Capsicum and Casper - a fairy tale about solving security problems

Mariusz Zaborski
Wheel Systems, FreeBSD
oshogbo@FreeBSD.org

1 Introduction

Every year the list of security bugs is shocking. Operating systems start to innovate new mitigation techniques like ASLR or canneries which only makes exploiting harder but many in many cases still possible. We also try to mitigate on a lower level like the NX bit in the CPU, but researchers again have found a method of bypassing this technique by the so-called return oriented programing.

Another way of handling security problems is by using a sandbox environment. Some researches try to use a sandbox on a big scale, like running different virtual machines for different tasks, while others try to use them on a smaller scale like in a process. In this article we focus just on two of them. Process sandboxing should be easy, flexible, scalable and lightweight. Two of the most recent work in BSD world deserve for our attention - Capsicum and Pledge.

These sandboxing techniques have their strengths and weaknesses. Process sandboxing can sometimes be too restrictive for some of the challenges found in the real world and this is the moment when Casper enters the scene.

2 Pledge

Pledge (historically *tame*) is a sandbox framework created for OpenBSD project [1]. Pledge assumes that we can divide any program into two parts. One of which is the initialization phase in which all the preparations are performed (e.g. arguments parsing) and the other one which is the main loop, containing the actual operations (e.g. parsing a file). Pledge allows us to decide which system calls can by be accessed by so called ”promises”. Each promise gives us a different set of system calls with different options. At the moment Pledge implements around 24 promises, in which we can count: [2]

- stdio - one of the most common, allows us to allocate memory, and perform basic io operations (like using stdout),
- rpath - allows functions which only cause read-only effects on the filesystem (eg. open(2) on an existing file with read flag only),
- wpath - allows system calls which may cause write-effects on filesystems,
- cpath - allows functions which may create new files,
- proc and exec - allow to fork and execute another program.

Pledge always allows you to reduce permissions by simply calling *pledge(2)* again.

Statistics show that Pledge is very easy to use to sandbox application. Over 400 programs in OpenBSD use pledge.

At listing 1 we have an example of how *cat(1)* program was sandboxed using *pledge(2)*. The program has access to stdio (*cat(1)* needs stdout to print result) and have access to all filesystem as long as it is read only. A big advantage is that *cat(1)* can't create any socket or pass descriptors. So in the end the usage of Pledge comes to understanding the program and to giving it the right promises.

```
@@ −66,6 +66,9 @@

setlocale(LC\_ALL, "");

if (pledge("stdio rpath", NULL) == −1)
    err(1, "pledge");

while ((ch = getopt(argc, argv, "benstuv")) !=
    −1)
    switch (ch) {
    case 'b':
```

Listing 1: pledge(2) used in a cat(1) program in OpenBSD.

As you can see in the listing 1 pledge has two arguments. The first was already describe in depth earlier. In the future it will be possible to pass as a second argument a list of paths which will be whitelisted from the Pledge sandbox. At the time of writing this article these options were not yet implemented and always end with the error message: "Invalid argument".

What is also very interesting and also quite surprising is that the author of Pledge has hard coded paths to some files in the kernel. For example, even in sandbox, you are always able to do operations like:

- *open(2)* files like */etc/localtime* and any files below */usr/share/zoneinfo*,
- *readlink(2)* may operate on */etc/malloc.conf*,
- *sysctl(3)* read-only operations like *getdomainname(3)*, *gethostname(3)*, *getifaddrs(3)*, *uname(3)*.

As motioned earlier, you can use an *exec* promise which allow you to execute another program. This is an example of promises which you should be very careful about, and which unfortunately is over used[1]. The real problem with *exec* is that promises are not inherits be the new process. If a potential attacker would exploit your program he can try then run other application from your system which can lead to a security gap.

Pledge in its assumption is very similar to the seccomp-bpf. The difference is that with pledge we have multiple predefined set of allowed/forbidden syscalls and arguments. In case of seccomp-bpf we must define such list and it is easily desynchronized after every update [3]. Pure seccomp-bpf is also very hard to use but now we can use libseccomp which simplify it a lot.

OpenBSD released pledge in 2015 and it is still considered as experimental feature and its API can change.

