`) or some external materials [9] [10] [11].

```
nvlist_t *nvl;

nvl = nvlist_create(0);
nvlist_add_string(nvl, "first", "foo");
nvlist_add_number(nvl, "second", 1234);

/*
 * What is also very interesting in nvlist is that we need to check
 * only last operation on nvlist. If one of previous adding would fail
 * we would know that at any point.
 */
if(nvlist_send(sock,nvl)<0){,
        fprintf(stderr, "Unable to send nvlist.\n");
        exit(1);
}
```

Example 3. Simple use of the `nvlist`. Add two values and send it over the network.

Casper Overview

While splitting programs in the privileged and unprivileged process, we will notice some common patterns. For example, many network tools need to have access to the DNS server. In capability mode, we can't connect to the server directly and need some other process that can talk with it. To simplify this and reduce the amount of the code needed for all applications, the Casper library was created. This library provides us a set of services like:

- `system.dns` - service for getting network host entry
- `system.grp` - service for group database operations
- `system.pwd` - service for password database operations
- `system.random` - service for getting entropy
- `system.sysctl` - service for getting or setting system information
- `system.syslog` - service for syslog

When creating a Casper instance, it forks from the original process and this requires the creation of Casper services (`cap_init(3)`) before entering capability mode. We can also receive a service from another process which has a service. All services are well documented, and in the man pages we can find examples of how to use them.

Currently we are working on one more service—`system.fileargs`. The goal of this service is to provide a simple tool for sandboxing applications, which, as an argument, takes a list of files. This service will provides descriptors via a similar API to `open(2)`. Thanks to the interface, applying it should be straightforward for existing applications. Even though this service is still under development, the project agreed that it will be initially treated as experimental [8]

We also have a plan to implement other services such as:

- `system.login` - a service for accessing the login class capabilities database
- `system.tls` - a service for creating a safe connection using TLS/SSL
- `system.socket` - a service for creating network connections
- `system.configuration` - a service for fetching unified configuration

These are all currently only ideas.

Casper and `dhclient(8)`

One of the new services is `system.syslog`. Let's discuss for a moment why we created it.

```
Starting devd.
Starting dhclient.
pid 336 (dhclient), uid (65): Path `/var/crash/dhclient.65.0.core'
failed on initial open test, error = 2
pid 336 (dhclient), uid 65: exited on signal 5
Trace/BPT trap/etc/rc.d/dhclient: WARNING: failed to start dhclient
Starting syslogd.
```

Example 4. `dhclient(8)` core dumping during start.

```
void syslog(int priority, const char *message, ...);
void vsyslog(int priority, const char *message, va_list args);
void openlog(const char *ident, int logopt, int facility);
void closelog(void);
int setlogmask(int maskpri);
```

Example 5. `syslog` API

While booting an operating system with **`kern.trap_enotcap sysctl`** enabled, we noticed that **`dhclient(8)`** was core dumping. In Example 4, we present this situation.

After analyzing this program, it turned out that dhclient was using **`syslog`** to report its status. If we look one more time at Example 4, we see that the **`syslogd(8)`** was started after **`dhclient(8)`**. This is the standard problem of the chicken and the egg. **`syslogd(8)`** sometimes needs a network to be configured and **`dhclient(8)`** needs a **`syslogd(8)`** to report status. Historically, we decided to run **`dhclient(8)`** before running **`syslogd(8)`**. What is worth noting is that **`dhclient(8)`** tried to connect to **`syslogd(8)`** before entering capability mode because the server did not exist and yet it failed. Surprisingly, each time the program was not connected, a syslog function tried to connect to it! Of course, because we are now running in Capsicum, we will never be able to establish a connection. So, to solve this issue, we decided to introduce a new Casper service syslog. It tries to connect to **`syslogd(8)`** and if it fails, it will try the next time there is something to report. It's also worth noticing that the syslog API (shown in Example 5) doesn't report any issues if something is wrong.

Summary

Sandboxing base systems is an ongoing process. We have introduced many tools which should lower the entrance barrier for new people wishing to use Capsicum.

Capsicumizing is a very good way to learn about operating systems--how they work, how they interact, and how some libraries behave. There are still many small programs waiting to be capsicumized. This can be a great lesson on operating systems--especially for people who dream about becoming FreeBSD developers. ●

MARIUSZ ZABORSKI is a lead software developer at *Wheel Systems*. He has been a proud owner of the FreeBSD commit bit since 2015. Mariusz's main areas of interest are OS security and low-level programming. At *Wheel Systems*, Mariusz leads a team that is developing the most advanced solution to monitor, record and control traffic in an IT infrastructure. In his free time, he enjoys blogging (http://oshogbo.vexillium.org).

BIBLIOGRAPHY

[1] J. Anderson, *A Comparison of Unix Sandboxing Techniques*, FreeBSD Journal Sept/Oct 2017

[2] P.Dawidek, M.Zaborski, *Sandboxing with Capsicum*, ;login: issue:December 2014, Vol. 39, No. 6

[3] M.Zaborski, *Capsicum and Casper - a fairy tale about solving security problems*, AsiaBSDCon 2016 http://oshogbo.vexillium.org/pdf/AsiaBSDcon2016.pdf

[4] http://en.wikipedia.org/wiki/Namespace

[5] Robert N.M. Watson, Jonathan Anderson, Ben Laurie, Kris Kennaway, *Introducing Capsicum: Practical Capabilities for UNIX, 2010*

[6] FreeBSD wiki, Capsicum page, https://wiki.freebsd.org/Capsicum

[7] Introduce **`caph_enter()`**, FreeBSD phabricator, https://reviews.freebsd.org/D14557

[8] Introduce **`system.fileargs`**, FreeBSD phabricator, https://reviews.freebsd.org/D14407

[9] Introduction to **`nvlist`** part 1, http://oshogbo.vexillium.org/blog/42/

[10] Introduction to **`nvlist`** part 2 - **`dnvlist`**, http://oshogbo.vexillium.org/blog/43/

[11] Introduction to **`nvlist`** part 3 - simple traversing, http://oshogbo.vexillium.org/blog/45/