3 Capsicum

Capsicum is a lightweight OS capability and sandbox framework implementing a hybrid capability system model [4]. The process sandbox system's architecture in FreeBSD can be divided into two modules:

- Tight sandboxing (*cap_enter(2)*)
- Capability rights (*cap_rights_limit(2)*)

By calling *cap_enter(2)* we enter a sandbox or so called capability mode. Capability mode is inherited by all descendent processes and cannot be removed. Any attempt to access to any global namespaces (such as path names or pid namespace) will be prevented by the kernel. Capabilities in Capsicum are represented by file descriptors. In sandbox you can't create a new descriptor from global namespace, but you can still use privileges that you already have. For instance, you can't open any new files[2], but we can use capability which we have, lets say descriptor to the directory and using *openat(2)* syscall we can still open files from that directory. In Capsicum we have two ways to obtain capabilities. The first way is to get credentials before entering sandbox, so for example in the initial phase of the program we open files, create sockets and then enter sandbox to perform complicated algorithms. The second way is to obtain them from other process. File descriptors are ideal as the carrier of the capabilities, they can be inherited by a child process after fork and also if the two processes share an Unix domain socket descriptors can be passed around.

The second part of Capsicum allows us to limit the process even further. Capability rights are a special flags added to file descriptors which tells the kernel how you can use them. We can limit descriptors to be read-only, write-only, read-write but what is very interesting and very useful is that you can even specify that descriptor to be append only[3]. In the FreeBSD we have defined around 80 capability rights. For example as file specific rights we have:

- CAP_FCHMOD allows change mode (*fchmod(2)*),
- CAP_FSTAT allows getting file stats (*fstat(2)*),
- CAP_UNLINKAT allows file deletion (*unlinkat(2)*).

A full list of the Capsicum capability rights can be found in the FreeBSD *rights(4)* manual page. Capsicum was first introduced in FreeBSD 9.0. Currently, there is ongoing work to port Capsicum to Linux and DragonFlyBSD [5] [6].

3.1 Sandboxing uniq(1)

As an example of using Capsicum in practice we will use a *uniq(1)* case. First what we need to do is to understand what the program is doing. The *uniq(1)* is used to filter out repeated lines in a file. In the code we can see that our application is operating on input and output descriptors. Both of them can be file or stdout/stdin descriptors. In listing 2 we have code responsible for setting capability rights on descriptor. We are using *cap_rights_init(3)* function to initialize a structure responsible for keeping capabilities (*cap_rights_t*). Unfortunately the description of this structure is beyond this

[1] At the moment around 70 programs is using it.

[2] We can't because we trying to access path namespace.

[3] In our software at Wheel company we use such descriptor as log descriptor, if somebody would break in to the sandboxed program he can't overwrite any previous log.

article but if you are interested why do we have it in such form you can look into another article [9].

After setting up all descriptors we can enter capability mode which we see on listing 3. Some perceptive reader can notice that we not only are checking result of *cap_rights_limit(3)* or *cap_enter(2)* we also are checking *errno* value. If the FreeBSD kernel is unable to enter capability mode it will fail but if it will fail with *ENOSYS* that means that the FreeBSD kernel is compiled without Capsicum support and we want to continue execution.

```
cap_rights_t rights;
...
ifp = stdin;
ifn = "stdin";
ofp = stdout;
if (argc > 0 && strcmp(argv[0], "-") != 0)
    ifp = file(ifn = argv[0], "r");
cap_rights_init(&rights, CAP_FSTAT, CAP_READ);
if (cap_rights_limit(fileno(ifp), &rights) < 0
    && errno != ENOSYS)
    err(1, "unable to limit rights for %s", ifn
        );
cap_rights_init(&rights, CAP_FSTAT, CAP_WRITE);
if (argc > 1)
    ofp = file(argv[1], "w");
else
    cap_rights_set(&rights, CAP_IOCTL);
if (cap_rights_limit(fileno(ofp), &rights) < 0
    && errno != ENOSYS) {
    err(1, "unable to limit rights for %s",
        argc > 1 ? argv[1] : "stdout");
}
```

Listing 2: uniq(1) setting descriptor capabilities.

```
if (cap_enter() < 0 && errno != ENOSYS)
    err(1, "unable to enter capability mode");
```

Listing 3: uniq(1) entering capability mode.

4 CloudABI

CloudABI is using Capsicum to provide a runtime environment that attempts to make it easier to use the UNIX-like operating system at the core of a cloud computing platform. The idea behind this solution is to allow users to provide a set of binaries that communicate with the operating system over a secure IPC, instead of forcing them into using full machine vitalization or OS-level vitalization. [7] CloudABI we can split in two modules:

- secure libc,
- runtime environment.

ClaudABI contains a libc with removes all functions that are considered insecure (for example *strcpy(3)*, *gets(3)* etc.) [8]. Its also has a special runtime environment which forces the user to use Capsicum because before entering the main function we are already in the capability mode. If we want to open a socket or open files in this environment we need to define a special YAML file with will be parsed before executing a user code and after entering main function descriptors will be available for use.

The author hopes that CloudABI will be used more frequently in the Cloud platforms.

5 Comparison

As we can see Capsicum and Pledge present a totally different approach to the security problem. In the case of pledge we are managing access to a global namespace by limiting syscalls and arguments to those syscall. This approach allows us to say that we have access to a path namespace and we can also say that we have read-only access to that namespace.

In the Capsicum approach we are managing privileges using file descriptors. This allows us to limit every single descriptor. We can choose which descriptor are read-only, append-only and so on. This approach is more fine-grained and gives us flexibility in bigger programs.

An advantage of pledge over Capsicum is the simplicity in usage of base system programs. As we have shown in this article it is much easier to sandbox *wc(1)* or *cat(1)* using pledge then using Capsicum.

But to be fair the author must point out that at some stage of Pledge architecture are not appear elegant. One of them is hard coded paths in kernel, this at some point can be a double edge sword for the project. In Capsicum for example you will need to run *localtime(3)* at least once before entering sandbox to allow a program read and cache the */etc/localtime* file. In pledge you don't have that problem because you can read that file at any time. This simplifies the code of application that we will sandbox, but moving userland paths to the kernel doesn't feel right.

The second problem where the author feel that there is room for improvement in pledge is escaping from sandbox by using *execv(2)*. When you enter sandbox you shouldn't be able to exit it at any point. We also don't gain anything by keeping this behavior.

Author also thinks that in the case of pledge there is inflexibility in that approach. If you will read a code of sandboxed application you recognize that all of them are done matching to one template. Author is not sure if Pledge will be so useful in bigger problems where the program need for example reload its configuration, or in which we cannot draw a clear line between initialization phase and the main part. We can get impression that Pledge was create mainly with the idea to sandbox the base system.

6 Casper

6.1 Introduction

Traditionally in the capability environment we have a negotiation between an unprivileged process with privileged process to access some additional rights. In the beginning programmers did that manually. They fork in program and in child they entered to sandbox and left privilege processes to serve new data to the sandboxed one. We can analyze this approach for example in *rwhod(8)* daemon in the FreeBSD base system. But developers noticed that with this solution there is a big amount of code which would be needed to be rewritten multiple times for different programs. To solve this problem the Casper daemon was introduced.

6.2 Initial architecture

The idea behind Casper was to allow in simple way to obtain more rights in a sandboxed process. If such process need for example access to the DNS it can create such services (before entering capability mode), limit that services and then use it safely in sandbox. What also was a very important goal was that API presented by Casper was as close to original as possible. Firstly Casper was created to solve challenges created by programs sandboxed with Capsicum, but now creators believes that Casper can be precious also for other projects.

When Casper was first implemented it was a daemon in an operating system. It was divided to four parts:

- Casper daemon (called *casperd(8)*),
- services (located in */libexec/casper*),
- list of services (located in */etc/casper*),
- IPC library (called libcapsicum(3)).

When *casperd(8)* was started it created a Unix domain socket for communication with other process and zygote process. The zygote process is a lightweight process which closes all additional descriptors and uses minimal memory. The process can connect to Casper and ask it to create new services, if it possible a zygote will be used to create it. The whole process of transforming zygote into services is shown in figure 1. To connect to *casperd(8)* we are using *cap_init(3)* function. The advantage of having Casper asa daemon was also that root user of FreeBSD could build a system with Capsicum and Casper support but he could also decided if he wants to use Casper in runtime. If for some reason he decided to resigned from using Casper he could just kill the *capserd(8)*. Disadvantage of this approach is that Casper is one in entire system so it was a single point of failure.

Services is the essence of Casper. After getting service it will allow you to use functions which are forbidden in the sandbox. At the moment in FreeBSD 11-CURRENT, we have five official services.

- system.dns allows the use of gethostbyname(3), gethostbyname2(3), gethostbyaddr(3), getaddrinfo(3), getnameinfo(3),
- system.grp provides a getgrent(3)-compatible API,
- system.pwd provides a getpwent(3)-compatible API,
- system.random allows obtaining entropy from /dev/random,
- system.sysctl provides a sysctlbyname(3)-compatible API.

Every service also implements also limitation functions. For example we can limit *system.pwd* to some command like we can allow to use *getgrent(3)* but prevent it to use *setgrent(3)*.

Next module of *casperd(8)* was list of services. The idea was that we can install new services from ports while compiling new software with Casper support. New services would be automaticly added to that list which would simplify the search for deamon.

The process needs to communicate some how with *casperd(8)*. The is why *libcaspicum* was introduced. This IPC library was providing simple API for communicate with daemon and services which were in the base system. As mentioned earlier Casper was initially recognized with Capsicum this is the reason how this library got its name. Similar libraries were planned to be installed from ports with new services to display interfaces for them.

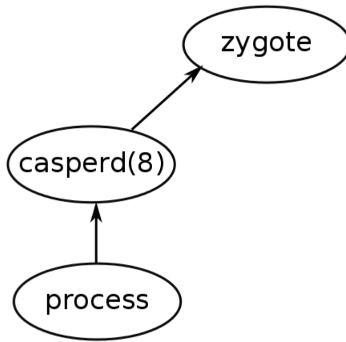
6.3 Sandboxing ping(8)

As a real life example of using Casper, we will use *ping(8)* from the FreeBSD base system. First we need to understand what kind of problem we want to solve. What we want to achieve is that after argument parsing all other operations are in sandbox. After an analysis of the source code we see that descriptor *IPPROTO_ICMP* is open very early, even before parsing arguments, this operation is not permitted in Capsicum, so lets leave it in the initial phase. So we will find only one such thing - dns access.

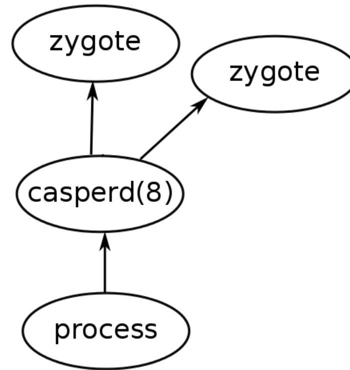
The actual code of sandbox we can find on listing 4. So first we open the connection to Casper like we described it in the previous chapter. Next we need to choose what services we need in our cases - dns. We will not create more services so we can close the connection to

(a) In the most common cases Casper daemon starts with the operating system.

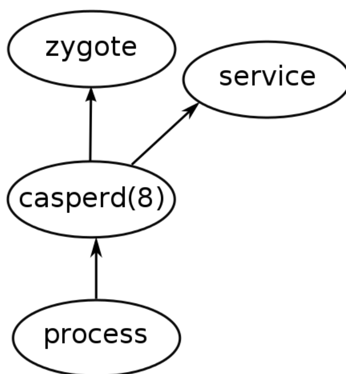
(b) When the daemon is started it spawns a new zygote process and starts to listing on an interface.



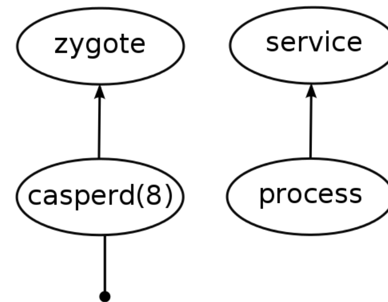
(c) A process asks Casper to create a new service using this interface.



(d) Daemon clones zygote process.



(e) Casper transforms one of the zygotes into the required service.



(f) Daemon sends a file descriptor to communicate with the service. The services exists as long as there is a connection to it.

Figure 1: The life cycle of a zygote in the Casper daemon.

Casper. The final step of configuration of Casper service is to limit its access, we set ADDR and NAME limitation this allow us to use *cap_gethostbyname* and *cap_gethostbyaddr* and also limiting family to AF_INET only. If we don't set any limitations all access to the dns namespace will be allowed. To complete sandboxing we need to change *gethostbyaddr* call to the *cap_gethostbyaddr* what was shown in listing 4.

So as we can see the API of the Casper calls is quite similar to the original API, but we still need to separate the usage of the libc and Casper one. Actual sandboxing of *ping(8)* is a little more complicated for example we use *gethostbyname* is done in multiple functions and we are shown only one.

```
static cap_channel_t *
capdns_setup(void)
{
    cap_channel_t *capcas, *capdnsloc;
    const char *types[2];
    int families[1];

    capcas = cap_init();
    if (capcas == NULL) {
        warn("unable to contact casperd");
        return (NULL);
    }
    capdnsloc = cap_service_open(capcas, "
        system.dns");
    /* Casper capability no longer needed. */
    cap_close(capcas);
    if (capdnsloc == NULL)
        err(1, "unable to open system.dns
            service");
    types[0] = "NAME";
    types[1] = "ADDR";
    if (cap_dns_type_limit(capdnsloc, types, 2)
         < 0)
        err(1, "unable to limit access to
            system.dns service");
    families[0] = AF_INET;
    if (cap_dns_family_limit(capdnsloc,
        families, 1) < 0)
        err(1, "unable to limit access to
            system.dns service");

    return (capdnsloc);
}
```

Listing 4: Configuring casper in ping(8).

```
if (capdns != NULL)
    hp = cap_gethostbyname2(capdns, source,
        AF_INET);
else
    hp = gethostbyname2(source, AF_INET);
if (!hp)
    errx(EX_NOHOST, "cannot resolve %s: %s",
        source, hstrerror(h_errno));
```

Listing 5: Casper usage in ping(8).

6.4 Problems

From the very beginning Casper was a research project. The approach with the daemon created some problems that we didn't expect and we discover them when we tried to sandbox more applications and create more services. The problems that we could identify are:

- different cwd from the original process,
- different uid, gid and groups,
- different MAC labels,
- different descriptor table,
- different routing table,
- different umask,
- different *cpuset(1)*.

The first problem is that we have a different current working directory (cwd). *casperd(8)* will be run from the root directory but the sandboxed program can be launched from any directory. If we try to use a service which covers path namespace, at some point it will do *open(2)* or a similar function. The relative path in the program and in Capser will be different, so daemon will try to open a different file. This problem even in the *casperd(8)* could be partially solved. For example we could send actual the path to Casper or what would be even better, a file descriptor which contains cwd and instead of doing open do *openat(2)*. When we open the file its too late to send the current working directory (because we are in the sandbox so we don't have access to current working directory) so we can only do this when we are creating a service, but in that case changing the working directory will not work.

Another problem with the *casperd(8)* implementation is that it is running from the root user or casperd user but not from our user. So potentially if we try for example, to create a file, Casper will create a file with a file with a different user thraeni the program is running from. This problem is easily solved because UNIX domain sockets allow us to receive other process credentials and we can do *setuid(2)*, *setgid(2)* and *setgroups(2)* to change that in our service. Only the root can do those functions so Casper must be launched from that user.

The final problem which we will discuss thoroughly is the problem with the descriptor table and the process substitution. To use the process substation in bash we use the notation of *<(list)* or *>(list)*. The process list is run with its input or output connected to the some file in */dev/fd*. That location contains a reference to the process descriptor table and at this point it is obviously that it is not inherited by the Casper. If we pass such a path to

Casper instead of cloning one of the descriptor which the process is the owner it would duplicate one of the Casper.

All other capabilities like MAC labels, umask, routing table and *cpuset(1)* are inherited from the original process of Casper and not from the process that we are running. In every cases our sandboxed program can run unexpectedly and illogically. Unfortunately we don't have any method to pass them in a convincing way to the Casper daemon.

6.5 New implementation

One of our goal is to support path namespace. Unfortunately because of problems described in the chapter 6.4 it is impossible to see all of these edge cases. We decided to reimplement the Casper.

We resigned from the Casper daemon approach and decided to go with a fork approach which at the time of the first implementation was impossible to achieve. One of the foundations of the approach are the process descriptors.

Process descriptors, in brief, are the same as file descriptors but instead of referring to the files or socket they refer to the process. Instead of using pids we can use a descriptor to manage the process. The process descriptors use the same table as ordinary descriptors. We can also use process descriptors in similar ways as normal descriptors for example we can pass them over a UNIX domain or use them in the *kqueue(1)* [4]. Another big advantage of the process descriptors is that they will not return any status in functions like *wait(2)* as long as we will not give them explicitly, the pid which the process descriptor have. This means that our fork from the process will almost be invisible in a sandboxed program and will not interact with a function which waits for any process [4], so using our new Casper will not change the behavior of program. What is also worth of mentioning is that the child process will live as long as there will be existing is at least one process descriptor pointing to it. We can destroy the process descriptor by calling close on it.

We reduce the amount of the modules from the original implementation. In the new architecture called *libcasper* we have two modules:

- libcasper,
- services (located in */lib/casper*),

We also noted that the split between the IPC and actual services is not practical. In the previous model, the program needed to be recompiled to linked to the IPC library, so we don't see any advantages of keeping them separate. Another advantage is that we need to link the IPC library to use services, in the *casperd(8)* initialize phase we could make a mistake of opening the services to which we didn't have IPC. When we link the *libcasper* and libraries repressing services they will automatically be registered in the Casper library. We achieved that thanks to the library constructor.

Another difference is when we call *cap_init(2)* we are using *pdfork(2)* to create Casper in the child we create zygote and then jumping to the internal loop of the *libcasper*. All communication is still done by the UNIX domain sockets.

From our research we managed to solve most of the problems described in the chapter 6.4. All of the resources, capabilities and limits are inherited from the original process that we run. What continues to be a problem and what developers should keep in mind is that if we changed some of those capabilities it will not affect Casper automatically. A recommended solution for that make alli the necessary changes before initializing *libcasper*.

What we didn't managed to save is flexibility about using Casper and not using it. In the previous version the administrator could shutdown *capserd(8)* and use programs, and after while run it again. Now if we don't want to use Casper you need to recompile the program. However the author thinks that this isn't a big loss, because we also removed a single point of failure.

We also achieved that all the life cycles of the zygotes described at figure 1 weren't changed. What is also interesting no API changes was made between versions. *libcasper* is not in base system at the time of writing this which is scheduled to be published on 12, 2016.

6.6 Future goals

What we want to achieve in the nearest future:

- lower the bar for the new Casper and Capsicum consumer,
- publish the system.filesystem or similar services which allow to interact with path namespace,
- remove the need of checking if Casper is available.

References

[1] Theo de Raddt, pledge() a new mitigation mechanism, hackfest 2015 http://www.openbsd.org/papers/hackfest2015-pledge/mgp00001.html

[2] pledge restrict system operations, OpenBSD Manual http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/pledge.2

[4] wait(2) with -1 argument for example.

[3] Seccomp, Mozilla https://wiki.mozilla.org/Security/Sandbox/Seccomp

[4] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, Kris Kennaway, Capsicum: Practical Capabilities for UNIX, 2010: https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf

[5] Robert Watson, Cambridge Computer Laboratory Web page, 2014 https://www.cl.cam.ac.uk/research/security/capsicum/

[6] Linux repository with ongoing work around Capsicum, https://github.com/google/capsicum-linux

[7] Ed Schouten, CloudABI, BSDCan, 2015 https://www.bsdcan.org/2015/schedule/attachments/330_2015-06-13%20CloudABI%20at%20BSDCan.pdf

[8] CloudABI libc repository, https://github.com/NuxiNL/cloudlibc

[9] Pawel Jakub Dawidek, Mariusz Zaborski, Sandboxing with Capsicum, December 2014 https://www.usenix.org/system/files/login/articles/login_dec14_03_dawidek.pdf